An Implementation of Ekiden on the Oasis Network

Oasis Protocol Project

***Abstract*—Smart contracts are applications that execute on blockchains. Today they manage billions of dollars in value and motivate visionary plans for pervasive blockchain deployment. While smart contracts inherit the availability and other security assurances of blockchains, however, they are impeded by blockchains' lack of *confidentiality* and poor *performance*.**

We present Ekiden, a system that addresses these critical gaps by combining blockchains with Trusted Execution Environments (TEEs). Ekiden leverages a novel architecture that separates consensus from execution, enabling efficient TEE-backed confidentiality-preserving smart contracts and high scalability. We describe our prototype implementation with Tendermint as the consensus layer.

Another contribution of this paper is that we systematically identify and treat the pitfalls arising from harmonizing TEEs and blockchains. Treated separately, both TEEs and blockchains provide powerful guarantees, but hybridized, they engender new attacks. For example, in naïve designs, privacy in TEE-backed contracts can be jeopardized by forgery of blocks, a seemingly unrelated attack vector. We believe the insights learned from Ekiden will prove to be of broad importance in hybridized TEE-blockchain systems.

I. Introduction

Smart contracts are protocols that digitally enforce agreements between or among distrusting parties. Typically executing on blockchains, they enforce trust through strong integrity assurance: Even the creator of a smart contract cannot feasibly modify its code or subvert its execution. Smart contracts have been proposed to improve applications across a range of industries, including finance, insurance, identity management, and supply chain management.

Smart contracts inherit some undesirable blockchain properties. To enable validation of state transitions during consensus, blockchain data is public. Existing smart contract systems thus *lack confidentiality or privacy*: They cannot safely store or compute on sensitive data (e.g., auction bids, financial transactions). Blockchain consensus requirements also hamper smart contracts with *poor performance* in terms of computational power, storage capacity, and transaction throughput. Ethereum, the most popular decentralized smart contract platform, is used almost exclusively today for technically simple applications such as tokens, and can incur costs vastly (eight orders of magnitude) more than ordinary cloud-computing environments. In short, the *application complexity of smart contracts today is highly constrained*. Without critical performance and confidentiality improvements, smart contracts may fail to deliver on their transformative promise.

Researchers have explored cryptographic solutions to these challenges, such as various zero-knowledge proof systems [34] and secure multiparty computation [72]. However, these approaches have significant performance overhead and are only applicable to limited use cases with relatively simple computations. A more performant and general-purpose option is use of a TEE.

A TEE provides a fully isolated environment that prevents other software applications, the operating system, and the host owner from tampering with or even learning the state of an application running in the TEE. For example, Intel Software Guard eXtensions (SGX) provides an implementation of a TEE. The Keystone-enclave project [3] aims to provide an open-source TEE design.

This document is an updated version of the Ekiden paper [16], the precursor to this work. Many sections are retained, where our current and ongoing work are consistent with the original paper.

A key observation driving our system design is that TEEs and blockchains have complementary properties. On the one hand, a blockchain can guarantee strong availability and persistence of its state, whereas a TEE cannot guarantee availability (as the host can terminate TEEs at its discretion), nor can it reliably access the network or persistent storage. On the flip side, a blockchain has very limited computation power, and must expose its entire state for public verification, whereas a TEE incurs minimal overhead compared with native computation, and offers verifiable computation with confidential state via remote attestation. Thus it appears appealing to build hybrid protocols that take advantage of both.

Harmonizing TEEs with blockchains, though, is a challenge. Subtle pitfalls arise when the two are naïvely glued together.

One such pitfall arises from a fundamental limitation of TEEs: A malicious host can arbitrarily manipulate their scheduling and I/O. Consequently, TEEs might terminate at any point, posing the risk and challenge of lost and/or conflicting state. This problem is exacerbated by the fact that the so-called trusted timer in TEEs (SGX, in particular) can in fact only provide a "no-earlier-than" notion of time, because a malicious host can also delay the clock read (a message transmitted over the bus). Thus, while it's tempting to use a blockchain to checkpoint a TEE's state (e.g. [33]), the lack of a reliable timer renders it tricky for a TEE to ascertain an up-to-date view of the blockchain. As we'll show later, naïve state-checkpointing protocols open up rewinding attacks (Section III). Another interesting and dangerous consequence is that seemingly unrelated attack vectors come into play. For example, the confidentiality of TEE-protected content could be jeopardized by integrity attacks against the blockchain: e.g., an attacker could circumvent a privacy budget enforced by a TEE by providing a forged blockchain to rewind its execution and sent it arbitrarily many queries. Other challenges include tolerating compromised TEEs, supporting robust and

consistent failover when TEEs crash, and key management for enclaves. We systematically identify and treat each of these pitfalls in this paper.

Following the above design principles, we present Ekiden, a system for highly performant and confidentiality-preserving smart contracts. To the best of our knowledge, Ekiden is the first confidentiality-preserving smart contract system capable of thousands of transactions per second. The key to this achievement is a secure and principled combination of blockchains and trusted hardware. Ekiden combines any desired underlying blockchain system (permissioned or permissionless) with TEE-based execution. Anchored in a formal security model expressed as a cryptographic ideal functionality [14], Ekiden's principled design supports rigorous analysis of its security properties.

Ekiden adopts an architecture where *computation* is separated from *consensus*. Ekiden uses *compute nodes* to perform smart contract computation over private data off chain in TEEs, then attest to their correct execution on chain. The underlying blockchain is maintained by *validator nodes*, which need not use trusted hardware. Ekiden is agnostic to consensus-layer mechanics, only requiring a blockchain capable of validating remote attestations from compute nodes. Ekiden can thus scale consensus and compute nodes independently according to performance and security needs.

By operating compute nodes in TEEs, Ekiden imposes minimal performance overhead relative to an ordinary (e.g., cloud) computing environment. In this way, we avoid the computational burden and latency of on-chain execution. TEE-based computation in Ekiden provides confidentiality, enabling efficient use of powerful cryptographic primitives that a TEE is known to emulate, such as functional encryption [21] and black-box obfuscation [51], and also provides a trustworthy source of randomness, a major acknowledged difficulty in blockchain systems [13].

To address the availability and network security limitations of TEEs, Ekiden supports on-chain checkpointing and (optional) storage of contract state. Ekiden thereby supports safe interaction among long-lived smart contracts across different trust domains. To address potential TEE failures, such as side channel attacks, we propose mitigations to preserve integrity and limit data leakage (Section III-A). Assuming blockchain integrity, users need not trust smart contract creators, miners, node operators or any other entity for liveness, persistence, confidentiality, or correctness. Ekiden thus enables self-sustaining services that can outlive any single node, user, or development effort.[1]

Technical challenges and contributions. Our work on Ekiden addresses several key technical challenges:

- *Formal security modeling:* While intuitively clear, the desired and achievable security properties required for Ekiden are challenging to define formally. We express the full range of security requirements of Ekiden in terms of an ideal functionality $\mathcal{F}_{\text{Ekiden}}$. We outline a security proof in the Universal Composability (UC) framework that shows that the Ekiden protocol matches $\mathcal{F}_{\text{Ekiden}}$ under concurrent composition.
- *A principled approach for hybridized TEE-blockchain systems:* We systematically enumerate the fundamental pitfalls arising from fusing blockchains and TEEs and offer general techniques for overcoming them. Further, we show that by appealing to cryptographic ideal functionalities, these techniques can be applied in a principled, provably secure, and performant way that we believe can be generalized to a broad range of hybridized TEE-blockchain systems.
- *Performance:* The blockchain is likely to be a performance bottleneck of a TEE-blockchain hybrid system. We provide optimization that minimize the use of blockchain without degrading security: We show that they realize the same $\mathcal{F}_{\text{Ekiden}}$ functionality as the unoptimized protocol.

[1]Our system name Ekiden refers to this property. "Ekiden" is a Japanese term for a long-distance relay running race.

II. Background

a) Smart Contracts and Blockchains: Blockchain-based smart contracts are programs executed by a network of participants who reach agreement on the programs' state. Existing smart contract systems replicate data and computation on all nodes in the system so that individual node can verify correct execution of the contract. Full replication on all nodes provides a high level of fault tolerance and availability. Smart contract systems such as Ethereum [20] has demonstrated their utility across a range of applications.

However, several critical limitations impede wider adoption of current smart contract systems. First, on-chain computation of fully replicated smart contracts is inherently expensive. For example in August 2017, it cost $26.55 to add 2 numbers together one million times in an Ethereum smart contract [20], a cost roughly 8 orders of magnitude higher than in AWS EC2 [59]. Furthermore, current systems offer no privacy guarantees. Users are identified by pseudonyms. As numerous studies have shown [57], [46], [49], [58], pseudonymity provides only weak privacy protection. Moreover, *contract state and user input must be public* in order for miners to verify correct computation. Lack of privacy fundamentally restricts the scope of applications of smart contracts.

b) Trusted Hardware with Attestation: A key building block of Ekiden is a *trusted execution environment* (TEE) that protects the confidentiality and integrity of computations, and can issue proofs, known as *attestations*, of computation correctness. Ekiden is implemented with Intel SGX [4], [27], [45], a specific TEE technology, but we emphasize that it may use any comparable TEE with attestation capabilities, such as the ongoing effort Keystone-enclave [3] aiming to realize open-source secure TEE hardware. We now offer brief background on TEEs, with a focus on Intel SGX.

Intel SGX provides a CPU-based implementation of TEEs—known as *enclaves* in SGX—for general-purpose computation. A host can instantiate multiple TEEs, which are not only isolated from each other, but also from the host.

Code running inside a TEE has a protected address space. When data from a TEE moves off the processor to memory, it is transparently encrypted with keys only available to the processor hardware and microcode. Thus the operating system, hypervisor, and other users cannot access the enclave's memory. The SGX memory encryption engine also guarantees data integrity and prevents memory replay attacks [25]. Intel SGX supports attested execution, i.e., it is able to prove the correct execution of a program, by issuing a *remote attestation*, a digital signature, using a private key known only to the hardware, over the program and an execution output. Remote attestation also allows remote users to establish encrypted and authenticated channels to an enclave [4]. Assuming trust in the hardware, and Intel, which authenticates attestation keys, it is infeasible for any entity other than an SGX platform to generate any attestation, i.e., attestations are existentially unforgeable.

However, attested execution realized by trusted hardware isn't perfect. For example, SGX alone cannot guarantee availability: a malicious host can terminate enclaves or drop messages arbitrarily. Even an honest host could accidentally lose state (e.g. when power cycles). The weak availability of SGX poses a fundamental challenge to the design of Ekiden. Also, the current SGX implementation is vulnerable to side-channel attacks [68], [53]. Ekiden is compatible with existing defenses [10], [51], [40], [66], [56]. We discuss side-channel resistance in Section III-A.

III. Technical Challenges in TEE-blockchain hybrid systems

Before diving into the specifics of Ekiden, we first describe and address the fundamental pitfalls that arise when harmonizing TEEs and blockchains. The solutions serve as building blocks of the Ekiden protocol, and we believe the insights learned from Ekiden will prove to be of broad importance in hybridized TEE-blockchain systems.

A. Tolerating TEE failures

Although designed to execute general purpose programs, trusted hardware is not a panacea. Here we analyze the limitations of TEEs and their impact on TEE-blockchain hybrid protocols.

a) Availability failures: Trusted hardware in general cannot ensure availability. In the case of SGX, a malicious host can terminate enclaves, and even an honest host could lose enclaves in a power cycle. A TEE-blockchain system must tolerate such host failures, ensuring that crashed TEEs can at most delay execution.

Our high-level approach is to treat TEEs as expendable and interchangeable, relying on the blockchain to resolve any conflicts resulting from concurrency. To ensure that any particular TEE is easily replaced, TEEs are *stateless*, and any persistent state is stored by the blockchain. We discuss later how TEEs can also keep soft state across invocations as a performance optimization, but we emphasize that the techniques in Ekiden ensure that losing such state at any point does not affect security.

b) Side channels: Although TEEs aim to protect confidentiality, recent work has uncovered data leakage via side-channel attacks. Existing defenses are generally application- and attack-specific (e.g., crypto libraries avoid certain data-dependent operations [10]); generalizing such protections remains challenging. Thus, Ekiden largely defers protections to the application developer.

Even though there is perhaps no definitive and practical panacea to all side-channel attacks, it is still desirable to limit the impact of compromised TEEs and provide graceful degradation in the face of small-scale compromise. Our approach is to compartmentalize both spatially and temporally. We design critical components in Ekiden, such as the key manager, against a strong adversarial model, allowing an attacker to break the confidentiality of a small fraction of TEEs, and limit the access to the key manager from other components. We also employ proactive key rotation [26] to confine the purview of a leaked key. Key management is fundamental to the availability of a TEE-blockchain system, as discussed below.

c) Timer failures: TEEs in general lack trusted time sources. In the case of SGX, although a trusted relative timer is available, the communication between enclaves and the timer (provided by an off-CPU component) can be delayed by the OS [31], [30]. Moreover server-grade Intel CPUs offer no support for SGX timers at the time of writing. Thus a TEE-blockchain hybrid protocol must minimize reliance on the TEE timer.

Our approach is to design protocols that do not require TEEs to have a current view of a blockchain. Specifically, instead of requiring a TEE to distinguish stale state from current state (without a synchronized clock, there is no definitive countermeasure to a network adversary delaying messages from the blockchain), our techniques rely on the blockchain to proactively reject any update based on a stale input state (a hash of which is included in the update).

The missing timer also makes it hard for TEEs to verify that an item has been persisted in the blockchain, i.e. to establish "proofs of publication," as coined by [33]. However [33] doesn't consider threats caused by lack of trustworthy time in TEEs—e.g., injection of old, fake, easily minable blocks—that are critical in PoW-based blockchains. One of our contributions is a general, time-based proof-of-publication protocol that is secure against network adversary delaying clock read, as we now briefly explain.

B. Proof of Publication for PoW blockchains

In order to leverage blockchains as persistent storage, a TEE must be able to efficiently verify that an item has been stored in the blockchain. For permissioned blockchains, such a proof can consist of signatures from a quorum of validator nodes. To establish proofs of publication for PoW-based blockchains, TEEs must be able to validate new blocks. As noted in [17], a trusted timer is needed to defend against an adversary isolating an enclave and presenting an invalid subchain. Unfortunately, timing sources over secure channels (e.g. SGX timers) cannot guarantee a bounded response time, as discussed above. To

work around this limitation, we leverage the confidentiality of TEEs so that an attacker delaying a timer's responses cannot prevent an enclave from successfully verifying blockchain contents. Our solution can even work without SGX timers given trust in, e.g. TLS-enabled NTP servers. Due to lack of space, we relegate our proof-of-publication protocol for PoW blockchains to Section V-A.

C. Key management in TEEs

A fundamental limitation of using a blockchain to persist TEE state is the lack of confidentiality. We showed previously how to avoid this problem by encryption. This, however, leads to another problem: how can one persist the encryption keys?

Generally the method is to replicate keys across multiple TEEs. However, the flip side is the challenge of minimizing the key exfiltration risk in the face of confidentiality breach (e.g. via side-channel attacks). There is in general a fundamental tension between exposure risk and availability: A higher replication factor means not only better resiliency to state loss, but also a larger attack surface. Therefore the tradeoff and achievable properties would depend on the threat model.

