Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution

Raymond Cheng[1,2], Fan Zhang[1,3], Jernej Kos[1], Warren He[1,2], Nicholas Hynes[1,2], Noah Johnson[1,2]
Ari Juels[4], Andrew Miller[1,5], Dawn Song[1,2]

[1]Oasis Labs, [2]University of California, Berkeley, [3]Cornell University, [4]Cornell Tech, [5]University of Illinois, Urbana-Champaign, IC3

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). Capable of operating on any desired blockchain, Ekiden permits concurrent, off-chain execution of smart contracts within TEE-backed compute nodes, yielding high performance, low cost, and confidentiality for sensitive data.

Ekiden enforces a strong set of security and availability properties. By maintaining on-chain state, it achieves consistency, meaning a single authoritative sequence of state transitions, and availability, meaning contracts can survive the failure of compute nodes. Ekiden is anchored in a formal security model expressed as an ideal functionality. We prove the security of the corresponding implemented protocol in the UC framework.

Our implementation of Ekiden supports contract development in Rust and the Ethereum Virtual Machine (EVM). We present experiments for applications including machine learning models, poker, and cryptocurrency tokens. Ekiden is designed to support multiple underlying blockchains. We have built one end-to-end instantiation of our system, Ekiden-BT, with a blockchain extending from Tendermint. Ekiden-BT achieves example performance of 600x more throughput and 400x less latency at 1000x less cost than on the Ethereum mainnet. When used with Ethereum as the backing blockchain, Ekiden still costs less than on-chain execution and supports contract confidentiality.

1 INTRODUCTION

Smart contracts are protocols that digitally enforce agreements between or among 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 [53] and secure multiparty computation [101]. However, these approaches have significant performance overhead and are only applicable for very limited use cases with relatively simple computations. A more performant and general-purpose option is use of a *trusted execution environment* (TEE).

An example TEE provides a fully isolated environment called an *enclave* that prevents other applications, the operating system, and the host owner from tampering with or even learning the state of an application running in the enclave. For example, Intel Software Guard eXtensions (SGX) provides an implementation of a secure enclave. The Keystone-enclave project aims to provide an open-source secure-enclave design [7].

A TEE thereby provides strong confidentiality for smart contract data that blockchains cannot. Unfortunately, a TEE alone cannot guarantee availability or provide secure networking or persistent storage. Thus, it cannot alone achieve blockchains' authoritative transaction ordering, persistent record keeping, or resilience to network attacks.

In this paper, we show that blockchains and TEEs have complementary properties that can be combined to improve both security and performance of smart contracts and enable novel and diverse applications.

Ekiden. We present Ekiden, a system for highly performant and confidentiality-preserving smart contract execution. To the best of our knowledge, Ekiden is the first confidentiality-preserving smart contract system that can perform thousands of transactions per second. The key to this achievement is the effective 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 that is expressed as a cryptographic ideal functionality [23], its principled design supports rigorous analysis of its security properties.

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 *consensus nodes*, which need not use trusted hardware. Ekiden is agnostic to consensus-layer mechanics, only requiring a blockchain

https://www.oasislabs.com

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. Enclave-based computation in Ekiden provides confidentiality, enabling efficient use of powerful cryptographic primitives that a TEE is known to emulate, such as functional encryption [40] and black-box obfuscation [73], and also provides a trustworthy source of randomness, a major acknowledged difficulty in blockchain systems [22].

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 4.1). 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 formally specify two protocol variations, a simple baseline and an optimization, to realize this functionality. 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.
- *Strong security properties:* Ekiden's security model and implementation realize strong notions of consistency (authoritative sequencing of concurrent transaction requests) and atomicity (all-or-nothing state checkpointing and delivery of messages to clients). Ensuring these properties in an asynchronous network is a challenge that we overcome by using the blockchain to checkpoint state and conditioning enclave communication on valid blockchain updates in an *atomic delivery* protocol. Ekiden thus requires a TEE to have a fresh, correct view of the blockchain, but some TEE implementations such as SGX lack a trusted time source. We address this additional challenge with a novel *proof of publication* protocol that requires only partially trustworthy relative timer (e.g., available in SGX). Ekiden also leverages enclave isolation to achieve contract confidentiality in a model of black-box execution.
- *High availability:* TEE hosts may crash or lose network connectivity, posing the risk and challenge of lost and/or conflicting state. Ekiden treats TEEs as expendable and interchangeable: Should one enclave be lost, failover to any other live enclave is possible. Ensuring such availability involves a strategy of enclave key management and blockchain key and state checkpointing, supported by Ekiden's atomicity to ensure consistency during failovers.
- *Performance:* Ekiden includes several performance optimizations that minimize use of the blockchain, which is a bottleneck. Our optimizations do not degrade security: We show that they realize the same $\mathcal{F}_{\text{Ekiden}}$ functionality as the unoptimized protocol. We evaluate their individual and cumulative impact, showing speed, throughput, and on-chain storage 2–4 orders of magnitude better than baseline on-chain Ethereum execution.

Evaluation. We evaluate the performance of Ekiden on a suite of applications that exercise the full range of system resources and demonstrate how Ekiden enables application deployment that would otherwise be impractical due to privacy and/or performance concerns. They include a machine learning framework, within which we implement medical-diagnosis and credit-scoring applications, a smart building thermal model, and a poker game. We also port an Ethereum Virtual Machine implementation to Ekiden, so that existing contracts (e.g., written in Solidity), such as Cryptokitties [1] and the ERC20 token, can run in our framework as well. We report on development effort, showing that the programming model in Ekiden lends itself to simple and intuitive application development. Contracts in Ekiden process transactions 2–3 orders of magnitude both faster and higher throughput over Ethereum. Our performance optimizations also greatly compress the amount of data stored on the blockchain, yielding a 2–4 order of magnitude improvement over the baseline. (The advantage is greater for read-write operations on contracts with large state, such as our token contract.) Furthermore, Ekiden decouples computation from the blockchain and shards contracts, which allows the system to scale horizontally. In contrast, all transactions for all contracts must be serialized on a single blockchain in Ethereum.

2 BACKGROUND

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. Each node can thus individually 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 [37] and NEO [8] have 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 [37], a cost roughly 8 orders of magnitude higher than in AWS EC2 [80]. Furthermore, current systems offer no privacy guarantees. Users are identified by pseudonyms. As numerous studies have shown [68, 72, 78, 79], 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.

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

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 [11, 45, 67], a specific TEE technology, but we emphasize that it may use any comparable TEE with attestation capabilities, such as the ongoing effort Keystone-enclave [7] aiming to realize open-source secure hardware enclave. 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 DRAM, it is transparently encrypted with keys only available to the processor. 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 [43]. 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 [11]. 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.

Attested execution realized by trusted hardware is imperfect, however. For example, SGX alone cannot guarantee availability. A malicious host can terminate enclaves or drop messages arbitrarily. Even an honest host could accidentally lose enclave state in the event of a power cycle. The weak availability of SGX poses a fundamental challenge to the design of Ekiden. Furthermore, recent attacks on Intel SGX have shown that current implementations of TEEs often leak information through side channels [75, 98]. Ekiden is compatible with existing TEE defenses [19, 63, 73, 77, 94]. In future work we plan to extend Ekiden to maintain its security guarantees under a stronger threat model, where individual enclaves can be compromised without affecting service integrity. Ekiden can be extended with other secure computation techniques, such as secure multi-party computation [12, 27, 62].

3 OVERVIEW OF EKIDEN

In this section, we provide an overview of the design and security properties of Ekiden. We begin with a motivating example, highlighting the challenge met by existing systems. Then we present the high-level architecture, the workflow, and the security goals of Ekiden. Finally, we state the threat model and assumptions.

3.1 Motivation

As an example to motivate our work, consider a credit scoring application—an example we implement and report on in section 6.1. Credit scores are widely used by lenders, insurers, and others to evaluate the creditworthiness of consumers [9]. Despite its considerable revenue ($10.8B in 2017 [47]), the credit reporting industry in the U.S. is concentrated among a handful of credit bureaus [47].