Since there is perhaps no definitive and practical full-system side-channel mitigation, our approach is to design the key manager against a stronger adversarial model where the attacker is allowed to break the confidentiality of a small fraction of TEEs, and limit the access from other components. We outline the key management protocol in Section V-B.

D. Atomic delivery of execution results

In blockchain systems, ensuring the atomicity of executions, namely either both executions e_1, e_2 finish or none of them, has been a fundamental problem, as exemplified by work on atomic cross-chain swaps [7]. A similar but more complicated problem arises in TEE-blockchain hybridization.

For a general stateful TEE-blockchain protocol, TEE execution yields two messages: m_1, which delivers the output to the caller, and m_2, which delivers the state update to the blockchain, both via adversarial channels. We emphasize that it is critical to enforce **atomic delivery** of the two messages, i.e. both m_1 and m_2 are delivered or the system has become permanently unavailable. m_1 is delivered when the caller receives it. The new state m_2 is delivered once accepted by the blockchain. Rejected state update are not considered delivered.

To see the necessity of atomic delivery, consider possible attacks when it's violated, i.e., when only one of the two messages is delivered. First, if only the output m_1 is delivered, a *rewind attack* becomes possible. Since TEE cannot tell whether an input state is fresh, an attacker can provide stale states to resume a TEE's execution from an old state. This enables grinding attacks against randomized TEE programs. An attacker may repeatedly rewind until receiving the desired output. Another example is that rewinding could defeat budget-based privacy protection, such as differential privacy. On the other hand, if only the state update m_2 is delivered, the user risks permanent loss of the output, as it might be impossible to reproduce the same output with the updated state.

We specify the atomic delivery protocol in Section V-C.

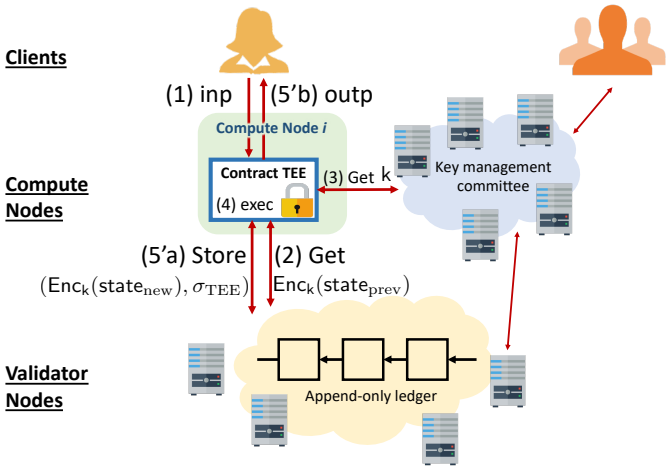

Fig. 1. Simplified overview of Ekiden architecture and workflow. Clients send inputs to confidentiality-preserving smart contracts, which are executed within a TEE at any compute node. The blockchain stores encrypted contract state. See Section IV-B for an overview.

IV. Overview of Ekiden

In this section, we provide an overview of the design and security properties of Ekiden.

A. Motivation

As an example to motivate our work, consider a credit scoring application—an example we implement and report on in Section VII-A. Credit scores are widely used by lenders, insurers, and others to evaluate the creditworthiness of consumers. Despite its considerable revenue ($10.8B in 2017 [29]), the credit reporting industry in the U.S. is concentrated among a handful of credit bureaus [29]. Such centralization creates large single points of failure and other problems, as highlighted by a recent data breach affecting nearly half the US population [9].

Blockchain-based decentralized credit scoring is thus an attractive and popular alternative. Bloom [38], for example, is a startup offering a credit scoring system on Ethereum. Their scheme, however, only supports a static credit scoring algorithm that omits important private data and cannot support predictive modeling. Such applications are bedeviled by two critical limitations of current smart contract systems: (1) A lack of *data confidentiality* needed to protect sensitive consumer records (e.g., loan-service history for credit scoring) and the proprietary prediction models derived from them and (2) A failure to achieve the *high performance* needed to handle global workloads.

To support large-scale, privacy-sensitive applications like credit scoring, it is essential to meet these two requirements while preserving the *integrity* and *availability* offered by blockchains—all without requiring a trusted third party. Ekiden offers a confidential, trustworthy, and performant platform that achieves precisely this goal for smart contract execution.

B. Ekiden Overview

Conceptually, Ekiden realizes a secure execution environment for rich user-defined smart contracts. An Ekiden contract is a deterministic stateful program. Without loss of generality, we assume contract programs take the form $(\mathsf{outp}, \mathsf{st}_{\text{new}}) := \mathsf{Contract}(\mathsf{st}_{\text{old}}, \mathsf{inp})$, ingesting as input a previous state st_{old} and a client's input inp, and generating an output outp and new state st_{new}. In this section, we describe Ekiden in this simplified form, and we will present the system in detail in an upcoming paper.

Once deployed on Ekiden, smart contracts are endowed with strong confidentiality, integrity and availability guarantees. Ekiden achieves these properties with a hybrid architecture combining trusted hardware and the blockchain. Figure 1 depicts the architecture of a simplified version of Ekiden and a workflow of Ekiden smart contracts. As it shows, there are three types of entities in Ekiden: clients, compute nodes and validator nodes.

- **Clients** are end users of smart contracts. In Ekiden, a client can create contracts or execute existing ones with secret input. In either case, clients delegate computation to compute nodes (discussed below). We expect clients to be lightweight, allowing both mobile and web applications to interact with contracts.
- **Compute nodes** process requests from clients by running the contract in a contract TEE and generating attestations proving the correctness of state updates. Anyone with a TEE-enabled platform can participate as a compute node, contributing to the liveness and scalability of the system.
- **Key manager nodes** run a distributed protocol in key management TEEs to manage keys used by contract TEEs. A contract TEE reaches out to the key management committee to create or retrieve keys. We describe key management in Section V-B.
- **Validator nodes** maintain a distributed append-only ledger, i.e. a blockchain, by running a consensus protocol. Contract state and attestations are persisted on this blockchain. Validator nodes are responsible for checking the validity of state updates using TEE attestations, as we discuss below.

C. Workflow

We now sketch the contract creation and request execution workflow, providing further details on Figure 1. The detailed formal protocol is presented in Section VI-B.

For simplicity, we assume a client has a priority list of compute nodes to use. In practice, a coordinator can be employed to facilitate compute node discovery and load balancing. We denote a client as $\mathcal{P}$ and a compute node as Comp.

a) Contract creation: When creating a contract, $\mathcal{P}$ sends a piece of contract code $\mathsf{Contract}$ to Comp. Comp loads $\mathsf{Contract}$ into a TEE (called contract TEE hereafter), and starts the initialization. The contract TEE creates a fresh contract id cid, obtains fresh $(\mathsf{pk}^{\text{in}}_{\mathsf{cid}}, \mathsf{sk}^{\text{in}}_{\mathsf{cid}})$ pair and $\mathsf{k}^{\text{state}}_{\mathsf{cid}}$ from the key management committee and generates an encrypted initial state $\mathsf{Enc}(\mathsf{k}^{\text{state}}_{\mathsf{cid}}, \vec{0})$ and an attestation σ_{TEE}, proving the correctness of initialization and that $\mathsf{pk}^{\text{in}}_{\mathsf{cid}}$ is the corresponding public key for contract cid. Finally, Comp obtains a proof of the correctness of σ_{TEE} by contacting the attestation service (detailed below); this proof and σ_{TEE} are bundled into a "certified" attestation π. Comp then sends $(\mathsf{Contract}, \mathsf{pk}^{\text{in}}_{\mathsf{cid}}, \mathsf{Enc}(\mathsf{k}^{\text{state}}_{\mathsf{cid}}, \vec{0}), \pi)$ to validator nodes. The full protocol for contract creation is specified in the "create" call of $\mathbf{Prot}_{\text{Ekiden}}$ (Fig. 2). Validator nodes verify π before accepting $\mathsf{Contract}$, the encrypted initial state, and $\mathsf{pk}^{\text{in}}_{\mathsf{cid}}$ as valid and placing it on the blockchain.

b) Request execution: The steps of request execution illustrated in Fig. 1 are as follows:

(1) To initiate the process of executing a contract cid with input inp, $\mathcal{P}$ first obtains $\mathsf{pk}^{\text{in}}_{\mathsf{cid}}$ associated with the contract cid from the blockchain, computes $\mathsf{inp}_{\text{ct}} = \mathsf{Enc}(\mathsf{pk}^{\text{in}}_{\mathsf{cid}}, \mathsf{inp})$ and sends to Comp a message $(\mathsf{cid}, \mathsf{inp}_{\text{ct}})$, as specified in Lines 8-11 of $\mathbf{Prot}_{\text{Ekiden}}$.

(2) Comp retrieves the contract code and the encrypted previous state $\mathsf{st}_{\text{ct}} = \mathsf{Enc}(\mathsf{k}^{\text{state}}_{\mathsf{cid}}, \mathsf{st}_{\text{old}})$ of contract cid, from the blockchain, and loads st_{ct} and inp_{ct} into a TEE and starts the execution, as specified in Line 30-33 of $\mathbf{Prot}_{\text{Ekiden}}$.

(3-4) From the key management committee, the contract TEE obtains $\mathsf{k}^{\text{state}}_{\mathsf{cid}}$ and $\mathsf{sk}^{\text{in}}_{\mathsf{cid}}$, with which it decrypts st_{ct} and inp_{ct} and executes, generating an output outp, a new encrypted state $\mathsf{st}'_{\text{ct}} = \mathsf{Enc}(\mathsf{k}^{\text{state}}_{\mathsf{cid}}, \mathsf{st}_{\text{new}})$, and an signature π proving correct computation, as specified in Line 7-13 of the TEE Wrapper (Fig. 6).

(5a, 5b) Finally, Comp and $\mathcal{P}$ conduct an atomic delivery protocol which delivers outp to $\mathcal{P}$ and $(\mathsf{st}'_{\text{ct}}, \pi)$ to the validator nodes. We defer the detail of atomic delivery to Section V-C. Briefly, Step 5a and Step 5b in Fig. 1 are executed atomically, i.e. outp is revealed to $\mathcal{P}$ if and only if $(\mathsf{st}'_{\text{ct}}, \pi)$ is accepted by validator nodes. Validator nodes verify π before accepting the new state as valid and placing it on the blockchain.

A key distinction between Ekiden and existing smart contract platforms (e.g. Ethereum [20]) is Ekiden decouples request execution from consensus. In Ethereum, request execution is replicated by all nodes in the network to reach consensus, rendering the entire network as slow as a single node. Whereas in Ekiden, request is only executed by K compute nodes for some small K (e.g. in Figure 1, we set $K = 1$) and validator nodes just verify K proofs of correct execution without repeating the execution.

In our implementation, a proof of correct execution takes the form of a signature π. Specifically, a compute node Comp obtains π as follows. Suppose the execution on Comp results in an output st'_{ct} and an attestation σ_{TEE} (a signature [12] over the contract code and st'_{ct}). Comp then sends σ_{TEE} to the Intel Attestation Service (IAS), which verifies σ_{TEE} and replies with $\pi = (b, \sigma_{\text{TEE}}, \sigma_{\text{IAS}})$, where $b \in \{0, 1\}$ indicates the validity of σ_{TEE} and σ_{IAS} is a signature over b and σ_{TEE} by IAS. π is then submitted to the validator nodes as a proof of correctness for st'_{ct}. As π is just a signature, validator nodes need neither trusted hardware nor to contact the IAS to verify it.

D. Ekiden Security Goals

Here we summarize the security goals of Ekiden. Briefly, Ekiden aims to support execution of general-purpose contracts while enforcing the following security properties:

Correct execution: Contract state transitions reflect correct execution of contract code on given state and inputs.

Consistency: At any time, the blockchain stores a single sequence of state transitions consistent with the view of each compute node.

Secrecy: During a period without any TEE breach, Ekiden guarantees that contract state and inputs from honest clients are kept secret from all other parties. Additionally, Ekiden is resilient to some key-manager TEEs being breached.

Graceful confidentiality degradation: Should a confidentiality breach occur in a computation node (as opposed to a key-manager node), Ekiden provides forward secrecy and reasonable isolation from the affected TEEs. Specifically, suppose a confidentiality breach happens at t. The attacker can at most access the history up to $t - \Delta$ where Δ is a system parameter. Moreover, a compromised TEE can only affect a subset of contracts.

Non-goals: Ekiden does *not* prevent contract-level leakage (e.g. through covert channels, bugs or side channels). Thus contract developers are responsible for ensuring that no secret is revealed through public output, and that the contract is free of bugs and side channels. We discuss supported mitigation in Section VI-D.

E. Assumptions and Threat Model

a) TEE: Recent work demonstrates that the confidentiality of SGX enclaves may be compromised via side-channel attacks. In light of this threat, we assume the adversary can compromise the confidentiality of a small fraction of TEEs. As noted above, the impact depends on whether the breaches affect key-manager or computation nodes. We assume that TEE hardware is otherwise correctly implemented and securely manufactured.

b) Blockchain: Ekiden is designed to be agnostic to the underlying consensus protocol. It can be deployed atop any blockchain implementation as long as the requirements specified below are met.

We assume the blockchain will perform prescribed computation correctly and is always available. In particular, Ekiden relies on validator nodes to verify attestations. We further assume the blockchain provides an efficient way to construct proofs of item inclusion on the blockchain, i.e., proofs of publication, as discussed in Section III-B.

c) Threat Model: All parties in the system must trust Ekiden and TEE. We assume the adversary can control the operating system and the network stack of all but one compute nodes. On controlled nodes, the adversary can reorder messages and schedule processes arbitrarily. We assume the attacker can compromise the confidentiality of a small fraction (e.g. $f\%$) of TEEs. The adversary observes global network traffic and may reorder and delay messages arbitrarily.

The adversary may corrupt any number of clients. Clients need not execute contracts themselves and do not require trusted hardware. We assume honest clients trust their own code and platform, but not other clients. Each contract has an explicit policy dictating how data is processed and requests are serviced. Ekiden does not (and cannot reasonably) prevent contracts from leaking secrets intentionally or unintentionally through software bugs.

V. Building blocks

Before diving to protocol details, we first present key building blocks of the Ekiden protocol, addressing the general technical challenges in TEE-blockchain systems, as reviewed in Section III.

A. Proof of Publication

We now present a proof of publication protocol for permissionless blockchains. Please refer to Section III-B for background and motivation. A proof of publication is an interactive proof between a verifier $\mathcal{E}$, in the form of a contract TEE, and a untrusted prover $\mathcal{P}$. The high level idea is to only give $\mathcal{P}$ a limited amount of time to publish the message in a block within a subchain of sufficient difficulty so that an adversary cannot feasibly forge it. The protocol is formally specified in Fig. 7. We give text description below so the formal specification is not required for understanding.