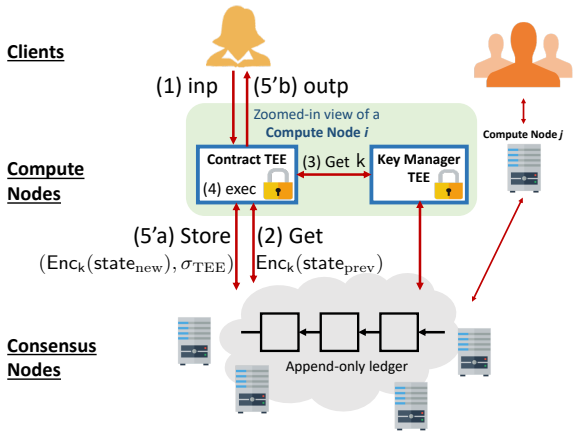

Figure 1: Overview of Ekiden architecture and workflow. Clients send inputs to confidentiality-preserving smart contracts, which are executed within a TEE at any compute node. The blockchain stores encrypted contract state. See Section 3.2 for details.

Such centralization creates large single points of failure and other problems, as highlighted by a recent data breach affecting nearly half the US population [18].

Blockchain-based decentralized credit scoring is thus an attractive and popular alternative. Bloom [60], 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.

3.2 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 $(\mathsf{outp}, \mathsf{st}_{\mathsf{new}}) := \mathsf{Contract}(\mathsf{st}_{\mathsf{prev}}, \mathsf{inp})$, ingesting as input a previous state $\mathsf{st}_{\mathsf{prev}}$ and a client's input inp, and generating an output outp and new state $\mathsf{st}_{\mathsf{new}}$.

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 Ekiden and a workflow of Ekiden smart contracts. As it shows, there are three types of entities in Ekiden: Clients, compute nodes and consensus 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** instantiate multiple TEEs to run contract programs. They also instantiate a service called a *key manager* in a TEE. 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. Compute nodes also perform key management for contracts in the key manager. Upon requested by the contract TEE, a key manager TEE creates or retrieves existing keys, as needed. We defer details of key management to Section 4.5. The key manager TEEs synchronize their state via the blockchain.
- **Consensus 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. Consensus nodes are responsible for checking the validity of state updates using TEE attestations, as we discuss below.

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 5.1.

For simplicity, we assume a client has a priority list of compute nodes to use. In Appendix E, we describe a coordinator that facilitates compute node discovery and load balancing. We denote a client as $\mathcal{P}$ and a compute node as Comp.

Contract creation. When creating a contract, $\mathcal{P}$ sends a piece of contract code Contract to Comp. Comp loads Contract into a TEE (called contract TEE hereafter), and starts the initialization. The contract TEE creates a fresh contract id cid, obtains fresh $(\mathsf{pk}^{\mathsf{in}}_{\mathsf{cid}}, \mathsf{sk}^{\mathsf{in}}_{\mathsf{cid}})$ pair and $\mathsf{k}^{\mathsf{state}}_{\mathsf{cid}}$ from the key manager TEE and generates an encrypted initial state $\mathsf{Enc}(\mathsf{k}^{\mathsf{state}}_{\mathsf{cid}}, \vec{0})$ and an attestation σ_{TEE}, proving the correctness of initialization and that $\mathsf{pk}^{\mathsf{in}}_{\mathsf{cid}}$ 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 $(\mathsf{Contract}, \mathsf{pk}^{\mathsf{in}}_{\mathsf{cid}}, \mathsf{Enc}(\mathsf{k}^{\mathsf{state}}_{\mathsf{cid}}, \vec{0}), \pi)$ to consensus nodes. The full protocol for contract creation is specified in the "create" call of $\mathbf{Prot}_{\mathsf{Ekiden}}$ (fig. 2). Consensus nodes verify π before accepting Contract, the encrypted initial state, and $\mathsf{pk}^{\mathsf{in}}_{\mathsf{cid}}$ as valid and placing it on the blockchain.

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 $\mathsf{pk}^{\mathsf{in}}_{\mathsf{cid}}$ associated with the contract cid from the blockchain, computes $\mathsf{inp}_{\mathsf{ct}} = \mathsf{Enc}(\mathsf{pk}^{\mathsf{in}}_{\mathsf{cid}}, \mathsf{inp})$ and sends to Comp a message $(\mathsf{cid}, \mathsf{inp}_{\mathsf{ct}})$, as specified in Lines 8-11 of $\mathbf{Prot}_{\mathsf{Ekiden}}$.

(2) Each contract is also associated with a secret state key $\mathsf{k}^{\mathsf{state}}_{\mathsf{cid}}$ known only to the contract and key manager. When executing a contract, Comp retrieves the contract code and $\mathsf{st}_{\mathsf{ct}} := \mathsf{Enc}(\mathsf{k}^{\mathsf{state}}_{\mathsf{cid}}, \mathsf{st}_{\mathsf{prev}})$, the encrypted previous state of contract cid, from the blockchain, and loads $\mathsf{st}_{\mathsf{ct}}$ and $\mathsf{inp}_{\mathsf{ct}}$ into a TEE and starts the execution, as specified in Line 30-33 of $\mathbf{Prot}_{\mathsf{Ekiden}}$.

(3-4) From the key manager TEE, the contract TEE obtains $\mathsf{k}^{\mathsf{state}}_{\mathsf{cid}}$ and $\mathsf{sk}^{\mathsf{in}}_{\mathsf{cid}}$, with which it decrypts $\mathsf{st}_{\mathsf{ct}}$ and $\mathsf{inp}_{\mathsf{ct}}$ and executes, generating an output outp, a new encrypted state $\mathsf{st}'_{\mathsf{ct}} := \mathsf{Enc}(\mathsf{k}^{\mathsf{state}}_{\mathsf{cid}}, \mathsf{st}_{\mathsf{new}})$, and an signature π proving correct computation, as specified in Line 7-13 of the TEE Wrapper (fig. 9). Key management is discussed in section 4.5.

(5a, 5b) Finally, Comp and $\mathcal{P}$ conduct an atomic delivery protocol which delivers outp to $\mathcal{P}$ and $(\mathsf{st}'_{\mathsf{ct}}, \pi)$ to the consensus nodes. We defer the detail of atomic delivery to Section 4.3. Briefly, Step 5a and Step 5b in fig. 1 are executed atomically, i.e. outp is revealed to $\mathcal{P}$ if and only if $(\mathsf{st}'_{\mathsf{ct}}, \pi)$ is accepted by consensus nodes. Consensus nodes verify π before accepting the new state as valid and placing it on the blockchain.

Concurrency. Ekiden compute nodes receive inputs and generate state updates concurrently. Thus, race conditions are possible, but handled by the consensus layer. If two compute nodes concurrently update the same state, only one will be accepted by the consensus layer. The rejected compute node will notify the client to retry.

Decoupling consensus from computation. In contrast to Ethereum, where contract execution is replicated by all nodes in the blockchain to reach consensus, Ekiden decouples consensus from contract execution. For every client request, the contract only needs to be executed by K compute nodes for some small K, a security parameter (e.g. in Figure 1, we set $K = 1$, which may be a reasonable choice in practice).

Agnostic to the specifics of contract execution, consensus nodes only need to verify π generated by TEEs. In our implementation, Comp obtains π from the Intel Attestation Service (IAS) [48]. As an SGX attestation is a group signature, its verification is facilitated by the IAS acting as group manager. To verify the correctness of an attestation σ_{TEE}, Comp first sends σ_{TEE} to IAS, which replies with a "certified" attestation $\pi := (b, \sigma_{\mathsf{TEE}}, \sigma_{\mathsf{IAS}})$, where $b \in \{0, 1\}$ indicates the validity of σ_{TEE} and σ_{IAS} is a signature over b and σ_{TEE} by IAS. As π is just a signature, consensus nodes need neither trusted hardware nor to contact the IAS to verify it.

Preventing replay attacks. When the output of a contract call has value, the system must prevent an adversary in control of a TEE host from conducting replay attacks, where a malicious compute node allows a malicious client to repeatedly execute queries on a prior state snapshot. For example, an attacker could try to repeatedly query a credit scoring contract that implements differential privacy, in order to exhaust the privacy budget and leak information about user data. Ekiden's atomic delivery protocol ensures that clients only see the output of a contract call after the system can prove that the state has been successfully written to the blockchain (Section 4.3). Atomic delivery enables the contract to impose query limits and transaction fees on queries to the smart contract.

3.3 Ekiden Security Goals

We formally characterize the security goals of Ekiden in Section 5. Some security properties, however, are implied by but not explicit in

the formal model. Here we call them out explicitly in order to clarify protocol design considerations in Ekiden. Briefly, Ekiden aims to support execution of general-purpose contracts while enforcing the following security properties:

Correct execution: Ekiden ensures that contract state transitions reflect correct execution of contract code on previous state with current inputs.

Secret state: Ekiden guarantees that contract state and inputs from honest clients are kept secret from all other parties. Contracts explicitly specify their *secret state*, which will automatically be encrypted with keys known only to enclave applications. Part of the contract state can also be marked *public*, in which case it is stored unencrypted on the blockchain with only integrity guarantees. We emphasize that Ekiden does not protect confidentiality at contract interfaces, thus application developers are responsible for ensuring that no secret is revealed through public output, and that the contract is free of side channels. We discuss defense against side-channels and application-level leakage in Section 4.4.

Fault-tolerance: Ekiden is resilient to network adversary model and compute node failures. In general, the system can make progress if at least one of the compute nodes is available. We discuss our threat model in Section 3.4 and tolerance to compute node failure and corruption in Section 4.1.

Consistency: Ekiden guarantees that at any time, the blockchain stores a single sequence of state transitions consistent with the view of each compute node. We rely on the consensus layer, i.e., blockchain, to ensure correct state transitions. Recall that contract programs take the form $(\text{outp}, \text{st}_{\text{new}}) := \text{Contract}(\text{st}_{\text{prev}}, \text{inp})$. Each attestation generated by compute nodes attests to the correct state transition from st_{prev} to st_{new}. While verifying attestations, consensus nodes also check that st_{prev} is the latest state stored on the blockchain, rejecting the attestation otherwise.

Atomic and isolated transactions: Ekiden ensures that concurrent client requests to a compute node are processed sequentially and in isolation. Furthermore, transactions are atomic, providing all-or-nothing delivery of messages to clients and on-chain checkpointing of state transitions, even in the face of a malicious network or host.

3.4 Threat Model and Assumptions

We present a formal adversarial model in Section 5. Informally, however, we assume the following:

Hardware Assumptions: We assume that TEE hardware is correctly implemented and securely manufactured. Recent work shows that the confidentiality of SGX enclaves may be compromised via side-channels [59, 97]. In light of this threat, we discuss various mechanisms to tolerate compromised enclaves in Section 4.1.

Blockchain Assumptions: Ekiden is designed to be agnostic to the underlying consensus protocol used by the blockchain. It can be deployed atop any blockchain implementation as long as the requirements specified below are met. Informally, we model a blockchain as a distributed append-only ledger that is trusted for integrity and availability, but not for privacy.

We assume the blockchain will perform prescribed computation correctly and is always available. In particular, Ekiden relies on consensus 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*. This is similar to the bulletin board model of blockchain used in [27].

In a permissioned blockchain, such proof can simply be a multisig signed by a majority of the consensus nodes. In permissionless blockchains, especially proof-of-work based ones, however, only a weaker notion of security can be achieved, as acknowledged in [27]. We discuss our strategy that confines the impact of potential blockchain synchronization failure in Section 4.1.

Shared-key bootstrapping: The Ekiden protocol involves establishing a shared master secret among all TEEs running the Ekiden program. The master secret is initially generated at a single TEE, but propagates to the rest in a peer-to-peer network. We thus assume that each new node can communicate at least once with an existing bootstrapped node to obtain the master secret. We discuss a key management protocol in detail in Section 4.5. We leave the exploration of other key management schemes, e.g., multiple master keys and secret-sharing schemes, for future work.

Threat Model: All parties in the system must trust Ekiden and TEE. We assume a strong adversary that can corrupt up to all but one compute nodes and any number of clients. By corrupting a compute node, the adversary gains full control of the operating system and the network stack, and thus can reorder messages and schedule processes arbitrarily. We assume the attacker cannot corrupt TEEs. A corrupted party reveals her entire internal state to the adversary and may deviate arbitrarily from the protocol.

Clients need not execute contracts themselves and do not require trusted hardware. We assume 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.

The adversary observes global network traffic and may reorder and delay messages arbitrarily. If a compute node times out when processing a client request, the client needs to resubmit the request to another (possibly randomly chosen) compute node. The adversary could also censor messages selectively. In Section 4.3, we discuss the atomic delivery protocol that ensures both output and state update are delivered atomically.

4 PROTOCOL DESIGN

Before diving into the details of the Ekiden protocol, we first describe the technical challenges involved in combining TEEs and blockchains to realize Ekiden's security goals and the building blocks we use to address them. In section 5, we formally specify the Ekiden protocol combining these building blocks.

4.1 Tolerating TEE failures

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. Ekiden is designed to tolerate such host failures, ensuring that crashed compute nodes can at most delay a request's execution.

Our high-level approach is to treat TEEs as *expendable* and *interchangeable*. Any TEE can process any query. The blockchain itself resolves any conflicts resulting from concurrency.

Formally, we model trusted hardware as a set of TEEs with distinct IDs (eids), and assume that at least one of them is not corrupted. Which TEE to invoke in a given request is exposed as a choice to the environment. This signifies that security is guaranteed regardless of how the TEEs are chosen, since the environment in UC stands in for arbitrary higher-level protocols. In Appendix E, we suggest a pragmatic solution involving a coordinator that is relied upon only for performance.

To ensure that any particular TEE is easily replaced, TEEs are stateless, and any persistent state is stored by the blockchain. We discuss later in the full protocol $\mathbf{Prot}^{\text{full}}_{\text{Ekiden}}$ (Figure 13) how TEEs can also keep soft state across invocations as a performance optimization, but we emphasize that losing such state at any point *does not* affect security.

Timer failures. Trusted hardware in general cannot provide trusted time. 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 [2][49]. Moreover server CPUs do not support trusted timer at the time of writing. Thus our protocol minimizes reliance on a timer. First, the protocol does not require TEEs to have a current view of blockchain. Specifically, instead of requiring a contract TEE to distinguish stale state from current state (without a synchronized clock, there is no definitive countermeasure to a network adversary delaying messages from consensus nodes), the protocol relies on consensus nodes to proactively reject any update based on a stale input state (a hash of which is included in the update). In Appendix D, we prove that security holds even if enclaves cannot obtain latest states. Second, to establish proofs of publication in a permissionless blockchain, we design a time-based protocol using a general secure timer (e.g. SGX timers or NTP servers over secure channels) that is secure even the timer is delayed, as explained below in section 4.2.

Side channels. Although trusted hardware aims to protect confidentiality for TEE, recent work has uncovered data leakage via side-channel attacks [21, 41, 44, 50, 58, 59, 71, 75, 84, 96, 98]. Existing defenses [26, 28, 56, 63, 77, 85–87, 89] are generally application- and attack-specific (e.g., crypto libraries avoid certain data-dependent operations [19]); generalizing such protections remains challenging. Thus, Ekiden largely defers protections to the application developer.

A range of applications, however, can be implemented with a flexible and efficient alternative side-channel defense for attested execution processor known as the Sealed-Glass Proof (SGP) model [94]. In this approach, TEEs are presumed to protect integrity, but not confidentiality, and thus sensitive data are kept within the hosts and are leakable exclusively to the data owner.

Ekiden supports the SGP model by permitting confinement of data to selected hosts, e.g., those of the data owner. This comes at the unavoidable cost of availability: keys for a TEE with confined data cannot be shared with other TEEs. Thus, availability depends on the data owner in such deployments—a drawback that may be acceptable when confidentiality is a key concern.