$\mathcal{E}$ stores a recent checkpoint block CB from the blockchain, from which a difficulty $\delta(CB)$, e.g. the number of leading zeroes in the block nonce, can be calculated. $\mathcal{E}$ will emit an (attested) version of CB to any requesting client, enabling the client to verify CB's freshness. Given a valid recent CB, $\mathcal{E}$ can verify new blocks based on $\delta(CB)$, assuming the difficulty is relatively stationary. (For simplicity in our analysis here, we assume constant difficulty, but our analysis can be extended under an assumption of bounded difficulty variations.)

To initiate publication of m, $\mathcal{E}$ calls the timer to get a timestamp t_1. As discussed, $\mathcal{E}$ may receive t_1 after a delay. After receiving t_1 (maybe at a time later than t_1), $\mathcal{E}$ generates a random nonce r and requires the prover to publish (m, r). Upon receiving a proof $\pi_{(m,r)}$ (a subchain containing (m, r)) from $\mathcal{P}$, $\mathcal{E}$ calls the timer again for t_2. Let n_c to be the number of confirmations in (m, r), τ be the expected block interval (an invariant of the blockchain), and ϵ be a multiplicative *slack* factor that accounts for variation in the time to generate blocks, which is a stochastic process. E.g., $\epsilon = 1.5$ means that production of $\pi_{(m,r)}$ is allowed to be up to 1.5 times slower than expected on the main chain. $\mathcal{E}$ accepts $\pi_{(m,r)}$ only if $t_2 - t_1 < n_c \times \tau \times \epsilon$.

Setting ϵ to a high value reduces the probability of false rejections (i.e., rejecting proofs from an honest $\mathcal{P}$ when the main chain growth was unluckily slow during some timeframe). However, a high ϵ also increases the possibility of false acceptance, i.e. accepting a forged subchain. For any $\epsilon > 1$, it is possible to require a large enough n_c so that the probability of a successful attack becomes negligible. However, a large n_c

means that an honest $\mathcal{P}$ needs to wait for a long time before $\mathcal{P}$ can obtain the output, may affecting the user experience.

For example, for a powerful attacker with 25% hash power (roughly the largest mining pool known to exist in Bitcoin and Ethereum at the time of writing), setting $n_c = 80$ and $\epsilon = 1.6$ means the attacker needs an expected 2^{112} hashes to forge a proof of publication,[2] while an honest proof will be rejected with probability 2^{-19}. Similar block-synchronization techniques and analysis are used in the recently proposed Tesseract TEE-based cryptocurrency exchange [7].

It is easy to see that delaying the timer's responses does not give the attacker more time than $t_2 - t_1$. Delaying timestamp t_1 shrinks this apparent interval of time, disadvantaging the attacker. $\mathcal{E}$'s checkpoint block can be updated with the same protocol, by publishing an empty message. Note that once a message is successfully published by a TEE, other TEEs can obtain the proof via secure channels established by attestations, saving the cost of repeating the protocol.

B. Key Management

Each Ekiden contract is associated with a set of keys, including a symmetric key for state encryption and a key pair to encrypt client input. Here we discuss the generation, distribution, and rotation of these keys. We discuss two key manager implementations: (i) a simpler protocol with key generation done in a TEE and replicated for availability and (ii) a more complex protocol that uses a distributed key generation protocol among a committee and additionally derives rotating short-term keys. In Ekiden multiple key managers can be available, with contract deployers choosing which key manager to be the authority of each contract's keys.

1) Replicated TEEs: In this protocol, a key manager node uses a TEE program to generate a master secret, and it replicates it across other nodes in the key management committee.

a) Adversarial model: We consider an adversary that can compromise the key manager nodes, but that cannot break the confidentiality of the TEEs.

In addition, we assume there is at least one node online and not compromised at any time so that the availability of keys are retained. In practice, participation can be motivated by economic rewards and penalties. We leave the incentive design for future work.

b) Desired properties: Ideally, a key management protocol should satisfy the following properties:

- *Confidentiality*: The adversary (within our model) cannot exfiltrate contracts' decryption keys or secrets used to generate them.
- *Availability*: An honest contract TEE can always access decryption keys.

c) Initialization: The first node in the key management committee generates a random master secret s. A node KM_a joining the committee obtains a copy of the master secret by requesting it from a node already in the committee with TEE KM_b. KM_a and KM_b mutually attest that the other is running the key management TEE. These remote attestations use nonces to ensure freshness rather than timestamps, so that they do not require a trusted view of time to validate. Then, KM_b sends s to KM_a over an encrypted channel.

d) Generating contract keys: To get the contract key for a contract with ID cid, a compute node Comp first establishes a secure channel and authenticates itself with a key manager node's TEE. Once verified that Comp is indeed executing cid, the key manager node's TEE computes $\mathsf{k}_{\mathsf{cid}} = \mathsf{H}(\mathsf{cid})^s$, where $\mathsf{H}(\cdot)^k$ is a keyed hash function with key k, and sends $\mathsf{k}_{\mathsf{cid}}$ to Comp.

2) Distributed key generation:

a) Adversarial model: We consider an adversary stronger than the adversary from Section V-B1. In addition to being able to compromise key manager nodes, the adversary can also break the confidentiality, e.g., via side-channel attacks, of some fraction (e.g. $f\%$) of the TEEs. The exact value of f depends on the deployment and enrollment model. f can be a very low value if enrollment is limited to well-managed nodes, e.g., ones hosted by capable and reputable organizations. But when deployed in a more open environment, f needs to be reasonably high. We assume the participating hosts have (at least partially) Sybil-resistant identities. One way to achieve this is to require a security deposit to join the protocol.

However, the adversary is weaker than the adversary from Section V-B1 in harming availability. We assume there are sufficiently many (e.g. more than $2f\%$ of) participants online at any time so that the availability of keys are retained. Similarly, a system of incentives can motivate this participation, and we leave the design to future work.

b) Desired properties: Since decryption keys are eventually revealed to a contract TEE, which itself may also be compromised, actively used keys (i.e. hot keys) must be short-lived, derived from a less-exposed long-term master secret. In addition to the properties desired of the replicated TEEs protocol, this protocol should satisfy an additional property:

- *Forward secrecy*: If a short-term key is compromised at time t, it cannot be used to decrypt messages encrypted before $t - \Delta$, for some system parameter Δ.

c) Preliminaries: Below we outline a key management protocol that satisfies the above requirements. We first review the building blocks, including distributed key generation (DKG) protocols and distributed pseudo-random functions (PRFs).

A **distributed key generation (DKG) protocol** (e.g. [23]) allows a set of N parties to generate unbiased, random keys. The outcome of a run of a DKG protocol is a secret s, but shared among parties using a secret-sharing scheme (typically Shamir's).

Informally, **pseudo-random functions (PRF)** are a collection of functions $\mathcal{F} = \{f_s\}_{s \in S}$, such that for a random index $s \leftarrow\!\!\$\ S$, $f_s(\cdot)$ is indistinguishable from a random function.

Naor *et al.* [50] introduce distributed PRFs, which are such that parties with shares of s can evaluate $f_s(\cdot)$ without

[2] as the time of writing, it takes roughly 2^{73} hashes to mine a Bitcoin block.

reconstructing s. Specifically, let G be a Schnorr group. Let $\mathsf{H} : \{0,1\}^* \to G$ be a hash function; [50] shows that $f_s(x) = \mathsf{H}(x)^s$ is a family of PRF.

Suppose s is shared among parties using a (k, n)-secret sharing scheme. To evaluate $f_s(x)$, party i simply computes and outputs $y_i = \mathsf{H}(x)^{s_i}$, computed with its share s_i. After collecting at least $k+1$ of $\{y_i\}$, one can derive $f_s(x)$ by polynomial interpolation in the exponent:

$$f_s(x) = \mathsf{H}(x)^S = \mathsf{H}(x)^{\sum_{i\in A} S_i \lambda_i} = \prod_{i\in A} y_i^{\lambda_i}$$

where λ_i are Lagrange coefficients $\lambda_i = \prod_{j\neq i} \frac{-j}{i-j}$.

d) Key management committees and long-term keys: Assuming Sybil-resistant identities, we can sample N nodes from the participants to form a key management committee (KMC). N is a system parameter. When initializing a contract c, KMC runs the DKG protocol to generate a long term key k_c, so that k_c is secret-shared among KMC members using a $(\lceil fN \rceil, N)$-secret sharing scheme. Previous work on proactive secret sharing (e.g. [26], [60]) can be used to periodically rotate the committee without changing the secret. [60] also allows a committee to be dynamically expanded.

e) Generating short-term keys: Suppose short-term keys expire every epoch. To get the short-term key for contract c at epoch t, a compute node Comp first establishes secure channels and authenticates itself with members in KMC. Once verified that Comp is indeed executing c, each KMC member i computes $\mathsf{k}_{c,t,i} = \mathsf{H}(t)^{\mathsf{k}_c^i}$ and sends $\mathsf{k}_{c,t,i}$ to Comp. After collecting $f+1$ outcomes from $A \subseteq$ KMC, Comp can construct the short-term key for epoch t by $\mathsf{k}_{c,t} = \prod_{i\in A} \mathsf{k}_{c,t,i}^{\lambda_i}$ where λ_i are Lagrange coefficients.

f) Breach isolation: We proactively quarantine confidentiality breaches by enforcing a privacy budget for each compute node. For this to work, we assume contract TEEs have unforgeable host identities. For example, SGX remote attestation uses Intel's *Enhanced Privacy ID* (EPID) scheme, which associates a host with such an identity. Key-manager nodes maintain a counter κ_{Comp} for each compute node Comp to record the number of queries. The counter is reset along with epoch advancement. Key-manager nodes fulfill a query only if $\kappa_{\mathsf{Comp}} < \kappa$ for some system parameter κ. With this in place, no matter how many TEEs a breached compute node spawns, it can at most obtain κ keys. In practice, requests to a depleted honest compute node can be redirected to other nodes, resulting in only a modest overhead.

g) Committee rotation: The key management committee may need to change over time. Maram et al.'s concurrent work proposes algorithms to manage a shared secret in a committee where members can join and leave over time [42]. An enhancement to this key manager protocol could incorporate such an algorithm to persist a long term keys k_c, while short term keys generated by the committee at the time of the request.

C. Atomic Delivery

Recall that TEE execution yields two messages: m_1, which delivers the output to the caller, and m_2, which delivers the state update to the blockchain, both via adversarial channels. As discussed in Section III-D, it is critical to enforce atomic delivery of the two messages, i.e. both m_1 and m_2 are delivered or the system has become permanently unavailable. Now we specify a protocol for atomic delivery.

Assuming a secure communication channel between a TEE and the calling client $\mathcal{P}$ (which in practice can be constructed with remote attestation), we realize atomic delivery of m_1 and m_2 (defined above) via the following two-phase protocol: To initiate atomic delivery, TEE obtains a fresh key k from the key manager and sends an attested $m_1^c = \mathsf{Enc}(\mathsf{k}, m_1)$ to $\mathcal{P}$ over a secure channel. Once $\mathcal{P}$ acknowledges receipt of m_1^c, the TEE sends m_2 to the blockchain. Finally, after seeing π_{m_2}, a proof of publication for m_2, TEE sends k to $\mathcal{P}$. Under the constraint that at least one node in the compute committee remains online to perform this step, $\mathcal{P}$ can thus decrypt m_1.

The above protocol realizes atomic delivery. On the one hand, as a TEE can ascertain the delivery of m_2 by verifying π_{m_2}, k is revealed *only if* m_2 is delivered. On the other hand, *if* m_2 has been delivered, k will be released eventually because at least one TEE is available and the key management protocol ensures that the availability of k.

VI. Protocol Details and Security Proof

In this section, we specify $\mathbf{Prot}_{\text{Ekiden}}$, the protocol realization of Ekiden. It aims to realize a Universal Composability (UC) [14] ideal functionality $\mathcal{F}_{\text{Ekiden}}$ that we defer to Appendix A for lack of space and encourage the reader to consult. Looking ahead, $\mathbf{Prot}_{\text{Ekiden}}$ UC-realizes $\mathcal{F}_{\text{Ekiden}}$.

A. Preliminary and Notation

a) Attested Execution: To formally model attested execution on trusted hardware, we adopt the ideal functionality $\mathcal{G}_{\text{att}}$ defined in [55]. Informally, a party first loads a program prog into a TEE with an "install" message. On a "resume" call, the program is run on the given input, generating an output outp along with an attestation $\sigma_{\text{TEE}} = \Sigma_{\text{TEE}}.\mathsf{Sig}(\mathsf{sk}_{\text{TEE}}, (\mathsf{prog}, \mathsf{outp}))$, a signature under a hardware key sk_{TEE}. The public key pk_{TEE} can be obtained from $\mathcal{G}_{\text{att}}$.getpk(). See [55] for details.

In practice it's useful to allow a TEE to output data that is not included in attestation. We extend $\mathcal{G}_{\text{att}}$ slightly to allow this: if a TEE program prog generates a pair of output $(\mathsf{outp}_1, \mathsf{outp}_2)$, the attestation only signs outp_1, i.e. $\sigma_{\text{TEE}} = \Sigma_{\text{TEE}}.\mathsf{Sig}(\mathsf{sk}_{\text{TEE}}, (\mathsf{prog}, \mathsf{outp}_1))$. A common pattern is to include a hash of outp_2 in outp_1, to allow parties to verify σ_{TEE} and outp_2 separately. Similar technique is used in [69].

Following the notation in [34], [66], we use contract wrappers (defined in Fig. 6) to abstract away routine functionality such as state encryption, key management, etc. A contract c augmented with the wrapper is denoted $\widehat{\mathsf{c}}$.

b) Blockchain: $\mathcal{F}_{\text{blockchain}}[\mathsf{succ}]$ (given in Appendix A) defines a general-purpose append-only ledger implemented by common blockchain protocols (formally defined in Figure 4 in the Appendix). The parameter succ is a function that specifies the criteria for a new item to be added to the storage, modeling

the notion of transaction validity. We retain the append-only property of blockchains but abstract away the inclusion of state updates in blocks. We assume overlay semantics that associate blockchain data with id's. In addition to read and write interfaces, $\mathcal{F}_{\text{blockchain}}$ provides a convenient interface by which clients can ascertain whether an item is included in the blockchain. In practice, this interface avoids the overhead of downloading the entire blockchain.

c) Parameterizing $\mathcal{F}_{blockchain}$: In Ekiden, the contents of storage are parsed as an ordered array of *state transitions*, defined as $\mathsf{trans}_i = (\mathsf{H}(\mathsf{st}_{i-1}), \mathsf{st}_i, \sigma_i)$, a tuple of a hash of the previous state, a new state, and proofs from the compute nodes' TEEs attesting to the correctness of a state transition. (Note that as a performance optimization, large user input—e.g. training data in an ML contract— may not be stored on chain.) Storage can be interpreted as a special initial state followed by a sequence of state transitions: $\mathsf{Storage} = ((\mathsf{Contract}, \mathsf{st}_0, \sigma_0), \{\mathsf{trans}_i\}_{i\geq 1})$.