[2] as confirmed by SGX SDK developers at https://github.com/intel/linux-sgx/issues/161

4.2 Proof of Publication for PoW blockchains

Ekiden relies on efficient *proofs of publication* that prove to a contract TEE that an item has been stored in the blockchain. For blockchains based on Byzantine fault tolerant consensus protocols, such a proof can be simply constructed with a multisig signed by a majority of the consensus nodes. To establish proofs of publication for PoW-based blockchains, contract TEEs must be able to validate new blocks. As noted in [27], 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 works with general secure timers, e.g. TLS-enabled NTP servers, when SGX timer is not available. Due to lack of space, we relegate our proof-of-publication protocol for PoW blockchains to Appendix C.

4.3 Atomic delivery of execution results

Contract execution causes TEE to send two messages: m_1, which delivers the output to the calling client, and m_2, which delivers the state transition 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 calling client receives it. The new state m_2 is delivered once accepted by the blockchain. Rejected state transitions are not considered delivered.

Attacks without atomic delivery. 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 *replay attack* becomes possible. Since TEE cannot tell whether a user-supplied state is fresh, an attacker can replay stale states to the TEE. Although the blockchain will reject attempts to extend a stale state (see Section 5.1.1), the output is delivered to the calling client. For example, a contract implementing a budgeted differential privacy policy can be caused to overrun its privacy budget via such replay attacks. For a contract with randomized methods, an attacker may repeatedly query the compute node until she gets the desired result.

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. Note that there must be some mechanism to prevent users from using earlier states otherwise replay attacks become possible.

Blockchain-based commit. Assuming a secure communication channel between a TEE and the calling client $\mathcal{P}_i$ (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 $m_1' := \mathsf{Enc}(\mathsf{k}, m_1)$ to $\mathcal{P}_i$ over a secure channel. Once $\mathcal{P}_i$ acknowledges receipt of m_1', the TEE sends m_2 to the blockchain. Finally, after seeing a proof π_{m_2} that m_2 has been included in the blockchain the TEE sends k to $\mathcal{P}_i$.

We claim this scheme 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 $\mathbf{Prot}_{\text{KM}}$ ensures that k can be retrieved from any TEE.

Although the above protocol is conceptually simple, $\mathbf{Prot}_{\text{Ekiden}}$ adopts a more efficient variant. In $\mathbf{Prot}_{\text{Ekiden}}$, the one-time key k is replaced with a persistent key $\mathsf{k}_{\text{cid}}^{\text{out}}$ that is reused across requests. Instead of sending $\mathsf{k}_{\text{cid}}^{\text{out}}$ to calling clients, TEE obtains m_1' from $\mathcal{P}_i$ and sends the decryption. This optimization preserves atomicity.

4.4 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 [93].) Common approaches to minimizing such leakage, e.g., restricting requests based on requester identity and/or a differential-privacy budget [33, 51], require persistent counters. The monotonic counters in SGX are untrustworthy, however [65].

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.

4.5 Key management

To ensure privacy, contract states are encrypted using per-contract keys that are only known to the trusted hardware. However, the flip side is the challenge of ensuring *availability* of these keys in the event of TEE failure. In Ekiden, we replicate keys across all compute nodes. Specifically, each compute node instantiates a key manager TEE running $\mathbf{Prot}_{\text{KM}}$ (defined in Figure 10).

All key managers share a master key $\mathsf{k}_{\text{master}}$. When initialized, a key manager first retrieves contact points of standing key managers by looking up the latest "km list" entry from the blockchain and obtains $\mathsf{k}_{\text{master}}$ from them. Communication between key managers is encrypted and authenticated via secure channels established through remote attestations. To initialize the key management functionality, the first key manager generates a fresh $\mathsf{k}_{\text{master}}$ and creates on the blockchain a "km list" entry containing its identity. Subsequent key managers bootstrap by requesting to "sync" with prior key managers, finally adding themselves to "km list". We rely on the consensus layer to handle race conditions. When multiple key managers create "km list" entries at the same time, one of them will be accepted by the blockchain while others are rejected and must retry. In practice, to protect $\mathsf{k}_{\text{master}}$, key managers must be carefully implemented and side-channel free. Efficient implementations of side-channel resistant encryption are available (e.g. AES-NI).

Once bootstrapped, key managers maintain a set of keys for every contract, coordinating via the blockchain by encrypting contract keys with the shared $\mathsf{k}_{\text{master}}$. Each key $(\mathcal{E}_i, \text{type}, \mathsf{k})$ is associated with a type and $\mathcal{E}_i$, the identity of the contract TEE that creates the key. Key $(\mathcal{E}_i, \text{type}, \mathsf{k})$ can only be accessed by TEEs with identity $\mathcal{E}_i$. Roughly speaking, the identity of a contract TEE is the hash of the contract code. In practice, distinct contracts will have different identities with high probability, thus $\mathbf{Prot}_{\text{KM}}$ enforces contract-level key isolation as is essential for security.

Contract TEEs $\{\mathcal{E}_i\}_i$ can reach out to any key manager to create and retrieve keys. Upon a request by some $\mathcal{E}_i$ for a key of type type, the key manager first checks if the same key $(\mathcal{E}_i, \text{type}, \mathsf{k}_{\text{ct}})$ has appeared on the blockchain, in which case the key manager just reuses it. Otherwise, the key manager samples a fresh key k, encrypts it with $\mathsf{k}_{\text{master}}$ and stores the ciphertext on the blockchain. Only then does the key manager send k to $\mathcal{E}_i$.

For key management, this paper adopts a simple global key, $\mathsf{k}_{\text{master}}$, as this suffices in our threat model assuming no broken nodes. In future extensions, we are pursuing key management strategies under stronger threat models, using techniques such as secret sharing [39, 82].

5 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) [23] ideal functionality $\mathcal{F}_{\text{Ekiden}}$ that we relegate to Appendix A for lack of space and encourage the reader to consult. In Appendix D, we prove that $\mathbf{Prot}_{\text{Ekiden}}$ UC-realizes $\mathcal{F}_{\text{Ekiden}}$, .

5.1 Formal Specification of the Protocol

The Ekiden protocol is formally specified in $\mathbf{Prot}_{\text{Ekiden}}$ (fig. 2). $\mathbf{Prot}_{\text{Ekiden}}$ depends upon $\mathcal{G}_{\text{att}}$ and $\mathcal{F}_{\text{blockchain}}$, ideal functionalities for attested execution and the blockchain, respectively. We first specify the $(\mathcal{G}_{\text{att}}, \mathcal{F}_{\text{blockchain}})$-hybrid model in which $\mathcal{F}_{\text{Ekiden}}$ can be UC-realized by $\mathbf{Prot}_{\text{Ekiden}}$. Then we discuss the details of $\mathbf{Prot}_{\text{Ekiden}}$.

5.1.1 *The $(\mathcal{G}_{att}, \mathcal{F}_{blockchain})$-Hybrid World.*

Attested Execution. To formally model attested execution on trusted hardware, we adopt the ideal functionality $\mathcal{G}_{\text{att}}$ defined in [76]. 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 $\sigma_{\text{TEE}} = \Sigma_{\text{TEE}}.\text{Sig}(\text{sk}_{\text{TEE}}, (\text{prog}, \text{outp}))$, a signature under a hardware key sk_{TEE}. The public key pk_{TEE} can be obtained from $\mathcal{G}_{\text{att}}.\text{getpk}()$. See [76] 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: in the extended $\mathcal{G}_{\text{att}}$, if a TEE program prog generates a pair of output $(\text{outp}_1, \text{outp}_2)$, the attestation only signs outp_1, i.e. $\sigma_{\text{TEE}} = \Sigma_{\text{TEE}}.\text{Sig}(\text{sk}_{\text{TEE}}, (\text{prog}, \text{outp}_1))$. 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 [99].