For a state transition to be *valid*, it must extend the latest state and the attestation must verify. Formally, this is achieved by parameterizing $\mathcal{F}_{\text{blockchain}}$ with a successor function $\mathsf{succ}(\cdot,\cdot)$ such that $\mathsf{succ}(\mathsf{Storage}, (h, \mathsf{st}_{\text{new}}, \sigma_{\text{TEE}})) = \mathsf{true}$ if and only if $h = \mathsf{H}(\mathsf{st}_{\text{old}})$ where st_{old} is the latest state in $\mathsf{Storage}$ and $\Sigma_{\text{TEE}}.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma_{\text{TEE}}, (h, \mathsf{st}_{\text{new}}))$. This guarantees that at any time there is a single sequence of state transitions consistent with the view of each party, i.e. the chain of state transitions is fork-free.

B. Formal Specification of the Protocol

The Ekiden protocol is formally specified in $\mathbf{Prot}_{\text{Ekiden}}$ (Fig. 2). $\mathbf{Prot}_{\text{Ekiden}}$ relies on $\mathcal{G}_{\text{att}}$ and $\mathcal{F}_{\text{blockchain}}$, ideal functionality for attested execution and the blockchain. $\mathbf{Prot}_{\text{Ekiden}}$ also use a digital signature scheme $\Sigma(\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$, a symmetric encryption scheme $\mathcal{SE}(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ and an asymmetric encryption scheme $\mathcal{AE}(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$.

a) Sharing state keys: Each contract is associated with a set of keys. As discussed in Section V-B, contract TEEs delegate key management to key manager TEEs. In $\mathbf{Prot}_{\text{Ekiden}}$, communication with key managers is abstracted away with the keyManager function.

b) Contract creation: To create a contract in Ekiden, a client $\mathcal{P}_i$ calls the `create` subroutine of a compute node Comp with input Contract, a piece of contract code. Comp loads the Contract into a TEE and starts the initialization by invoking the "create" call. As specified in Fig. 6, the contract TEE creates a fresh contract cid, obtains $\mathsf{k}_{\mathsf{cid}}$ from the key manager, derives a $(\mathsf{pk}^{\text{in}}_{\mathsf{cid}}, \mathsf{sk}^{\text{in}}_{\mathsf{cid}})$ pair and $\mathsf{k}^{\text{state}}_{\mathsf{cid}}$, and generates an encrypted initial state st_0 and an attestation σ_{TEE}. The attestation proves the st_0 is correctly initialized and that $\mathsf{pk}^{\text{in}}_{\mathsf{cid}}$ is the corresponding public key for contract cid. The compute node Comp sends $(\mathsf{Contract}, \mathsf{cid}, \mathsf{st}_0, \mathsf{pk}^{\text{in}}_{\mathsf{cid}}, \sigma_{\text{TEE}})$ to $\mathcal{F}_{\text{blockchain}}$ and waits for an receipt. Comp returns the contract cid to $\mathcal{P}_i$, who will verify that contract cid is properly stored on $\mathcal{F}_{\text{blockchain}}$.

c) Request execution: To execute a request to contract cid, a client $\mathcal{P}_i$ first obtains the input encryption key $\mathsf{pk}^{\text{in}}_{\mathsf{cid}}$ from $\mathcal{F}_{\text{blockchain}}$. Then $\mathcal{P}_i$ calls the `request` subroutine of Comp with input $(\mathsf{cid}, \mathsf{inp}_{\mathsf{ct}})$, where $\mathsf{inp}_{\mathsf{ct}}$ is $\mathcal{P}_i$'s input encrypted with $\mathsf{pk}^{\text{in}}_{\mathsf{cid}}$ and authenticated with spk_i. Comp fetches the encrypted previous state $\mathsf{st}_{\mathsf{ct}}$ from $\mathcal{F}_{\text{blockchain}}$ and launches an contract TEE with code Contract and input $(\mathsf{cid}, \mathsf{inp}_{\mathsf{ct}}, \mathsf{st}_{\mathsf{ct}})$.

As specified in Fig. 6, if $\sigma_{\mathcal{P}_i}$ verifies, the contract TEE decrypts $\mathsf{st}_{\mathsf{ct}}$ and $\mathsf{inp}_{\mathsf{ct}}$ with keys obtained from the key manager and executes the contract program Contract to get $(\mathsf{st}_{\text{new}}, \mathsf{outp})$. To ensure the new state and the output are delivered atomically, Comp and $\mathcal{P}_i$ conduct an atomic delivery protocol as specified in Section V-C:

- First the contract TEE computes $\mathsf{outp}_{\mathsf{ct}} = \mathsf{Enc}(\mathsf{k}^{\text{out}}_{\mathsf{cid}}, \mathsf{outp})$ and $\mathsf{st}'_{\mathsf{ct}} = \mathsf{Enc}(\mathsf{k}^{\text{state}}_{\mathsf{cid}}, \mathsf{st}_{\text{new}})$, and send both and proper attestation to $\mathcal{P}_i$ in a secure channel established by epk_i.
- $\mathcal{P}_i$ acknowledges the reception by calling the `claim-output` subroutine of Comp, which triggers the contract TEE to send $m_1 = (\mathsf{st}'_{\mathsf{ct}}, \mathsf{outp}_{\mathsf{ct}}, \sigma)$ to $\mathcal{F}_{\text{blockchain}}$. σ protects the integrity of m_1 and cryptographically binds the new state and output to a previous state and a input, thus a malicious Comp cannot tamper with it.
- Once m_1 is accepted by $\mathcal{F}_{\text{blockchain}}$, the contract TEE sends the decryption of $\mathsf{outp}_{\mathsf{ct}}$ to $\mathcal{P}_i$ in a secure channel.

C. Security of $\mathbf{Prot}_{Ekiden}$

Theorem 1 characterizes the security of $\mathbf{Prot}_{\text{Ekiden}}$. A proof sketch is given in Appendix C.

Theorem 1 (Security of $\mathbf{Prot}_{\text{Ekiden}}$)**.** *Assume that $\mathcal{G}_{att}$'s attestation scheme Σ_{TEE} and the digital signature Σ are existentially unforgeable under chosen message attacks (EU-CMA), that H is second pre-image resistant, and that $\mathcal{AE}$ and $\mathcal{SE}$ are IND-CPA secure. Then $\mathbf{Prot}_{Ekiden}$ securely realizes $\mathcal{F}_{Ekiden}$ in the $(\mathcal{G}_{att}, \mathcal{F}_{blockchain})$-hybrid model, for static adversaries.*

D. Mitigating app-level leakage

While Ekiden protects within-TEE data, it is not designed to protect data at contract interfaces, i.e., data leakage resulting from the contract design. (E.g., a secret prediction model may be "extracted" via client queries [65].) Common approaches to minimizing such leakage, e.g., restricting requests based on requester identity and/or a differential-privacy budget [19], [32], require persistent counters. The monotonic counters in SGX are untrustworthy, however [43].

Ekiden instead supports stateful approaches to mitigate application-level privacy leakage by enabling persistent application state—e.g., counters, total consumed differential privacy budget, etc.—to be maintained securely on chain. Moreover, the aforementioned atomic delivery guarantee ensures that the output is only revealed if this state is correctly updated.

E. Performance Optimizations

Given an additional mechanism for revocation, a simple modification *eliminates reliance on the IAS apart from initialization*. When initialized, an enclave creates a signing key $(\mathsf{pk}, \mathsf{sk})$, and outputs pk with an attestation. Subsequently, attestations are replaced with signatures under sk. Since pk is bound to the TEE code (by the initial attestation), signatures

under sk prove the integrity of output, just as attestations do. As with other keys, (pk, sk) are managed by the key manager (c.f. Section V-B).

In Appendix D we discuss an extended version of the protocol with several other performance optimizations.

VII. Implementation

In Ekiden, applications are written on top of a *runtime*, which combines the contract wrapper and other common functionality into an enclave program. The runtime we use in our experiments has an interpreter for EVM and Web Assembly (Wasm) bytecode plus libraries to support Ethereum's account-based state model.

We use the Fortanix Enclave Development Platform [22] to build and run our enclave programs. We also implemented a compiler that automatically builds contracts into executables that can be loaded into our Ethereum compatibility runtime, which we describe in Section VII-A.

Ekiden is compatible with many existing blockchains. We have built one end-to-end instantiation, *Ekiden-BT*, with a blockchain extending from Tendermint [37], which required no changes to Tendermint.

A. Programming Model

We support a general-purpose programming model for specifying applications. An application has access to a mutable key-value store as its state, which Ekiden transparently serializes, encrypts, and synchronizes with the validator committee after contract calls. An application must be deterministic and terminate in bounded time. Within our runtime, we implemented two programming environments. In the first environment, developers can write contracts using a subset of the Rust programming language, and thus benefit from a range of open source libraries. This environment compiles the Rust source code to Web Assembly bytecode. For the second environment, we ported the Ethereum Virtual Machine (EVM), thereby supporting any contract written for the Ethereum platform. Both of these environments produce bytecode which our runtime executes in an interpreter. The interpreter isolates applications from each other, so that they cannot directly access each others' state or contract keys. Applications interpreted in this runtime runtime share an enclave, so they can call each other seamlessly.

B. Applications

We now describe several different applications we developed to show the versatility of Ekiden's programming model. Figure 3 highlights the secret state and application complexity of each contract.

a) Tokens: The most popular kind of Ethereum contract is the ERC20 token standard. Using the Ethereum port (Section VII-A), we can run existing ERC20 token contracts. Ekiden automatically provides privacy and anonymity, which the contract would not receive on the Ethereum mainnet. The secret state in the token the account balance for each user.

b) CryptoKitties: CryptoKitties [1] is an Ethereum game that allows users to breed virtual cats, which are stored on chain as ERC721 tokens [2]. Each cat has a unique set of genes that determine its appearance and therefore its value. The traits of offspring are determined by a smart contract that mixes the genes of its parents. The source code of the gene mixing contract is not publicly available: The game developers aimed to make the breeding process unpredictable.

We obtained the bytecode for the gene mixing contract from the Ethereum blockchain and executed it using our Ekiden Ethereum compatibility runtime. We verified correct behavior by reproducing real transactions from the Ethereum network.

This example demonstrates that Ekiden can execute an Ethereum contract even when source code is not available. Further, Ekiden can provide unique benefits for smart contracts requiring secrecy or unpredictability such as CryptoKitties. These properties are difficult to achieve with Ethereum. E.g., the CryptoKitties gene mixing algorithm has been reverse-engineered [71], which allows strategic players to optimize their chance of breeding cats with rare traits, thus undermining the game's ecosystem. By contrast, an Ekiden contract has access to a source of randomness in hardware and allows secret elements of a game's algorithm to be stored in encrypted state.

c) Origin: Origin [54] is a platform for building online marketplaces on top of Ethereum. We ported a demo application which allows users to list and purchase items with Ether. This application further demonstrates that development frameworks built for Ethereum can be easily used by Ekiden: the smart contracts used in the demo work without modification; we were able to integrate the rest of the demo, namely, a user-facing web server, with minor modifications. Built on Ekiden, users' transaction history in the blockchain are kept private, and transactions are confirmed faster than on Ethereum.

VIII. Related Work

Confidential smart contracts: Hawk [34] is a smart contract system that provides confidentiality by executing contracts off-chain and posting only zero-knowledge proofs on-chain. As the zero-knowledge proofs in Hawk (zk-SNARKs) incur very high computational overhead, Ekiden is significantly faster. Additionally, Hawk was designed for a single compute node (called the "manager"), and thus cannot (as designed) offer high availability. While Ekiden does require trust in the security of Intel SGX, Hawk's "manager" must be trusted for privacy. Hawk supports only a limited range of contract types, not the general functionality of Ekiden.

The idea of combining ledgers with trusted hardware for smart contract execution is briefly mentioned in Hawk and also treated in [17], [33]. [17] combines blockchain with TEE to achieve one-time programs that resemble smart contracts but only aim for a restricted functionality (one-shot MPC with N parties providing input). [33] includes a basic prototype, but omits critical system design issues; e.g., its permissionless "proof-of-publication" overlooks the technical difficulties arising from lack of trusted wall-clock time in enclaves.

$\mathbf{Prot}_{\mathbf{Ekiden}}(\lambda, \mathcal{AE}, \mathcal{SE}, \Sigma, \{\mathcal{P}_i\}_{i\in[N]})$

Clients $\mathcal{P}_i$:

Initialize: $(\mathsf{ssk}_i, \mathsf{spk}_i) \leftarrow\$ \Sigma.\mathsf{KGen}(1^\lambda)$

$(\mathsf{esk}_i, \mathsf{epk}_i) \leftarrow\$ \mathcal{AE}.\mathsf{KGen}(1^\lambda)$

On receive ("create", Contract) from environment $\mathcal{Z}$:

cid := create(Contract); assert cid initialized on $\mathcal{F}_{\text{blockchain}}$

output ("receipt", cid)

On receive ("request", cid, inp, eid) from environment $\mathcal{Z}$:

$\sigma_{\mathcal{P}_i} := \mathsf{Sig}(\mathsf{ssk}_i, (\mathsf{cid}, \mathsf{inp}))$

get $\mathsf{pk}^{\text{in}}_{\text{cid}}$ from $\mathcal{F}_{\text{blockchain}}$;

let $\mathsf{inp}_{\text{ct}} := \mathcal{AE}.\mathsf{Enc}(\mathsf{pk}^{\text{in}}_{\text{cid}}, (\mathsf{inp}, \sigma_{\mathcal{P}_i}))$

$(\mathsf{st}'_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma) := \text{request}(\mathsf{cid}, \mathsf{inp}_{\text{ct}})$

parse σ as $(\sigma_{\text{TEE}}, h_{\text{inp}}, h_{\text{old}}, h_{\text{outp}}, \mathsf{spk}_i)$

assert $\mathsf{H}(\mathsf{inp}_{\text{ct}}) = h_{\text{inp}}$; assert $\mathsf{outp}_{\text{ct}}$ is correct by verifying σ

$o :=$ claim-output$(\mathsf{cid}, \mathsf{st}'_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma, \mathsf{epk}_i)$

// retry if the previous state has been used by a parallel query

if $o = \perp$ **then** jump to the beginning of the "request" call

parse o as $(\mathsf{outp}'_{\text{ct}}, \sigma_{\text{TEE}})$

assert $\Sigma_{\text{TEE}}.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma_{\text{TEE}}, \mathsf{outp}'_{\text{ct}})$ // $\mathsf{pk}_{\text{TEE}} := \mathcal{G}_{\text{att}}.\text{getpk}()$

output $\mathcal{AE}.\mathsf{Dec}(\mathsf{esk}_i, \mathsf{outp}'_{\text{ct}})$

On receive ("read", cid) from environment $\mathcal{Z}$:

send ("read", cid) to $\mathcal{F}_{\text{blockchain}}$ and relay output

Compute Nodes Subroutines (called by clients $\mathcal{P}_i$):

On input create(Contract):

send ("install", $\widehat{\text{Contract}}$) to $\mathcal{G}_{\text{att}}$, wait for eid

send (eid, "resume", ("create")) to $\mathcal{G}_{\text{att}}$

wait for $((\mathsf{Contract}, \mathsf{cid}, \mathsf{st}_0, \mathsf{pk}^{\text{in}}_{\text{cid}}), \sigma_{\text{TEE}})$

send ("write", $(\mathsf{Contract}, \mathsf{cid}, \mathsf{st}_0, \mathsf{pk}^{\text{in}}_{\text{cid}}, \sigma_{\text{TEE}})$) to $\mathcal{F}_{\text{blockchain}}$

wait to receive ("receipt", cid)

On input request(cid, inp_{ct}):

send ("read", cid) to $\mathcal{F}_{\text{blockchain}}$ and wait for st_{ct}

// non-existing eid is assumed to be created transparently

send (eid, "resume", ("request", cid, $\mathsf{inp}_{\text{ct}}, \mathsf{st}_{\text{ct}}$)) to $\mathcal{G}_{\text{att}}$

receive $((\text{"atom-deliver"}, h_{\text{inp}}, h_{\text{old}}, \mathsf{st}'_{\text{ct}}, h_{\text{outp}}, \mathsf{spk}_i), \sigma_{\text{TEE}}, \mathsf{outp}_{\text{ct}})$

// $\sigma_{\text{TEE}} = \Sigma_{\text{TEE}}.\mathsf{Sig}(\mathsf{sk}_{\text{TEE}}, (h_{\text{inp}}, h_{\text{old}}, \mathsf{st}'_{\text{ct}}, h_{\text{outp}}, \mathsf{spk}_i))$

let $\sigma := (\sigma_{\text{TEE}}, h_{\text{inp}}, h_{\text{old}}, h_{\text{outp}}, \mathsf{spk}_i)$

return $(\mathsf{st}'_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma)$

On input claim-output(cid, $\mathsf{st}'_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma, \mathsf{epk}_i$):

send ("write", cid, $(\mathsf{st}'_{\text{ct}}, \sigma)$) to $\mathcal{F}_{\text{blockchain}}$

if receive ("reject", cid) from $\mathcal{F}_{\text{blockchain}}$ **then** : return $\perp$

send (eid, "resume", ("claim output", $\mathsf{st}'_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma, \mathsf{epk}_i$)) to $\mathcal{G}_{\text{att}}$

receive ("output", $\mathsf{outp}'_{\text{ct}}, \sigma_{\text{TEE}}$) from $\mathcal{G}_{\text{att}}$ or abort

return $(\mathsf{outp}'_{\text{ct}}, \sigma_{\text{TEE}})$

Fig. 2. Ekiden Protocol. The contract TEE program $\widehat{\text{Contract}}$ is defined in Figure 6, in Appendix A.