Blockchain. $\mathcal{F}_{\text{blockchain}}[\text{succ}]$ (given in Appendix B.1) defines a general-purpose append-only ledger implemented by common blockchain protocols (formally defined in Figure 8 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

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

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 has been stored on $\mathcal{F}_{\text{blockchain}}$
- output ("receipt", cid)

On receive ("request", cid, inp, eid) from environment $\mathcal{Z}$:
- $\sigma_{\mathcal{P}_i} := \mathsf{Sig}(\mathsf{ssk}_i, (\mathsf{cid}, \mathsf{inp}))$
- obtains $\mathsf{pk}^{\text{in}}_{\text{cid}}$ from $\mathcal{F}_{\text{blockchain}}$; let $\mathsf{inp}_{\text{ct}} := \mathcal{AE}.\mathsf{Enc}(\mathsf{pk}^{\text{in}}_{\text{cid}}, (\mathsf{inp}, \sigma_{\mathcal{P}_i}))$
- $(\mathsf{st}'_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma) := \mathsf{request}(\mathsf{cid}, \mathsf{inp}_{\text{ct}})$
- parse σ as $(\sigma_{\text{TEE}}, h_{\text{inp}}, h_{\text{prev}}, h_{\text{outp}}, \mathsf{spk}_i)$
- assert $\mathsf{H}(\mathsf{inp}_{\text{ct}}) = h_{\text{inp}}$; assert $\mathsf{outp}_{\text{ct}}$ is correct by verifying σ
- $o :=$ claim-output(cid, st'_{ct}, $\mathsf{outp}_{\text{ct}}$, σ, 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 $\Sigma_{\text{TEE}}.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma_{\text{TEE}}, \mathsf{outp}'_{\text{ct}})$ // $\mathsf{pk}_{\text{TEE}} := \mathcal{G}_{\text{att}}.\mathsf{getpk}()$
- 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 $\mathcal{G}_{\text{att}}$ and wait for ((Contract, cid, st_0, $\mathsf{pk}^{\text{in}}_{\text{cid}}$), σ_{TEE})
- send ("write", (Contract, cid, st_0, $\mathsf{pk}^{\text{in}}_{\text{cid}}$, σ_{TEE})) to $\mathcal{F}_{\text{blockchain}}$, and 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, inp_{ct}, st_{ct})) to $\mathcal{G}_{\text{att}}$
- receive (("atom-deliver", h_{inp}, h_{prev}, st'_{ct}, h_{outp}, spk_i), σ_{TEE}, $\mathsf{outp}_{\text{ct}}$)
- // $\sigma_{\text{TEE}} = \Sigma_{\text{TEE}}.\mathsf{Sig}(\mathsf{sk}_{\text{TEE}}, (h_{\text{inp}}, h_{\text{prev}}, \mathsf{st}'_{\text{ct}}, h_{\text{outp}}, \mathsf{spk}_i))$
- let $\sigma := (\sigma_{\text{TEE}}, h_{\text{inp}}, h_{\text{prev}}, h_{\text{outp}}, \mathsf{spk}_i)$
- **return** $(\mathsf{st}'_{\text{ct}}, \mathsf{outp}_{\text{ct}}, \sigma)$

On input claim-output(cid, st'_{ct}, $\mathsf{outp}_{\text{ct}}$, σ, 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", st'_{ct}, $\mathsf{outp}_{\text{ct}}$, σ, epk_i)) to $\mathcal{G}_{\text{att}}$
- receive ("output", $\mathsf{outp}'_{\text{ct}}$, σ_{TEE}) from $\mathcal{G}_{\text{att}}$ or abort
- **return** $(\mathsf{outp}'_{\text{ct}}, \sigma_{\text{TEE}})$

Figure 2: Ekiden Protocol. The contract TEE program $\widehat{\text{Contract}}$ is defined in Figure 9, in Appendix B.

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.

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 a proof from TEE 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:

$$\text{Storage} = ((\text{Contract}, \mathsf{st}_0, \sigma_0), \{\mathsf{trans}_i\}_{i\geq 1}).$$

For a storage instance to be *valid*, each state transition must correctly reference the previous state and the attestation must verify. Formally, this is achieved by parameterizing $\mathcal{F}_{\text{blockchain}}$ with a $\mathsf{succ}(\cdot,\cdot)$ such that

$$\mathsf{succ}(\text{Storage}, (h, \mathsf{st}_{\text{new}}, \sigma_{\text{TEE}})) = \text{true}$$

if and only if $h = \mathsf{H}(\mathsf{st}_{\text{prev}})$ where $\mathsf{st}_{\text{prev}}$ is the previous state in Storage and $\Sigma_{\text{TEE}}.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma_{\text{TEE}}, (h, \mathsf{st}_{\text{new}}))$.

Attestation σ proves that st' is correctly derived from the previous state with hash h. The practical significance of $\mathsf{succ}(\cdot,\cdot)$ is that it guarantees that at any time there is a single sequence of state transitions consistent with the view of each party. It thus guarantees that the chain of state transitions is fork-free.

5.1.2 Protocol. fig. 2 presents the main protocol for Ekiden. $\mathbf{Prot}_{\text{Ekiden}}$ makes use of 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 $\mathcal{AE}(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$.

Sharing state keys. Each contract is associated with a set of keys. As discussed in Section 4.5, 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. Please refer to Figure 10 for pseudocode specifying the key manager.

Contract creation. Ekiden contracts are deterministic programs written in a general-purpose programming language. We use an TEE wrapper (Figure 9) to provide routine functionalities used by all contracts, such as state encryption, key management, etc. See Appendix B. A properly wrapped contract, denoted $\widehat{\text{Contract}}$, can be executed in a TEE. To create a contract in Ekiden, a client $\mathcal{P}_i$ calls the `create` subroutine of a compute node Comp with intput Contract, a piece of contract code. Comp loads the $\widehat{\text{Contract}}$ into a TEE and starts the initialization by invoking the "create" call. As specified in fig. 9, the contract TEE creates a fresh contract cid, obtains fresh $(\mathsf{pk}^{\text{in}}_{\text{cid}}, \mathsf{sk}^{\text{in}}_{\text{cid}})$ pair and $\mathsf{k}^{\text{state}}_{\text{cid}}$ from the key manager and generates an encrypted initial state st_0 and an attestation σ_{TEE}. The attestation proves the st_0 is correctly initialized and that $\mathsf{pk}^{\text{in}}_{\text{cid}}$ is the corresponding public key for contract cid. The compute node Comp sends $(\text{Contract}, \text{cid}, \mathsf{st}_0, \mathsf{pk}^{\text{in}}_{\text{cid}}, \sigma_{\text{TEE}})$ 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 $\mathcal{F}_{\text{blockchain}}$.

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

As specified in fig. 9, 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 $(\mathsf{st}_{\mathsf{new}}, \mathsf{outp})$. 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 4.3:

- First the contract TEE computes $\mathsf{outp}_{\mathsf{ct}} = \mathsf{Enc}(\mathsf{k}^{\mathsf{out}}_{\mathsf{cid}}, \mathsf{outp})$ and $\mathsf{st}'_{\mathsf{ct}} = \mathsf{Enc}(\mathsf{k}^{\mathsf{state}}_{\mathsf{cid}}, \mathsf{st}_{\mathsf{new}})$, 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 $\mathcal{F}_{\mathsf{blockchain}}$. 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 $\mathcal{F}_{\mathsf{blockchain}}$, the contract TEE sends the decryption of $\mathsf{outp}_{\mathsf{ct}}$ to $\mathcal{P}_i$ in a secure channel.

5.2 Security of $\mathbf{Prot}_{\mathbf{Ekiden}}$

Theorem 5.1 characterizes the security of $\mathbf{Prot}_{\mathrm{Ekiden}}$. A proof sketch is given in Appendix D.

Theorem 5.1 (Security of $\mathbf{Prot}_{\mathrm{Ekiden}}$). *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.*

5.3 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 (pk, 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 4.5).

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

6 IMPLEMENTATION

We implemented an Ekiden prototype in 7486 lines of Rust. Developers in Ekiden do not need expertise in secure computing or Intel SGX. We have implemented a compiler, which automatically builds contracts into executables that can be loaded into a compute node, using the Rust SGX SDK [31]. We leave compiling to different targets, such as secure multi-party computation for future work.