Application	Language	LoC	**Secret Input/Output**	**Secret State**
ERC20 Token	Solidity	68	Transfer (from, to, amount)	Account balances
CryptoKitties	EVM Bytecode	54*	Random mutations	Breeding algorithm
Origin Demo	Solidity, JS	19*	Purchase orders	Purchase history

Fig. 3. Ekiden applications. For each, we specify the implementation language, development effort (LoC), as well as secret inputs, outputs, and state. Secret inputs and outputs are only accessible to the contract and the invoking user. Secret state is only accessible to the contract. For CryptoKitties and Origin Demo, we only include LoC specific to porting, as marked by $*$.

Ekiden is also closely related to and influenced by Hyperledger Private Data Objects (PDO) [11] from Intel. PDOs use smart contracts, executed in SGX enclaves, to mediate access to data objects shared amongst mutually distrusting parties. To the best of our knowledge, PDOs target permissioned and managed settings (requiring, e.g., special-purpose validation rules), while Ekiden supports permisionless and open settings as well. This leads to key technical differences. For example, PDO uses a set of Provisioning Services to store encryption keys without worrying about availability risk, which cannot be easily realized in the Ekiden setting where churn is possible. In contrast, Ekiden uses a secret-sharing-based key management protocol that tolerates churn and allows flexible committee reconfiguration.

The Microsoft Coco Framework [47] is concurrent and independent work to port existing smart contract systems, such as Ethereum, into an SGX enclave. To the best of our knowledge, only a whitepaper containing a high-level overview has been produced. No details of a protocol or implementation have yet been released.

Blockchain transaction privacy: Ekiden's goals relate to mechanisms for enhancing transaction privacy on public blockchains. Maxwell proposed a confidential transaction scheme [44] for Bitcoin that conceals transaction amounts, but not identities. Zerocash [6] as well as Cryptonote [62], [67], Solidus [15], and Zerocoin [48] provides stronger confidentiality guarantees by concealing identities. These schemes, however, do not support smart contracts.

Privacy-preserving systems based on trusted hardware: Trusted hardware, particularly Intel SGX, has seen a wide spectrum of applications in distributed systems. M^2R [18], VC3 [61], Opaque [70] and Ohrimenko *et al.* [52] leverage SGX to offer privacy-preserving data analytics and machine learning with various security guarantees, Ryoan [28] is a distributed sandbox platform using SGX to confine privacy leakage from untrusted applications that process sensitive data. These systems do not address state integrity and confidentiality over a long-lived system. In comparison, Ekiden provides a stronger integrity and availability guarantees by persisting contract states on a blockchain.

Blockchains for verifiable computations and secure multi-

party computations: Several related works offer blockchain-based guarantees of computation integrity, but cannot guarantee privacy [41], [64], [63]. Other works have used a blockchain for fairness in MPC by requiring parties to forfeit security deposits if they abort [8], [36], [35], [5], [72], [17]. Compared to these, Ekiden can guarantee that all data can be recovered if *any* compute node remains online. TEE-based computation is also far more performant than MPC. A theoretical scheme [24] combines witness encryption with proof-of-stake blockchains to achieve one-time programs that resemble smart contracts but avoid use of trusted hardware. This scheme is regrettably even more impractical than MPC.

IX. Conclusion

Ekiden demonstrates that blockchains and trusted enclaves have complementary security properties that can be combined effectively to provide a powerful, generic platform for confidentiality-preserving smart contracts. The result is a compelling programming model that overcomes significant challenges in blockchain smart contracts. We show that Ekiden can be used to implement a variety of secure decentralized applications that compute on sensitive data.

In future work we plan to extend Ekiden to operate under a stronger threat model, leveraging techniques such as secure multi-party computation [39], [17], [5], to protect the system's more critical features, such as key management and coordination across compute nodes. Coordination can also facilitate parallelism in contract execution, merging concurrent output from multiple enclaves to obtain still higher performance from Ekiden.

Acknowledgments

We wish to thank the authors of the Ekiden paper, Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song, for their contributions to the research project leading up to this work.

We wish to thank Intel, and Mic Bowman in particular, for ongoing research discussions and generous support of a number of aspects of this work. Our discussions regarding Intel's PDO system illuminated important technical challenges in Ekiden and influenced and helped us refine its design.

We also wish to thank Iddo Bentov, Joe Near, Chang Liu, Jian Liu, and Lun Wang for their helpful feedback and discussion. We also thank Pranav Gaddamadugu and Andy Wang for their contributions to application development. This material is in part based upon work supported by the Center for Long-Term Cybersecurity, DARPA (award number N66001-15-C-4066) IC3 industry partners, and the National Science Foundation (NSF award numbers TWC-1518899 CNS-1330599, CNS-1514163, CNS-1564102, CNS-1704615, and ARO W911NF-16-1-0145). This work was also supported in part by FORCES (Foundations Of Resilient CybEr-Physical Systems), which receives support from the National Science Foundation (NSF award numbers CNS-1238959, CNS-1238962, CNS-1239054, CNS-1239166). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] "CryptoKitties - Collect and breed digital cats," https://www.cryptokitties.co/.

[2] "EIP 721: ERC-721 Non-Fungible Token Standard," https://eips.ethereum.org/EIPS/eip-721.

[3] "Keystone Project," https://keystone-enclave.github.io/.

[4] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *HASP'13*, 2013, pp. 1–7.

[5] 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.

[6] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *IEEE Symposium on Security and Privacy*, 2014.

[7] I. Bentov, Y. Ji, F. Zhang, Y. Li, X. Zhao, L. Breidenbach, P. Daian, and A. Juels, "Tesseract: Real-time cryptocurrency exchange using trusted hardware," 2017, https://eprint.iacr.org/2017/1153.

[8] I. Bentov, R. Kumaresan, and A. Miller, "Instantaneous decentralized poker," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 410–440.

[9] T. Bernard, T. Hsu, N. Perlroth, and R. Lieber, "Equifax Says Cyberattack May Have Affected 143 Million in the U.S." https://www.nytimes.com/2017/09/07/business/equifax-cyberattack.html.

[10] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *LatinCrypto*, 2012.

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

[12] E. Brickell and J. Li, "Enhanced privacy id from bilinear pairing," Cryptology ePrint Archive, Report 2009/095, 2009, https://eprint.iacr.org/2009/095.

[13] B. Bünz, S. Goldfeder, and J. Bonneau, "Proofs-of-delay and randomness beacons in Ethereum," *IEEE Security and Privacy on the Blockchain (IEEE S&B)*, 2017.

[14] R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," Cryptology ePrint Archive, Report 2000/067, 2000, https://eprint.iacr.org/2000/067.

[15] E. Cecchetti, F. Zhang, Y. Ji, A. E. Kosba, A. Juels, and E. Shi, "Solidus: Confidential distributed ledger transactions via PVORM," in *ACM CCS*, 2017.

[16] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *4th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[17] 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 *ACM CCS*, 2017.

[18] 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*, 2015.

[19] C. Dwork, "Differential privacy: A survey of results," in *International Conference on Theory and Applications of Models of Computation*. Springer, 2008, pp. 1–19.

[20] Ethereum Foundation, "Ethereum: Blockchain App Platform," https://www.ethereum.org/.

[21] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: functional encryption using Intel SGX," in *ACM CCS*, 2017.

[22] Fortanix, Inc., "Fortanix enclave development platform," 2019, https://edp.fortanix.com/.

[23] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999, pp. 295–310.

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

[25] S. Gueron, "A memory encryption engine suitable for general purpose processors." *IACR Cryptology ePrint Archive*, vol. 2016, p. 204, 2016.

[26] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or: How to cope with perpetual leakage," in *Advances in Cryptology — CRYPT0' 95*, D. Coppersmith, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 339–352.

[27] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *HASP*, 2013.

[28] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *USENIX OSDI*, 2016.

[29] IBISWorld, "Credit Bureaus & Rating Agencies in the US," http://clients1.ibisworld.com/reports/us/industry/ataglance.aspx?entid=1475.

[30] Intel, "Intel SGX platform services," https://software.intel.com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services.pdf, (Accessed on 01/29/2018).

[31] "GitHub discussion on sgx_get_trusted_time," Intel SGX SDK Developers, 9 2017, https://github.com/intel/linux-sgx/issues/161.

[32] N. M. Johnson, J. P. Near, and D. X. Song, "Practical differential privacy for SQL queries using elastic sensitivity," *CoRR*, vol. abs/1706.09479, 2017. [Online]. Available: http://arxiv.org/abs/1706.09479

[33] 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, Tech. Rep., 2017.

[34] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *IEEE Security and Privacy*, 2016.

[35] R. Kumaresan and I. Bentov, "Amortizing secure computation with penalties," in *ACM CCS*, 2016.

[36] R. Kumaresan, T. Moran, and I. Bentov, "How to use Bitcoin to play decentralized poker," in *ACM CCS*, 2015.

[37] J. Kwon, "Tendermint: Consensus without mining," 2014.

[38] J. Leimgruber and A. M. J. Backus, "Bloom protocol:decentralized credit scoring powered by Ethereum and IPFS," 27 Jan. 2018.

[39] Y. Lindell and B. Pinkas, "Secure multiparty computation for privacy-preserving data mining," *Journal of Privacy and Confidentiality*, vol. 1, no. 1, p. 5, 2009.

[40] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *IEEE Security and Privacy (S&P)*, 2015.

[41] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena, "Demystifying incentives in the consensus computer," in *ACM CCS*, 2015.

[42] S. K. D. Maram, F. Zhang, L. Wang, A. Low, Y. Zhang, A. Juels, and D. Song, "Churp: Dynamic-committee proactive secret sharing." *IACR Cryptology ePrint Archive*, vol. 2019, p. 17, 2019.

[43] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," in *USENIX Security Symposium, USENIX Security*, 2017.

[44] G. Maxwell, "Confidential values," https://people.xiph.org/~greg/confidential_values.txt, (Accessed on 01/31/2018).

[45] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *HASP*, 2013.

[46] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage, "A fistful of Bitcoins: characterizing payments among men with no names," in *Proceedings of the 2013 conference on Internet measurement conference*. ACM, 2013, pp. 127–140.

[47] Microsoft, "The Coco Framework: Technical Overview," https://github.com/Azure/coco-framework/.

[48] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin," in *IEEE Security and Privacy, S&P*, 2013.

[49] M. Möser and R. Böhme, "The price of anonymity: empirical evidence from a market for Bitcoin anonymization," *Journal of Cybersecurity*, 2017.

[50] M. Naor, B. Pinkas, and O. Reingold, "Distributed pseudo-random functions and KDCs," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999.

[51] K. Nayak, C. Fletcher, L. Ren, N. Chandran, S. Lokam, E. Shi, and V. Goyal, "Hop: Hardware makes obfuscation practical," in *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[52] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors." in *USENIX Security Symposium*, 2016, pp. 619–636.

[53] D. O'Keeffe, "SGXSpectre," 2018, https://github.com/lsds/spectre-attack-sgx.

[54] Origin Protocol, Inc., "Origin protocol," https://www.originprotocol.com/, 2018.

[55] R. Pass, E. Shi, and F. Tramer, "Formal abstractions for attested execution secure processors," Cryptology ePrint Archive, Report 2016/1027, 2016, https://eprint.iacr.org/2016/1027.

[56] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security)*, 2015.

[57] F. Reid and M. Harrigan, "An analysis of anonymity in the Bitcoin system," in *Security and privacy in social networks*. Springer, 2013, pp. 197–223.

[58] D. Ron and A. Shamir, "Quantitative analysis of the full Bitcoin transaction graph," in *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 6–24.

[59] D. Ryan, "Calculating Costs in Ethereum Contracts," https://hackernoon.com/ether-purchase-power-df40a38c5a2f.

[60] D. Schultz, B. Liskov, and M. Liskov, "MPSS: Mobile proactive secret sharing," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 4, p. 34, 2010.

[61] 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 *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 38–54.

[62] S.-F. Sun, M. H. Au, J. K. Liu, and T. H. Yuen, "Ringct 2.0: A compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency monero," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 456–474.

[63] J. Teutsch, V. Buterin, and C. Brown, "Interactive coin offerings," *URl: https://people. cs. uchicago. edu/˜ teutsch/papers/ico. pdf (visited on 11/16/2017)*, 2017.

[64] J. Teutsch and C. Reitwießner, "Truebit: a scalable verification solution for blockchains," 2017.

[65] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction APIs," in *USENIX Security Symposium*, 2016, pp. 601–618.

[66] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi, "Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[67] N. Van Saberhagen, "Cryptonote v2.0," 2013.

[68] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE Symposium on Security and Privacy, SP*, 2015, pp. 640–656.

[69] F. Zhang, I. Eyal, R. Escriva, A. Juels, and R. V. Renesse, "REM: Resource-efficient mining for blockchains," in *USENIX Security Symposium (USENIX Security)*, Vancouver, BC, 2017.

[70] 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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[71] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: Reverse engineering ethereum's opaque smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[72] G. Zyskind, O. Nathan *et al.*, "Decentralizing privacy: Using blockchain to protect personal data," in *Security and Privacy Workshops (SPW), 2015 IEEE*. IEEE, 2015, pp. 180–184.

APPENDIX

A. Supplementary Formalism

1) Ideal Blockchain: We specify the ideal functionality for a blockchain in Fig. 4.

2) Ideal functionality $\mathcal{F}_{Ekiden}$: We specify the security goals of Ekiden in the ideal functionality $\mathcal{F}_{\text{Ekiden}}$ defined in Figure 5.

$\mathcal{F}_{\text{Ekiden}}$ allows parties to create contracts and interact with them. Each party $\mathcal{P}_i$ is identified by a unique id simply denoted $\mathcal{P}_i$. Parties send messages over *authenticated channels*. To capture the allowed information leakage from the encryption, we follow the convention of [14] and parameterize $\mathcal{F}_{\text{Ekiden}}$ with a leakage function $\ell(\cdot)$. We use the standard *delayed*

$\mathcal{F}_{\text{blockchain}}[\text{succ}, \{0,1\}^* \times \{0,1\}^* \to \{0,1\}]$

Parameter: successor relation succ: $\{0,1\}^* \times \{0,1\}^* \to \{0,1\}$

On receive ("init"): Storage := $\emptyset$

On receive ("read", id): output Storage[id], or $\perp$ if not found

On receive ("write", id, inp) from $\mathcal{P}$:

  let val := Storage[id], set to $\perp$ if not found

  if succ(val, inp) = 1 **then**

    Storage[id] := val || (inp, $\mathcal{P}$); output ("receipt", id)

  else output ("reject", id)

On receive ("$\in$", id, val):

  if val $\in$ Storage[id] **then** output true **else** output false

Fig. 4. Ideal blockchain. The parameter succ defines the validity of new items. A new item can only be appended to the storage if the evaluation of succ outputs 1.

$\mathcal{F}_{\text{Ekiden}}[\ell(\cdot), \{\mathcal{P}_i\}_{i\in[N]}, \{0,1\}^*]$

Parameter: leakage function $\ell(\cdot)$, $\{\mathcal{P}_i\}_{i\in[N]}$

On receive ("init"): Storage := $\emptyset$

// Create a new contract

On receive ("create", Contract) from $\mathcal{P}_i$ for some $i \in [N]$:

  cid $\leftarrow\!\!\$$ $\{0,1\}^\lambda$

  notify $\mathcal{A}$ of ("create", $\mathcal{P}_i$, cid, Contract); block until $\mathcal{A}$ replies

  Storage[cid] := (Contract, $\vec{0}$)

  send a public delayed output ("receipt", cid) to $\mathcal{P}_i$

// Send queries to a contract

On receive ("request", cid, inp, eid) from $\mathcal{P}_i$ for some $i \in [N]$:

  notify $\mathcal{A}$ of ("request", cid, $\mathcal{P}_i$, ℓ(inp))

  (Contract, st, _) := Storage[cid]; abort if not found

  (outp, st′) := Contract($\mathcal{P}_i$, inp, st)

  let $\ell_{\text{st}} = \ell$(st)

  notify $\mathcal{A}$ of (cid, $\ell_{\text{st}'}$, ℓ(outp), eid)

  wait for "ok" from $\mathcal{A}$ and halt if other messages received

  update Storage[cid] := (Contract, st′, $\ell_{\text{st}'}$)

  send a secret delayed output outp to $\mathcal{P}_i$

// Allow public access to encrypted state

On receive ("read", cid) from $\mathcal{P}_i$ for some $i \in [N]$:

  (_, _, ℓ_{st}) := Storage[cid]; abort if not found

  send ℓ_{st} to $\mathcal{P}_i$

  if $\mathcal{P}_i$ is corrupted: send ℓ_{st} to $\mathcal{A}$

Fig. 5. The ideal functionality of Ekiden.

output terminology [14] to model the power of the network adversary. Specifically, when $\mathcal{F}_{\text{Ekiden}}$ sends a delayed output outp to $\mathcal{P}$, this means that outp is first sent to the adversary $\mathcal{A}$ and forwarded to $\mathcal{P}$ after acknowledgement by $\mathcal{A}$. If the message is secret, only the allowed amount of leakage (i.e., that specified by the leakage function) is revealed to $\mathcal{S}$.

A Contract is a user-provided program. Each smart contract is associated with a piece of persistent storage where the contract code and st can be stored. The storage is public; therefore $\mathcal{F}_{\text{Ekiden}}$ allows any party, including $\mathcal{A}$, to read the storage content. The information leakage through such reading is also defined by the leakage function ℓ.

Users can send queries to $\mathcal{F}_{\text{Ekiden}}$ to execute the contract code with user-provided input. The execution of a contract will result in a secret output (denoted outp) returned to the invoker and a secret transition to a new contract state (denoted

On input ("create", Contract) **Contract TEE wrapper** $\widehat{\text{Contract}}$

  cid := H(Contract)

  $(\text{pk}^{\text{in}}_{\text{cid}}, \text{sk}^{\text{in}}_{\text{cid}})$:= keyManager("input key")

  $\text{k}^{\text{state}}_{\text{cid}}$:= keyManager("state key")

  $\text{st}_0 = \mathcal{SE}.\text{Enc}(\text{k}^{\text{state}}_{\text{cid}}, \vec{0})$

  return (Contract, cid, state_0, $\text{pk}^{\text{in}}_{\text{cid}}$)

On input ("request", cid, inp_{ct}, st_{ct}):

  // retrieve $\text{sk}^{\text{in}}_{\text{cid}}$, $\text{k}^{\text{state}}_{\text{cid}}$ from a key manager as above

  $(\text{inp}, \sigma_{\mathcal{P}_i})$:= $\mathcal{AE}.\text{Dec}(\text{sk}^{\text{in}}_{\text{cid}}, \text{inp}_{\text{ct}})$

  assert $\text{Vf}(\sigma_{\mathcal{P}_i}, \text{spk}_i, (\text{cid}, \text{inp}))$ // spk_i is publicly known

  st_{old} := $\mathcal{SE}.\text{Dec}(\text{k}^{\text{state}}_{\text{cid}}, \text{st}_{\text{ct}})$

  $\text{st}_{\text{new}}, \text{outp}$:= $\text{Contract}(\text{st}_{\text{old}}, \text{inp}, \text{spk}_i)$

  st'_{ct} := $\mathcal{SE}.\text{Enc}(\text{k}^{\text{state}}_{\text{cid}}, \text{st}_{\text{new}})$

  // initiate atomic delivery

  $\text{k}^{\text{out}}_{\text{cid}}$:= keyManager("output key")

  outp_{ct} := $\mathcal{SE}.\text{Enc}(\text{k}^{\text{out}}_{\text{cid}}, \text{outp})$

  let $h_{\text{inp}} = \text{H}(\text{inp}_{\text{ct}})$, $h_{\text{old}} = \text{H}(\text{st}_{\text{ct}})$, $h_{\text{outp}} = \text{H}(\text{outp}_{\text{ct}})$

  return (("atom-deliver", $h_{\text{inp}}, h_{\text{old}}, \text{st}'_{\text{ct}}, h_{\text{outp}}, \text{spk}_i$), outp_{ct})

On input ("claim output", st'_{ct}, outp_{ct}, σ, epk_i):

  parse σ as $(\sigma_{\text{TEE}}, h_{\text{inp}}, h_{\text{old}}, h_{\text{outp}}, \text{spk}_i)$

  assert $\text{H}(\text{outp}_{\text{ct}}) = h_{\text{outp}}$

  send ("$\in$", cid, $(\text{st}'_{\text{ct}}, \sigma)$) to $\mathcal{F}_{\text{blockchain}}$

  receive true from $\mathcal{F}_{\text{blockchain}}$ or abort

  $\text{k}^{\text{out}}_{\text{cid}}$:= keyManager("output key")

  outp := $\mathcal{SE}.\text{Dec}(\text{k}^{\text{out}}_{\text{cid}}, \text{outp}_{\text{ct}})$

  return ("output", $\mathcal{AE}.\text{Enc}(\text{epk}, \text{outp})$)

Fig. 6. Contract TEE wrapper.

st′), equivalent intuitively to black-box contract execution (modulo leakage). Although any party may send messages to the contract, the contract code can enforce access control based on the calling pseudonym passed to the contract.

a) Corruption model: $\mathcal{F}_{\text{Ekiden}}$ adopts the standard corruption model of [14]. $\mathcal{A}$ can corrupt any number of clients, and up to all but one contract executors. When $\mathcal{A}$ corrupts a TEE (or similarly a party), $\mathcal{A}$ sends the message ("corrupt", eid) to $\mathcal{F}_{\text{Ekiden}}$. If a query includes an invalid TEE id, $\mathcal{F}_{\text{Ekiden}}$ aborts if instructed by $\mathcal{A}$. Otherwise the ideal functionality ignores eids, which are included in $\mathcal{F}_{\text{Ekiden}}$ only as a technical requirement to ensure interface compatibility with $\mathbf{Prot}_{\text{Ekiden}}$, given below.

3) Contract TEE wrapper: The contract TEE wrapper $\widehat{\text{Contract}}$ is specified in Fig. 6.

B. Proof of Publication

The protocol for proof of publication is specified in Fig. 7.

C. Proof of Main Theorem

Here we give our proof of Theorem 1, given in Section VI.

We prove that $\mathbf{Prot}_{\text{Ekiden}}[\lambda, \mathcal{AE}, \mathcal{SE}, \Sigma, \{\mathcal{P}_i\}_{i\in[N]}]$ UC-realizes the ideal functionality $\mathcal{F}_{\text{Ekiden}}[\lambda, \ell, \{\mathcal{P}_i\}]$ with respect to a leakage function $\ell(x)$ that only reveals the length of x, i.e. $\ell(x) = 0^{|x|}$. In the protocol, $\ell(\cdot)$ is realized with IND-CPA encryption schemes.

Parameters: **Proof of Publication of m between verifier $\mathcal{E}$ and prover $\mathcal{P}$**

n_c: publication of m needs at least n_c confirmation

CB : a recent checkpoint block

$\delta(CB)$: difficulty of CB

τ: expected block interval of main chain

ϵ: slackness factor

Verifier $\mathcal{E}$ (a contract TEE):

$t_1 \leftarrow$ TEE.timer()

$r \leftarrow\$ \{0,1\}^\lambda$

send (m, r) to $\mathcal{P}$

receive $\pi_{(m,r)} = (CB, B_1, \cdots, B_n)$ from $\mathcal{P}$

$t_2 \leftarrow$ TEE.timer()

if $\pi_{(m,r)}$ is not a valid chain, output false

let $B_i \in \pi_{(m,r)}$ be the block that contains (m, r), output false if $\nexists B_i$

if B_i has less than n_c confirmation, i.e. $n - i < n_c$, output false

if any $B \in \pi_{(m,r)}$ has a lower difficulty than $\delta(CB)$, output false

if $t_2 - t_1 < (n - i) \times \tau \times \epsilon$: output true and update checkpoint $CB = B_n$

else : output false

Prover $\mathcal{P}$:

On receive (m, r) from $\mathcal{E}$:

send (m, r) to the blockchain, denote the including block B_i

send a subchain from CB to B_{i+n_c} (inclusive) to $\mathcal{E}$

Fig. 7. Proof of Publication

Proof. Let $\mathcal{Z}$ be an environment and $\mathcal{A}$ be a "dummy adversary" [14] who simply relays messages between $\mathcal{Z}$ and parties. To show that $\mathbf{Prot}_{\text{Ekiden}}$ UC-realizes $\mathcal{F}_{\text{Ekiden}}$, we specify below a simulator Sim such that no environment can distinguish an interaction between $\mathbf{Prot}_{\text{Ekiden}}$ and $\mathcal{A}$ from an interaction with $\mathcal{F}_{\text{Ekiden}}$ and Sim, i.e. Sim satisfies

$$\forall \mathcal{Z}, \text{EXEC}_{\mathbf{Prot}_{\text{Ekiden}}, \mathcal{A}, \mathcal{Z}} \approx \text{EXEC}_{\mathcal{F}_{\text{Ekiden}}, \text{Sim}, \mathcal{Z}}.$$

a) Construction of Sim*:* Sim generally proceeds as follows: if a message is sent by an honest party to $\mathcal{F}_{\text{Ekiden}}$, Sim emulates appropriate real world "network traffic" for $\mathcal{Z}$ with information obtained from $\mathcal{F}_{\text{Ekiden}}$. If a message is sent to $\mathcal{F}_{\text{Ekiden}}$ by a corrupted party, Sim extracts the input and interacts with the corrupted party with the help of $\mathcal{F}_{\text{Ekiden}}$. We provide further details on the processing of specific messages.

(1) Contract creation:

- If $\mathcal{P}_i$ is honest, Sim obtains $(\mathcal{P}_i, \mathsf{cid}, \mathsf{Contract})$ from $\mathcal{F}_{\text{Ekiden}}$ and emulates an execution of the "create" call of $\mathbf{Prot}_{\text{Ekiden}}$.
- If $\mathcal{P}_i$ is corrupted, Sim extracts Contract from $\mathcal{Z}$. On behalf of $\mathcal{P}_i$, Sim sends ("create", Contract) to $\mathcal{F}_{\text{Ekiden}}$ and instructs $\mathcal{F}_{\text{Ekiden}}$ to deliver the output.
- In both cases, Sim simulates the interaction between $\mathcal{F}_{\text{blockchain}}$ and $\mathcal{G}_{\text{att}}$, on behalf of the adversary or honest parties.

(2) Query execution:

Case 1: When an *honest* party $\mathcal{P}_i$ is given input ("request", cid, inp, eid) by $\mathcal{Z}$, Sim works as follows:

- Upon receiving $(\mathsf{cid}, \mathcal{P}_i, \ell(\mathsf{inp}))$ from $\mathcal{F}_{\text{Ekiden}}$, Sim queries the "read" interface of $\mathcal{F}_{\text{Ekiden}}$ to obtain the dummy state (i.e. a random string with the same length as the real state) of cid, denoted s. Sim computes $c_{\mathsf{inp}} = \mathsf{Enc}(\mathsf{pk}^{\text{in}}_{\mathsf{cid}}, \vec{0})$ with length $\ell(\mathsf{inp})$, and emulates a "resume" message to $\mathcal{G}_{\text{att}}$ with input ("request", cid, c_{inp}, s) on behalf of $\mathcal{P}_i$.
- Upon receiving $\ell_{\mathsf{st}'}$ and $\ell(\mathsf{outp})$ from $\mathcal{F}_{\text{Ekiden}}$, Sim computes $c = \mathsf{Enc}(\mathsf{k}^{\text{out}}_{\mathsf{cid}}, 0^{|\mathsf{outp}|})$ and emulates a message $((\text{"atom-deliver"}, \mathsf{H}(c_{\mathsf{inp}}), \mathsf{H}(s), \ell_{\mathsf{st}'}, \mathsf{H}(c), \mathsf{spk}_i), \sigma_{\text{TEE}}, c)$ from $\mathcal{G}_{\text{att}}$ to $\mathcal{P}_i$.
- Sim proceeds by emulating the interaction between $\mathcal{F}_{\text{blockchain}}$ and $\mathcal{G}_{\text{att}}$, and a message ("output", $\mathsf{Enc}(\mathsf{epk}_i, 0^{|\mathsf{outp}|}), \sigma_{\text{TEE}}$) from $\mathcal{G}_{\text{att}}$ to $\mathcal{P}_i$.
- Finally, Sim instructs $\mathcal{F}_{\text{Ekiden}}$ by sending a "ok" message.