Ekiden is compatible with many existing blockchains. We have built one end-to-end instantiation of our system, *Ekiden-BT*, with a blockchain extending from Tendermint [57], which required no changes to Tendermint. Tendermint is based on the DLS Byzantine fault tolerant consensus protocol [34]. We leave implementing instantiations of Ekiden on other blockchains for future work. For blockchains where the application cannot define a custom block verification procedure, one may need to make small changes to verify attestations that prove correct computation of the TEE.

6.1 Programming Model

We support a general-purpose programming model for specifying contracts. A contract registers a mutable struct as its state, which Ekiden transparently serializes, encrypts, and synchronizes with the blockchain after method calls. Contract methods must be deterministic and terminate in bounded time. Within this model, we implemented two smart-contract programming environments. In the Rust backend, developers can write contracts using a subset of the Rust programming language, and thus benefit from a range of open source libraries. We also ported the Ethereum Virtual Machine (EVM), thereby supporting any contract written for the Ethereum platform. The system currently does not support calling contract functions from another contract. We leave this for future work.

6.1.1 *Common Components.*

RPC library. Ekiden comes with its own RPC library, which facilitates remote procedure calls into an enclave. Our compiler automatically generates client stubs to which other Rust applications can link. These stubs include logic to perform the remote attestation protocol for authenticated encrypted channels into enclaves.

Randomness. Intel SGX provides a native source of secure randomness. We expose SGX random number generation to developers as an input to their contract methods. Good on-chain randomness is challenging to obtain in blockchain systems, often leading to smart-contract vulnerabilities [29]. Among our example contracts, the poker game and Cryptokitties require secure randomness.

6.1.2 *Smart Contract Languages.*

Rust Contracts. We built a compiler for Ekiden contracts written in the Rust programming language. A single struct is used to represent persisted state. Clients remotely call methods on the struct using our RPC library. In our Rust token contract, the contract state contains a mapping from client public keys to account balances, denoted in tokens. When a client issues a `transfer` request for a given amount, the smart contract first checks if the sender's token balance is sufficient, i.e., at least the requested amount. If so, it deducts the amount from the sender's balance and adds it to the recipient's balance. Ekiden ensures that this transaction modifies contract state atomically.

Ethereum Virtual Machine (EVM) Contracts. We have ported the SputnikVM implementation of the Ethereum Virtual Machine (EVM) to run inside an Ekiden enclave. Support for EVM means automatic support existing contracts written for Ethereum. For example, in our evaluation we use Ekiden to run existing ERC20 tokens written for Ethereum. These contracts automatically inherit the guarantees of Ekiden, including secret contract state and high performance. Compared to Ethereum, execution happens off-chain on Ekiden compute nodes, rather than on the Ethereum blockchain.

6.2 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.

Machine Learning Contracts. To facilitate shared learning on secret data, we ported rusty-machine [13], a machine learning library

Application	LoC	**Secret Input/Output**	**Secret State**
Machine Learning	806	Training data, predictions	Model
Thermal Modeling	621	Sensor data, temperature	Building model
Token (Rust)	514	Transfer(from, to, amount)	Account balances
Poker	883	Players' cards	Shuffled deck
Cryptokitties	54	Random mutations	Breeding algorithm
Ethereum VM	774	Input and output	Contract state

Figure 3: Ekiden smart contracts written in Rust. For each, we specify the number of lines of code, 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 the EVM, we only include the cost of porting SputnikVM, which is 5445 lines of code. For cryptokitties, a contract written for the EVM, we only include the work specific to porting this contract.

for Rust, to run inside our contracts. We then implemented two contracts. In the *credit scoring* contract, we trained a model of credit scores based on financial records [14]. In the *medical diagnosis* contract, we trained a model predicting the likelihood of heart disease based on medical records [81]. For both applications, we generated clients that sourced data from the UCI machine learning repository [61]. Our machine learning contracts allow clients with sensitive data to train a shared model in a secure setting. Plaintext training data is never exposed outside of the contract. The contract also stores the secret trained model in its secret state, allowing other remote clients to issue inference requests. Because the model is stored on chain, new compute nodes can scale up capacity to serve inference requests without affecting correctness or privacy.

Because inference results can leak information about training data in membership inference attacks [88], differential privacy is commonly used to protect against extraction of data from the model via black-box queries. Without confidentiality-preserving smart contracts, developers would need to apply differential privacy in the local model, where noise is added to data before leaving the client device, at the cost of model accuracy [35]. Because Ekiden provides black-box confidentiality for data and computation, it allows differential privacy mechanisms in the central model, where noise is added in the contract during the training process. Thus, Ekiden enables the same privacy guarantee as the local model of differential privacy, with better accuracy and utility. We extended our machine learning contract with our own implementation of differentially-private stochastic gradient descent [15].

Smart Building Thermal Modeling. We ported an implementation of non-linear least squares, which is used to predict temperatures based on time series thermal data from smart buildings [30]. We have deployed this smart contract to train a shared model across real-time data from select buildings in Berkeley, CA. These buildings sample their temperature sensors every 20 seconds, generating data used to update the predictive model. Ekiden allows the contract to run its model while keeping the sensor data and model secret, demonstrating that our system is sufficiently responsive for highly interactive workloads in an online setting.

Tokens. The most popular kind of Ethereum contract is the ERC20 token standard. At the time of writing, ERC20 tokens together comprise a $35 billion USD market.[3] Using the Ethereum port (Section 6.1), we can run existing ERC20 token contracts. We also implemented a token contract written directly in Rust, which yields moderate performance improvement (see Section 7). In either case, Ekiden automatically provides privacy and anonymity, which the contract would not receive on the Ethereum mainnet. The secret state in the token is the `balances` mapping, which stores the account balance for each user.

Cryptokitties. Cryptokitties [1] is an Ethereum game that allows users to breed virtual cats, which are stored on chain as ERC721 tokens [3]. 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 EVM port. We verified correct behavior by reproducing real transactions from the Ethereum network, ensuring that the Ekiden application returned the same genetic results given the same inputs. The contract uses blockhash of a previous block as a source of entropy, so for this experiment we initialized our EVM state to return the appropriate hash values from Ethereum mainnet.

This example demonstrates that Ekiden can execute an Ethereum contract even when source code is not available. Further, Ekiden can provide unique benefits for games requiring secrecy or unpredictability such as Cryptokitties. These properties are difficult to achieve with Ethereum, which makes contract code and data public. For example, the Cryptokitties gene mixing algorithm has been reverse-engineered by players seeking to maximize their chance of breeding cats with rare traits [2, 5], 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.

Poker. We also implemented a poker contract, where users take turns submitting their actions to the contract, and the smart contract contains all of the game logic for shuffling and (selectively) revealing cards. Poker is a common benchmark application for blockchain systems and secure multi-party computation called *mental poker* [12, 17, 54, 55]. Ekiden is significantly more robust than these prior implementations in how it handles player aborts. In most mental poker, if a party aborts, its secret hand cannot be reconstructed by others, so the game aborts. Handling faults in secure multi-party computation requires application-specific changes to the cryptographic protocol [24]. Because Ekiden persists state to

[3] https://coinmarketcap.com/tokens/views/all/

the blockchain after each action, and can be accessed from any enclave, secret cards can still be revealed if a player aborts.

7 EVALUATION

In this section, we present evaluation results for end-to-end latency and peak throughput. We evaluated the five applications of Section 6.2: a Rust token contract **Token**, implementing an ERC20-like token in the Rust language, two Ethereum contracts, **ERC20** and **Cryptokitties**, running in the ported EVM, and two machine learning applications, **Credit** and **Thermal**. Compared to an ERC20 contract on Ethereum mainnet, Ekiden-BT can support a token contract with 600x greater throughput, 400x less latency, at 1000x less monetary cost. While we expect some mild performance degradation when deployed with a larger scale blockchain, our performance optimizations significantly reduce the effect of the blockchain's speed, as shown below. Furthermore, we demonstrate that Ekiden can efficiently support computation-intensive workloads such as machine learning applications which would be cost-prohibitive on Ethereum. We also quantify the performance gains from each of the optimizations described in Appendix E. We show that batching, caching, and a write-ahead log improve performance and reduce the network costs of synchronizing state with the blockchain.