Case 2: When a *corrupted* party $\mathcal{P}_i$ is given input ("request", cid, inp, eid) by $\mathcal{Z}$, Sim learns the input when Sim works as follows:

- If $\mathcal{P}_i$ sends ("read", cid) to $\mathcal{F}_{\text{blockchain}}$, Sim obtains the latest state (denoted s) from $\mathcal{F}_{\text{Ekiden}}$, and sends s to $\mathcal{P}_i$ on behalf of $\mathcal{F}_{\text{blockchain}}$.
- If $\mathcal{P}_i$ sends a "resume" message to $\mathcal{G}_{\text{att}}$ with input ("request", cid, $\mathsf{inp}_{\mathsf{ct}}$, s), Sim emulates $\mathcal{G}_{\text{att}}$ as follows: Sim queries $\mathcal{F}_{\text{Ekiden}}$ to check if s is not the latest state, Sim aborts. Sim computes $\mathsf{inp}' = \mathsf{Dec}(\mathsf{sk}^{\text{in}}_{\mathsf{cid}}, \mathsf{inp}_{\mathsf{ct}})$. Then Sim sends ("request", cid, inp', eid) to $\mathcal{F}_{\text{Ekiden}}$ on $\mathcal{P}_i$'s behalf.
- Upon receiving $\ell_{\mathsf{st}'_{\mathsf{ct}}}$ and $\ell(\mathsf{outp})$ from $\mathcal{F}_{\text{Ekiden}}$, Sim computes $c = \mathsf{Enc}(\mathsf{k}^{\text{out}}_{\mathsf{cid}}, 0^{|\mathsf{outp}|})$ and sends $((\text{"atom-deliver"}, \mathsf{H}(\mathsf{inp}_{\mathsf{ct}}), \mathsf{H}(s), \ell_{\mathsf{st}'_{\mathsf{ct}}}, \mathsf{H}(c)), \sigma_{\text{TEE}}, c)$ from $\mathcal{G}_{\text{att}}$ to $\mathcal{P}_i$. Sim records c.
- If $\mathcal{P}_i$ sends a "resume" message to $\mathcal{G}_{\text{att}}$ with input ("claim output", cid, $(\mathsf{st}'_{\mathsf{ct}}, \mathsf{outp}_{\mathsf{ct}}, \sigma, \mathsf{epk}_i)$), Sim emulates $\mathcal{G}_{\text{att}}$ as follows: Sim first checks that $\mathcal{G}_{\text{att}}$ has previously sent $\mathsf{outp}_{\mathsf{ct}}$ to $\mathcal{P}_i$ and that $(\mathsf{st}'_{\mathsf{ct}}, \sigma)$ has been stored by $\mathcal{F}_{\text{blockchain}}$. Sim aborts if any of the above checks fails. Sim obtains outp from $\mathcal{F}_{\text{Ekiden}}$ and sends ("output", $\mathsf{Enc}(\mathsf{epk}_i, \mathsf{outp}), \sigma$) to $\mathcal{P}_i$.

(3) Public read: On any call ("read", cid) from $\mathcal{P}_i$, Sim emulates a "read" message to $\mathcal{F}_{\text{blockchain}}$. If $\mathcal{P}_i$ is corrupted, Sim sends to $\mathcal{F}_{\text{Ekiden}}$ a "read" message on $\mathcal{P}_i$'s behalf and forward the response to $\mathcal{A}$.

(4) Corrupted enclaves:

Sim obtains eids of corrupted enclaves when $\mathcal{Z}$ corrupts them. In real world, $\mathcal{Z}$ could terminate a corrupted enclave at any point, or could strategically drop some messages while letting others go through. To faithfully emulate $\mathcal{Z}$'s "damage", Sim sends every messages leaving or entering a corrupted enclave to $\mathcal{Z}$ and only delivers the message if $\mathcal{Z}$ permits. Sim instructs $\mathcal{F}_{\text{Ekiden}}$ to abort if the emulated execution is terminated by $\mathcal{Z}$ prematurely. Specifically, upon receiving $(\mathsf{cid}, \ell(\mathsf{st}'), \ell(\mathsf{outp}), \mathsf{eid})$ from $\mathcal{F}_{\text{Ekiden}}$, Sim replies with "ok" only if the corresponding "output" message from $\mathcal{G}_{\text{att}}$ is allowed by $\mathcal{Z}$.

b) Validity of Sim*:* We show that no environment can distinguish an interaction with $\mathcal{A}$ and $\mathbf{Prot}_{\text{Ekiden}}$ from one with Sim and $\mathcal{F}_{\text{Ekiden}}$ by hybrid arguments. Consider a sequence of hybrids, starting with the real protocol execution. Hybrid

H_1 lets Sim to emulate $\mathcal{G}_{\text{att}}$ and $\mathcal{F}_{\text{blockchain}}$. H_2 filters out the forgery attacks against Σ_{TEE}. H_3 filters out the second pre-image attacks against the hash function. H_4 has Sim emulate the creation phase. H_5 replaces the encryption of input and output with encryption of 0, and replaces encryption of states with random strings with the same length. The indispensability between adjacent hybrids are shown below.

Hybrid H_1 proceeds as in the real world protocol, except that Sim emulates $\mathcal{G}_{\text{att}}$ and $\mathcal{F}_{\text{blockchain}}$. Specially Sim generates a key pair $(\mathsf{pk}_{\text{TEE}}, \mathsf{sk}_{\text{TEE}})$ for Σ_{TEE} and publishes pk_{TEE}. Whenever $\mathcal{A}$ wants to communicate with $\mathcal{G}_{\text{att}}$, Sim records $\mathcal{A}$'s messages and faithfully emulates $\mathcal{G}_{\text{att}}$'s behavior. Similarly, Sim emulates $\mathcal{F}_{\text{blockchain}}$ by storing items internally.

As $\mathcal{A}$'s view in H_1 is perfectly simulated as in the real world, $\mathcal{Z}$ cannot distinguish between H_1 and the real execution.

Hybrid H_2 proceeds as in H_1, except for the following modifications. If $\mathcal{A}$ invoked $\mathcal{G}_{\text{att}}$ with a correct message $(\text{“install”}, \widehat{\mathsf{Contract}})$, then for all sequential “resume” calls, Sim records a tuple $(\mathsf{outp}, \sigma_{\text{TEE}})$ where outp is the output of $\widehat{\mathsf{Contract}}$ and σ_{TEE} is an attestation under sk_{TEE}. Let Ω denote the set of all such tuples. Whenever $\mathcal{A}$ sends an attested output $(\mathsf{outp}, \sigma_{\text{TEE}}) \notin \Omega$ to $\mathcal{F}_{\text{blockchain}}$ or an honest party $\mathcal{P}_i$, Sim aborts.

The indistinguishability between H_1 and H_2 can be shown by the following reduction to the the EU-CMA property of Σ: In H_1, if $\mathcal{A}$ sends forged attestations to $\mathcal{F}_{\text{blockchain}}$ or $\mathcal{P}_i$, signature verification by $\mathcal{F}_{\text{blockchain}}$ or an honest party $\mathcal{P}_i$ will fail with all but negligible probability. If $\mathcal{Z}$ can distinguish H_2 from H_1, $\mathcal{Z}$ and $\mathcal{A}$ can be used to win the game of signature forgery.

Hybrid H_3 is the same as H_2 besides the following modifications. If $\mathcal{A}$ invoked $\mathcal{G}_{\text{att}}$ with a correct “request” message, Sim records execution result $\mathsf{outp}_{\text{ct}}$ before outputting it. Whenever $\mathcal{A}$ sends to $\mathcal{G}_{\text{att}}$ a “claim output” message with a input $\mathsf{outp}'_{\text{ct}}$ that is not previously generated by $\mathcal{G}_{\text{att}}$, Sim aborts.

The indistinguishability between H_3 and H_2 can be shown by a reduction to the second pre-image resistance property of the hash function. In H_2, $\mathcal{A}$ obtains $\mathcal{H} = \left\{\mathsf{H}(\mathsf{outp}^i_{\text{ct}})\right\}_i$ and $\mathcal{O} = \left\{\mathsf{outp}^i_{\text{ct}}\right\}_i$ from $\mathcal{G}_{\text{att}}$ through “request” calls. If $\mathcal{A}$ sends a “claim output” message with $\mathsf{outp}_{\text{ct}} \notin \mathcal{O}$, $\mathcal{G}_{\text{att}}$ aborts unless a $\mathsf{H}(\mathsf{outp}_{\text{ct}}) \in \mathcal{H}$. If $\mathcal{Z}$ can distinguish H_3 from H_2, it follows that $\mathcal{A}$ can break the second pre-image resistancy.

Hybrid H_4 is the same as H_3 but has Sim emulate the contract creation, i.e. honest parties will send “create” to $\mathcal{F}_{\text{Ekiden}}$. Sim emulates messages from $\mathcal{G}_{\text{att}}$ and $\mathcal{F}_{\text{blockchain}}$ as described above. If $\mathcal{P}_i$ is corrupted, Sim sends $(\text{“create”}, \mathsf{Contract})$ to $\mathcal{F}_{\text{Ekiden}}$ as $\mathcal{P}_i$.

It is clear that the $\mathcal{A}$'s view is distributed exactly as in H_3, as Sim can emulate $\mathcal{G}_{\text{att}}$ and $\mathcal{F}_{\text{blockchain}}$ perfectly.

Hybrid H_5 is the same as H_4 except that honest parties also sends “request” messages to $\mathcal{F}_{\text{Ekiden}}$. If $\mathcal{P}_i$ is corrupted, Sim emulates real-world messages with the help of $\mathcal{F}_{\text{Ekiden}}$, as described above.

In $\mathcal{A}$'s view, the difference between H_5 and H_4 are the following.

- Any message $(\text{“atom-deliver”}, h_{\mathsf{inp}}, h_{\mathsf{old}}, s, h_{\mathsf{outp}}, c)$ sent from $\mathcal{G}_{\text{att}}$ to $\mathcal{P}_i$ with $s = \mathcal{SE}.\mathsf{Enc}(\mathsf{k}^{\text{state}}_{\mathsf{cid}}, \mathsf{st}')$ and $c = \mathcal{SE}.\mathsf{Enc}(\mathsf{k}^{\text{out}}_{\mathsf{cid}}, \mathsf{outp}))$ in H_4 is replaced with $(\text{“atom-deliver”}, h_{\mathsf{inp}}, h_{\mathsf{old}}, \ell_{\mathsf{st}'_{\text{ct}}}, \mathsf{H}(c'), c')$ where $c' = \mathsf{Enc}(\mathsf{k}^{\text{out}}_{\mathsf{cid}}, 0^{|c|})$. Recall that $\ell_{\mathsf{st}'_{\text{ct}}}$ is a random string with length $|\mathsf{st}'_{\text{ct}}|$ chosen by $\mathcal{F}_{\text{Ekiden}}$ when generating state st_{ct}.
- If $\mathcal{P}_i$ is an honest party, any message $(\text{“request”}, \mathsf{cid}, \mathcal{AE}.\mathsf{Enc}(\mathsf{pk}^{\text{in}}_{\mathsf{cid}}, \mathsf{inp}), s)$ sent to $\mathcal{G}_{\text{att}}$ is replaced with $(\text{“request”}, \mathsf{cid}, c', s)$ where $c' = \mathsf{Enc}(\mathsf{pk}^{\text{in}}_{\mathsf{cid}}, 0)$, and any message $(\text{“output”}, \mathcal{AE}.\mathsf{Enc}(\mathsf{k}^{\text{out}}_{\mathsf{cid}}, \mathsf{outp}))$ sent from $\mathcal{G}_{\text{att}}$ to $\mathcal{P}_i$ is replaced with $(\text{“output”}, \mathsf{Enc}(\mathsf{epk}_i, 0))$.

Indistinguishability between H_5 and H_4 can be directly reduced to the IND-CPA property of $\mathcal{AE}$ and $\mathcal{SE}$. Having no knowledge of the secret key, $\mathcal{A}$ cannot distinguish encryption of $\vec{0}$ from encryption of other messages. Note that we don't require IND-CCA security because $\mathcal{A}$ do not have direct access to an decryption oracle.

It remains to observe that H_5 is identical to the ideal protocol. Throughout the simulation, we maintain the following invariant: $\mathcal{F}_{\textbf{Ekiden}}$ **always has the latest state**, regardless who created the contract and who has queried the contract. This invariant ensures that H_5 precisely reflects ideal execution of $\mathcal{F}_{\text{Ekiden}}$. □

D. Ekiden Performance Extensions

In this section we discuss several performance optimizations to the simple protocol. Together, these optimizations reduce the number of round trips and storage capacity required from the blockchain, and reduce work for compute nodes. Despite the performance improvements, all optimizations are transparent to the security interface: we use the same ideal functionality for both the simple and extended protocols. We present a formal protocol block defining the enhanced protocol $\mathbf{Prot}^{\text{full}}_{\text{Ekiden}}$ in Figure 8. For now, we provide a high-level description of the insight and challenges involved in each application.

a) Using a Merkleized state store: In the original protocol, the entire encrypted state st_{ct} is written to the blockchain after each query. The entire state needs to be re-encrypted because the modification side-effect should not leak information to the adversary. However, this approach is inefficient when each st is very large yet each query modifies only a small part. In our Token application, for example, we model a token with 500,000 different user accounts, even though each transaction only debits one account and credits one other.

We observe that a Merkleized data structure, where the data is broken down into pieces and a “root hash” cryptographically summarizes the collection of pieces, would allow us to efficiently store and update pieces of the state and be able to verify its integrity with a single root hash tracked by the validator committee. To read a piece of the current state, the enclave

must retrieve the state root hash from the validator committee and Merkle tree nodes from the storage committee to verify a read's authenticity. In the token transaction, each transaction touches a constant number of records, hence requiring $O(Tlog(M))$ storage complexity for T transactions if there are M users, compared to $O(MT)$ in the simple protocol.

The set of Merkle tree nodes used may leak information about which query was invoked. We note that the ideal functionality $\mathcal{F}_{\text{Ekiden}}$ is parameterized by a leakage function ℓ, such that the notation is in place to model the effect leakage resulting from Merkle tree queries.

b) Caching intermediate states at the enclave: In the simple protocol, each round begins with reading the state ciphertext from the blockchain, and ends with writing the next state ciphertext from the blockchain. In the case that In our extended protocol, we optimistically use the previous state in the Cache, if available. This results in a performance improvement when the same enclave eid is used for multiple sequential queries. This is especially beneficial when there are transactions accessing the same nodes in the Merkle tree.

Bootstrapping from genesis seems to be necessary whenever a query is sent to a new enclave (e.g., because the previously-used enclave host has crashed). In practice, we also define a policy for checkpoints by storing the entire state (not just the diff) after every fixed number of intervals. We leave the formal presentation of this generalization to future work.

c) Batching transactions off-chain: Just as the caching optimization above removes the need to read from the blockchain in each query, we can also coalesce the writes for multiple sequential queries into a single message to the blockchain. This reduces both the number of network round trips, as well as the total communication cost. When multiple queries in a batch write to the same location, only the last write needs to be stored on the blockchain.

In our protocol we do not define a policy for how many transactions must go in a batch. Instead, we formally expose this choice to the adversary. The choice of batching strategy has no impact on the security guarantees of our formalism. Each `query` invocation simply stores the inputs in a buffer, and the adversary can invoke the `commitBatch` method at any time to commit the entire buffer.

Batching is not a panacea. In order to maintain security, the *decrypted* outputs must not leave the enclave unless the updated state $\Delta\mathsf{st}_{\text{ct}}$ is committed in the blockchain. Hence a user cannot receive output from a query until the entire batch is committed, and so only input-independent queries can appear in the same batch.

d) Transaction pipelining with optimistic consensus: We add a step before the compute nodes send the results of a batch of transactions to the client and the validator committee. The compute nodes have committed to their results by signing them. Then, they first send their commitments to a compute node designated as the *leader*. If the compute node leader collects commitments with matching results from all compute nodes (including itself), then the system can continue without waiting for the results to be committed in the blockchain. If there is a discrepancy among the collected commitments, or if the compute node leader fails, then the system ignores the optimistic consensus step for this batch of transactions and waits for the results to be committed in the blockchain.

In the optimistic case, the compute committee is assured that the network will reach consensus on the same output, and the nodes start working on the next batch of transactions right away, thus saving multiple network round trips the compute committee would have been idle waiting for the validator nodes to finish their BFT protocol.

e) Coordinating the choice of compute nodes: The Ekiden protocol leaves it up to the client to decide which compute node and enclave to query. All of the security guarantees of $\mathcal{F}_{\text{Ekiden}}$ hold regardless of this choice. As a pragmatic solution, we propose to have clients defer to centralized *coordinators* that perform load balancing and random assignment of compute nodes to tasks, based on reputations and prior experience. If a task is not completed after some timeout, the coordinator can signal the client to repeat the query at another enclave. Randomization can ensure that a host cannot adaptively choose a particular target task to degrade service. In this way Ekiden would prevent an adversary from degrading service for targeted applications. Following other work, incentives can be aligned by having compute miners make security deposits before they are assigned to a task.

1) Extended Protocol: The enhanced Ekiden protocol is specified in Figure 8.

$\mathbf{Prot}^{\text{full}}_{\text{Ekiden}}(\{\mathcal{P}_i\}_{i\in[N]})$

Clients $\mathcal{P}_i$:

Initialize: $(\mathsf{ssk}_i, \mathsf{spk}_i) \leftarrow\!\$\ \Sigma.\mathsf{KGen}(1^\lambda)$, $(\mathsf{esk}_i, \mathsf{epk}_i) \leftarrow\!\$\ \mathcal{AE}.\mathsf{KGen}(1^\lambda)$

On input ("create", Contract) from environment $\mathcal{Z}$:
- $\mathsf{cid} := \mathsf{create}(\mathsf{Contract})$
- assert cid has been stored on $\mathcal{F}_{\text{blockchain}}$
- output ("receipt", cid)

On input ("request", cid, inp, eid) from environment $\mathcal{Z}$:
- obtains $\mathsf{pk}^{\text{in}}_{\mathsf{cid}}$ from $\mathcal{F}_{\text{blockchain}}$
- let $\mathsf{inp}_{\text{ct}} := \mathcal{AE}.\mathsf{Enc}(\mathsf{pk}^{\text{in}}_{\mathsf{cid}}, \mathsf{inp})$
- $\sigma_{\mathcal{P}_i} := \mathsf{Sig}(\mathsf{ssk}_i, (\mathsf{cid}, \mathsf{inp}_{\text{ct}}))$
- $(\Delta\mathsf{st}_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma) := \mathsf{query}(\mathsf{cid}, \mathsf{inp}_{\text{ct}}, \sigma_{\mathcal{P}_i})$
- parse σ as $(\sigma_{\text{TEE}}, h_{\mathsf{inp}}, h_{\mathsf{old}}, h_{\mathsf{outp}}, \mathsf{spk}_i)$
- assert σ verifies
- assert $\exists n\ s.t.\ h^n_{\mathsf{inp}} = \mathsf{H}(\mathsf{inp}_{\text{ct}})$
- $o := \text{claim-output}(\mathsf{cid}, \Delta\mathsf{st}_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma, \mathsf{epk}_i)$
- // if the previous state has been used by a parallel query
- **if** $o = \perp$ **then** : jump to the beginning of this call
- parse o as $(\mathsf{outp}'_{\text{ct}}, \sigma_{\text{TEE}})$
- assert $\Sigma_{\text{TEE}}.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma_{\text{TEE}}, \mathsf{outp}'_{\text{ct}})$ // $\mathsf{pk}_{\text{TEE}} := \mathcal{G}_{\text{att}}.\text{getpk()}$
- output $\mathcal{AE}.\mathsf{Dec}(\mathsf{esk}_i, \mathsf{outp}'_{\text{ct}})$

On receive ("commit batch", cid, eid) from $\mathcal{A}$:
- // optimistically commit a batch without providing state
- send (eid, "resume", ("commit batch", cid, $\perp$)) to $\mathcal{G}_{\text{att}}$
- **if** receive ("cache miss") from $\mathcal{G}_{\text{att}}$ **then**
  - send ("read", cid) to $\mathcal{F}_{\text{blockchain}}$
  - receive val from $\mathcal{F}_{\text{blockchain}}$
  - send (eid, "resume", ("commit batch", cid, val)) to $\mathcal{G}_{\text{att}}$

On receive ("read", cid) from environment $\mathcal{Z}$:
- send ("read", cid) to $\mathcal{F}_{\text{blockchain}}$
- receive val from $\mathcal{F}_{\text{blockchain}}$ and **return** val

Compute Node Subroutines (called by $\mathcal{P}_i$):

On input create(Contract):
- send ("install", $\widehat{\mathsf{Contract}}$) to $\mathcal{G}_{\text{att}}$, wait for eid
- send (eid, "resume", ("create")) to $\mathcal{G}_{\text{att}}$
- wait for $((\mathsf{Contract}, \mathsf{cid}, \mathsf{st}_0, \mathsf{pk}^{\text{in}}_{\mathsf{cid}}), \sigma_{\text{TEE}})$ from $\mathcal{G}_{\text{att}}$
- send ("write", cid, (Contract, cid, $\mathsf{st}_0, \mathsf{pk}^{\text{in}}_{\mathsf{cid}}$)) to $\mathcal{F}_{\text{blockchain}}$
- receive ("receipt", cid) from $\mathcal{F}_{\text{blockchain}}$ and **return**

On input query(cid, $\mathsf{inp}_{\text{ct}}, \sigma_{\mathcal{P}_i}$):
- send ("read", cid) to $\mathcal{F}_{\text{blockchain}}$ and wait for st_{ct}
- send (eid, "resume", ("request", cid, $\mathsf{inp}_{\text{ct}}, \sigma_{\mathcal{P}_i}, \mathsf{st}_{\text{ct}}$)) to $\mathcal{G}_{\text{att}}$
- receive $((h_{\mathsf{inp}}, h_{\mathsf{old}}, \Delta\mathsf{st}_{\text{ct}}, h_{\mathsf{outp}}, \mathsf{spk}_i), \sigma_{\text{TEE}}, \mathsf{outp}_{\text{ct}})$ from $\mathcal{G}_{\text{att}}$
- let $\sigma := (\sigma_{\text{TEE}}, h_{\mathsf{inp}}, h_{\mathsf{old}}, h_{\mathsf{outp}}, \mathsf{spk}_i)$
- **return** $(\Delta\mathsf{st}_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma)$

On input claim-output(cid, $\Delta\mathsf{st}_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma, \mathsf{epk}_i$):
- send ("write", cid, $(\Delta\mathsf{st}_{\text{ct}}, \sigma)$) to $\mathcal{F}_{\text{blockchain}}$
- **if** receive ("reject", cid) from $\mathcal{F}_{\text{blockchain}}$: **return** $\perp$
- send (eid, "resume", ("claim output", $\Delta\mathsf{st}_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma, \mathsf{epk}_i$)) to $\mathcal{G}_{\text{att}}$
- receive ("output", $\mathsf{outp}_{\text{ct}}, \sigma_{\text{TEE}}$) from $\mathcal{G}_{\text{att}}$or abort
- **return** $(\mathsf{outp}_{\text{ct}}, \sigma_{\text{TEE}})$

Enclave program $\widehat{\text{Contract}}$

Local state: Cache $:= \emptyset$, Batch $:= \emptyset$

On input ("create")
- $\mathsf{cid} := \mathsf{H}(\mathsf{Contract})$
- $(\mathsf{pk}^{\text{in}}_{\mathsf{cid}}, \mathsf{sk}^{\text{in}}_{\mathsf{cid}}) := \text{keyManager}(\text{"input key"})$
- $\mathsf{k}^{\text{state}}_{\mathsf{cid}} := \text{keyManager}(\text{"state key"})$
- $\mathsf{st}_0 := \mathcal{SE}.\mathsf{Enc}(\mathsf{k}^{\text{state}}_{\mathsf{cid}}, \vec{0})$
- $\mathsf{Cache}[\mathsf{cid}] = \mathsf{st}_0$ // cache state locally
- **return** $(\mathsf{Contract}, \mathsf{cid}, \mathsf{st}_0, \mathsf{pk}^{\text{in}}_{\mathsf{cid}})$

On input ("request", cid, $\mathsf{inp}_{\text{ct}}, \sigma_{\mathcal{P}_i}, \mathsf{st}_{\text{ct}}$) from $\mathcal{P}$:
- assert $\Sigma.\mathsf{Vf}(\mathsf{spk}_i, \sigma_{\mathcal{P}_i}, (\mathsf{cid}, \mathsf{inp}_{\text{ct}}))$
- add $(\mathsf{inp}_{\text{ct}}, \mathsf{spk}_i)$ to $\mathsf{Batch}[\mathsf{cid}]$

On input ("commit batch", cid, inp):
- make a local copy of Batch and parse it as $\{(\mathsf{inp}_{\text{ct}}{}^i, \mathsf{spk}_i)\}_{i\in[N]}$
- reset the global batch: $\mathsf{Batch} = \emptyset$
- // retrieve $\mathsf{pk}^{\text{in}}_{\mathsf{cid}}, \mathsf{sk}^{\text{in}}_{\mathsf{cid}}, \mathsf{k}^{\text{state}}_{\mathsf{cid}}$ from keyManager as above
- $\mathsf{inp}_i := \mathcal{AE}.\mathsf{Dec}(\mathsf{sk}^{\text{in}}_{\mathsf{cid}}, \mathsf{inp}_{\text{ct}}{}^i)$ for $i \in [N]$
- **if** $\mathsf{Cache}[\mathsf{cid}] = \perp \wedge \mathsf{inp} = \perp$ **then** :
  - return ("cache miss")
- **if** $\mathsf{Cache}[\mathsf{cid}] = \perp$ **then** :
  - send ("$\in$", cid, inp) to $\mathcal{F}_{\text{blockchain}}$; wait for true or abort
  - parse inp as $\mathsf{st}^0_{\text{ct}} \parallel \{\Delta\mathsf{st}^n_{\text{ct}}\}_n$
  - reconstruct latest state and store it at $\mathsf{Cache}[\mathsf{cid}]$
- $\mathsf{k}^{\text{out}}_{\mathsf{cid}} := \text{keyManager}(\text{"output key"})$
- let $\mathsf{st}[0] = \mathsf{Cache}[\mathsf{cid}]$
- **for** $i = 1 \ldots N$:
  - $\mathsf{st}[i], \mathsf{outp}[i] = \mathsf{Contract}(\mathsf{st}[i-1], \mathsf{inp}_i, \mathsf{pk}_i)$
  - $\mathsf{outp}_{\text{ct}}[i] = \mathcal{SE}.\mathsf{Enc}(\mathsf{k}^{\text{out}}_{\mathsf{cid}}, \mathsf{outp}[i])$
- $\mathsf{Cache}[\mathsf{cid}] = \mathsf{st}[N]$ // cache the latest state
- $\Delta\mathsf{st} := \mathsf{diff}(\mathsf{st}[N], \mathsf{st}[0])$
- $h_{\mathsf{inp}} := \!= \mathsf{H}(\mathsf{inp}_{\text{ct}}[1]) \parallel \cdots \parallel \mathsf{H}(\mathsf{inp}_{\text{ct}}[N])$
- $h_{\mathsf{old}} := \mathsf{H}(\mathsf{st}[0])$
- $h_{\mathsf{outp}} := \mathsf{H}(\mathsf{outp}_{\text{ct}}[1]) \parallel \cdots \parallel \mathsf{H}(\mathsf{outp}_{\text{ct}}[N])$
- $\Delta\mathsf{st}_{\text{ct}} := \mathcal{SE}.\mathsf{Enc}(\mathsf{k}^{\text{state}}_{\mathsf{cid}}, \Delta\mathsf{st})$
- $\mathsf{outp}_{\text{ct}} := \mathsf{outp}_{\text{ct}}[1] \parallel \cdots \parallel \mathsf{outp}_{\text{ct}}[N]$
- send $((h_{\mathsf{inp}}, h_{\mathsf{old}}, \Delta\mathsf{st}_{\text{ct}}, h_{\mathsf{outp}}, \mathsf{spk}_i), \mathsf{outp}_{\text{ct}})$ to all $\{\mathcal{P}_i\}_{i\in[N]}$

On input ("claim output", $\Delta\mathsf{st}_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma, \mathsf{epk}_i$):
- parse σ as $(\sigma_{\text{TEE}}, h_{\mathsf{inp}}, h_{\mathsf{old}}, h_{\mathsf{outp}}, \mathsf{spk}_i)$
- parse h_{outp} as $h^1_{\mathsf{outp}} \parallel \cdots \parallel h^n_{\mathsf{outp}}$
- assert $\exists n\ s.t.\ h^n_{\mathsf{outp}} = \mathsf{H}(\mathsf{outp}_{\text{ct}})$
- send ("$\in$", cid, $(\Delta\mathsf{st}_{\text{ct}}, \sigma)$) to $\mathcal{F}_{\text{blockchain}}$
- receive true from $\mathcal{F}_{\text{blockchain}}$
- $\mathsf{k}^{\text{out}}_{\mathsf{cid}} := \text{keyManager}(\text{"output key"})$
- $\mathsf{outp} := \mathcal{SE}.\mathsf{Dec}(\mathsf{k}^{\text{out}}_{\mathsf{cid}}, \mathsf{outp}_{\text{ct}})$
- **return** ("output", $\mathcal{AE}.\mathsf{Enc}(\mathsf{epk}_i, \mathsf{outp})$) // reveal the output

Fig. 8. Enhanced Ekiden Protocol. $\mathsf{diff}(\cdot,\cdot)$ is a function that takes in two states and output the difference.