7.1 Experimental Setup

To evaluate the performance of Ekiden-BT, we ran experiments with four consensus nodes hosted on Amazon EC2 [10] and one compute node (with a Core i7-6500U CPU with 8GB of memory) hosted locally, as EC2 does not offer SGX-enabled instances at the time of writing. Transactions are only run once on the compute node ($K = 1$). On EC2, we ran our Ekiden-BT blockchain extending Tendermint with four consensus nodes, distributed evenly across different availability zones in Oregon. Each consensus node was run on an t2.medium instance, with 2 CPU cores and 4 GB of memory. As shown in Section 7.3, we do not expect throughput performance to be significantly impacted by a larger slower blockchain, because many transactions can be compressed into a single write onto the blockchain. By separating contract execution from state agreement, these layers can work in parallel. However achieving consensus among a larger group of consensus nodes will result in higher end-to-end latencies.

7.2 End-to-End Latency

Figure 4 shows end-to-end latency for calling the token, Cryptokitties, and machine learning contracts, plotted on a log scale. For the "Ekiden-BT" plot, we start our timer when the client triggers a request and end when the smart-contract response, committed on chain, is decrypted. For read-only transactions like "Token:get" or "Credit:infer", compute nodes use a locally cached copy of state. Writes to the Ekiden-BT blockchain take up to a second to confirm. Latencies in Ekiden are dominated by the time to commit on chain. This relative cost is lower for compute-intensive workloads like machine learning training. For comparison, we include a bar ("compute-only") that measures computation time only.

For the three transactions that could be run on the Ethereum network, we plot the publicly reported block rates of the Ethereum mainnet in March 2018 [38], which represents the optimistic case

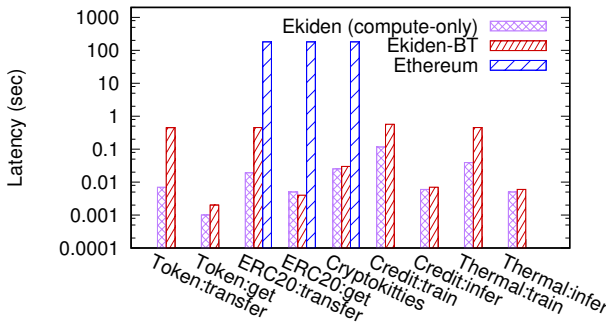

Figure 4: End-to-end latency of client requests for various contracts, plotted on a log scale. Running Rust token and ERC20 token contracts on Ekiden-BT yields transactions 2-5 orders of magnitude faster than Ethereum. Read-write transactions are dominated by confirmation times of the underlying blockchain. Read-write transactions on the Ekiden-BT blockchain take about a second. Caching avoids writes to the blockchain for read-only transactions (e.g. `get`). We only compare Ethereum for the ERC20 contract, as there are no comparable machine learning contracts on Ethereum.

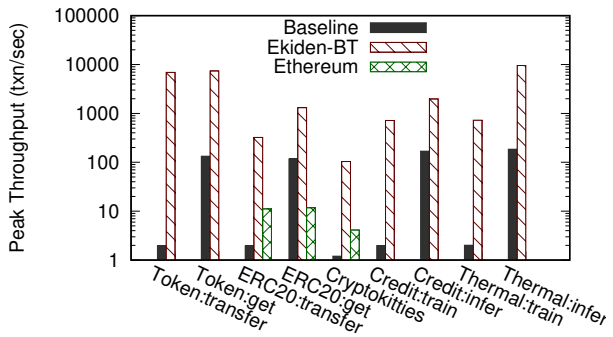

Figure 5: Throughput comparison across contracts and systems. Our baseline reads and writes to a blockchain for every request. Throughput is limited by blockchain performance. Our optimizations improve performance by 2–4 orders of magnitude over the baseline, with more advantage for read-write operations on contracts with large state (e.g. Token). In-EVM operations incur about 10x higher cost compared to our Rust token. For ERC20, we achieve 1–2 orders of magnitude higher performance than Ethereum.

that transactions are incorporated in the next block. Compared to the proof-of-work protocol used in Ethereum, Ekiden-BT has 2-3 orders of magnitude faster confirmations, in part due to the use of a faster blockchain. For the ERC20 token, which runs on the EVM in Ekiden-BT, we see similar performance to the Rust token contract, because both use the same consensus protocol.

7.3 Throughput

To measure Ekiden-BT's peak performance, we conducted an experiment with 1000 clients, each sending 100 serialized requests to a compute node. For each data point, we disregard the first and last 10% of requests, averaging the stable performance under stress. Figure 5 shows the results for the token, Cryptokitties, and machine learning contracts. For the baseline, we implement the simplest Ekiden-BT protocol, where each request triggers a full state checkpoint on our blockchain. In the "Ekiden-BT" bar, we include our optimizations, as described in Appendix E. Batching

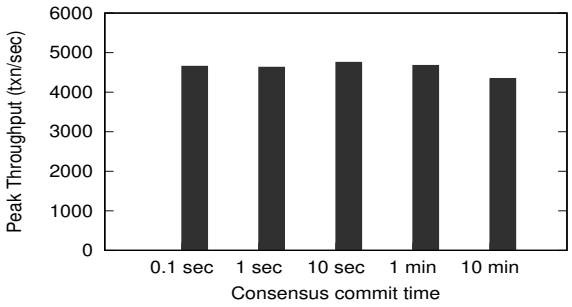

Figure 6: Peak throughput performance of token transfers under different consensus layer commit times. Because contract execution occurs in parallel to state agreement, we show that good throughput performance for a wide range of commit times on the consensus layer. We expect Ekiden to perform well on a variety of blockchains.

compresses multiple state checkpoints into a single commit on the blockchain. We then cache the latest state on compute nodes and use a write-ahead log for state updates. Our optimizations have the greatest benefit for read-write operations, like `transfer`. They have less benefit for contracts with smaller states, such as the machine learning contract with small models. Conversely, writes to the blockchain significantly impact performance for read-write transactions, compared to read-only transactions with cached state. For comparison on the transactions that could be run on the Ethereum network, we plot the publicly reported transaction throughput of the Ethereum mainnet in March 2018 [38]. Because CryptoKitties incurs higher computational cost, we can fit fewer transactions in a block due to the gas limit, compared to ERC20 transactions.

7.4 Impact of Consensus on Throughput

To understand the impact of using different consensus protocols with Ekiden, we measured peak throughput performance of token transfers as a function of the time to commit state to the blockchain. In order to simulate slower consensus protocols, we inject a variable delay for writes to the consensus nodes. Figure 6 shows that token transfers have good performance for a wide range of commit latencies seen in popular blockchains.

Because state is cached at compute nodes, compute nodes can opportunistically execute new transactions without waiting for a response from consensus nodes. Periodically, compute nodes asynchronously commit the state to the blockchain, as defined by the batch size. By separating contract execution from agreement on state, the layers can operate in parallel.

In contrast, Ethereum transactions are broadcast to all miners. Miners execute transactions sequentially, and all contracts are serialized onto a single blockchain. At the time of writing, there are 36974 ERC20 token contracts, all using the Ethereum blockchain [38]. In contrast, Ekiden parallelizes contracts across compute nodes, eliminating computational bottlenecks for better performance. However, implementation of full cross-contract calls remains future work.

7.5 Transaction Costs

In March 2018 on Ethereum, it cost 52K gas ($0.17 USD) to perform a transfer on an ERC20 token contract and 130K gas ($0.39 USD) to compute the breeding algorithm on Cryptokitties [4]. By contrast, IBM rents machines with Intel SGX processors useable by Ekiden for $260.00 per month. These can do a token transfer in 2ms and Cryptokitties breeding in 100ms, at a cost of roughly 10^{-7} and 10^{-5} dollars respectively, and a cost of 10^{-5} dollars for each call to *train* in our machine learning contract. For these contracts, the cost to commit state to the Ethereum blockchain ranges from $0.0688 for Cryptokitties to $1.92 to store a 1KB machine learning model. Because Ekiden can compress results from multiple requests into a single write to the blockchain, our system has a total cost vastly less than that of on-chain execution. There are no current public deployments of Tendermint for comparison.

8 RELATED WORK

Confidential smart contracts: Hawk [53] 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 [52]. The latter 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.

Ekiden is also closely related to and influenced by Hyperledger Private Data Objects (PDO) [6, 20] from Intel. PDOs use smart contracts, executed in SGX enclaves, to mediate access to data objects shared amongst mutually distrusting parties. While prototype code is available [6], no substantive written technical materials have been published to date. Thus the exact security model and goals are unclear, while one of the major contributions of Ekiden is its expression and realization of a formal security model. Comparison between the two systems is thus difficult. To the best of our knowledge, PDOs target permissioned settings, while Ekiden supports permisionless settings as well. Since PDOs aim at sharing of potentially big data, they rely on a security model involving off-chain storage, one that cannot be easily realized in the Ekiden setting.

The Microsoft Coco Framework [69] 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 [66] for Bitcoin that conceals transaction amounts, but not identities. Zerocash [16] as well as Cryptonote [90, 95], Solidus [25], and Zerocoin [70] 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^{2}R [32], VC3 [83], Opaque [100] and Ohrimenko *et al.* [74] leverage SGX to offer privacy-preserving data analytics and machine learning with various security guarantees, Ryoan [46] 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 multiparty computations: Several related works offer blockchain-based guarantees of computation integrity, but cannot guarantee privacy [64, 91, 92]. Other works have used a blockchain for fairness in MPC by requiring parties to forfeit security deposits if they abort [12, 17, 54, 55, 101]. 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 [42] 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.

9 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 [12, 27, 62], 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 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] CryptoKitties GeneScience algorithm. https://medium.com/@alexhegyi/cryptokitties-genescience-1f5b41963b0d
[3] EIP 721: ERC-721 Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721.
[4] Eth gas station. https://ethgasstation.info
[5] Genetics Fur Cats: Premier genetic testing services for your CryptoKitties based on machine learning and the blockchain. http://www.kitty.services/
[6] Hyperledger Private Data Objects. https://github.com/hyperledger-labs/private-data-objects.
[7] Keystone Project. https://keystone-enclave.github.io/.
[8] Neo: An Open Network For Smart Economy. https://neo.org/
[9] Altman, E. I., and Saunders, A. Credit risk measurement: Developments over the last 20 years. *Journal of banking & finance 21*, 11 (1997), 1721–1742.
[10] Amazon. Elastic Compute Cloud. https://aws.amazon.com/
[11] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative Technology for CPU Based Attestation and Sealing. In *HASP'13* (2013), pp. 1–7.
[12] Andrychowicz, M., Dziembowski, S., Malinowski, D., and Mazurek, L. Secure multiparty computations on Bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 443–458.
[13] AtheMathmo. rusty-machine. https://github.com/AtheMathmo/rusty-machine
[14] Baesens, B., Van Gestel, T., Viaene, S., Stepanova, M., Suykens, J., and Vanthienen, J. Benchmarking state-of-the-art classification algorithms for credit scoring. *Journal of the operational research society 54*, 6 (2003), 627–635.
[15] Bassily, R., Smith, A., and Thakurta, A. Private empirical risk minimization: Efficient algorithms and tight error bounds. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on* (2014), IEEE, pp. 464–473.
[16] Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., and Virza, M. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014* (2014), IEEE Computer Society, pp. 459–474.
[17] Bentov, I., Kumaresan, R., and Miller, A. Instantaneous decentralized poker. In *International Conference on the Theory and Application of Cryptology and Information Security* (2017), Springer, pp. 410–440.
[18] Bernard, T., Hsu, T., Perlroth, N., and Lieber, R. Equifax Says Cyberattack May Have Affected 143 Million in the U.S. https://www.nytimes.com/2017/09/07/business/equifax-cyberattack.html
[19] Bernstein, D. J., Lange, T., and Schwabe, P. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America* (2012), Springer, pp. 159–176.
[20] Bowman, M. Personal Communication, 2017-8.
[21] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. Software grand exposure: Sgx cache attacks are practical. *arXiv preprint arXiv:1702.07521* (2017), 33.
[22] Bünz, B., Goldfeder, S., and Bonneau, J. Proofs-of-delay and randomness beacons in ethereum. *IEEE Security and Privacy on the Blockchain (IEEE S&B)* (2017).
[23] Canetti, R. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067, 2000. https://eprint.iacr.org/2000/067.
[24] Castella-Roca, J., Sebé, F., and Domingo-Ferrer, J. Dropout-tolerant TTP-free mental poker. In *International Conference on Trust, Privacy and Security in Digital Business* (2005), Springer, pp. 30–40.
[25] Cecchetti, E., Zhang, F., Ji, Y., Kosba, A. E., Juels, A., and Shi, E. Solidus: Confidential distributed ledger transactions via PVORM. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, pp. 701–717.
[26] Chen, S., Zhang, X., Reiter, M. K., and Zhang, Y. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 7–18.
[27] Choudhuri, A. R., Green, M., Jain, A., Kaptchuk, G., and Miers, I. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. Cryptology ePrint Archive, Report 2017/1091, 2017. https://eprint.iacr.org/2017/1091.
[28] Costan, V., Lebedev, I. A., and Devadas, S. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium* (2016),

pp. 857–874.

[29] Delmolino, K., Arnett, M., Kosba, A., Miller, A., and Shi, E. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security* (2016), Springer, pp. 79–94.

[30] Dewson, T., Day, B., and Irving, A. Least squares parameter estimation of a reduced order thermal model of an experimental building. *Building and Environment 28*, 2 (1993), 127–137.

[31] Ding, Y., Duan, R., Li, L., Cheng, Y., Zhang, Y., Chen, T., Wei, T., and Wang, H. Rust SGX SDK: Towards Memory Safety in Intel SGX Enclave. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 2491–2493.

[32] Dinh, T. T. A., Saxena, P., Chang, E.-C., Ooi, B. C., and Zhang, C. M2R: Enabling Stronger Privacy in MapReduce Computation. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 447–462.

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

[34] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM) 35*, 2 (1988), 288–323.

[35] Dwork, C., Roth, A., et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science 9*, 3–4 (2014), 211–407.

[36] et al., I. B. Tesseract: Real-time cryptocurrency exchange using trusted hardware, 2017.

[37] Ethereum Foundation. Ethereum: Blockchain App Platform. https://www.ethereum.org/

[38] Etherscan. Etherscan: The Ethereum Blockchain Explorer. https://etherscan.io/

[39] Feldman, P. A practical scheme for non-interactive verifiable secret sharing. In *Foundations of Computer Science, 1987., 28th Annual Symposium on* (1987), IEEE, pp. 427–438.

[40] Fisch, B., Vinayagamurthy, D., Boneh, D., and Gorbunov, S. Iron: functional encryption using Intel SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 765–782.

[41] Götzfried, J., Eckert, M., Schinzel, S., and Müller, T. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security* (2017), ACM, p. 2.

[42] Goyal, R., and Goyal, V. Overcoming cryptographic impossibility results using blockchains. In *Theory of Cryptography Conference* (2017), Springer, pp. 529–561.

[43] Gueron, S. A memory encryption engine suitable for general purpose processors. *IACR Cryptology ePrint Archive 2016* (2016), 204.

[44] Hähnel, M., Cui, W., and Peinado, M. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA* (2017), pp. 299–312.

[45] Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., and Del Cuvillo, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy - HASP '13* (2013), pp. 1–1.

[46] Hunt, T., Zhu, Z., Xu, Y., Peter, S., and Witchel, E. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 533–549.

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

[48] Intel. Attestation Service for Intel® Software Guard Extensions (Intel® SGX): API Documentation. https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf. (Accessed on 02/06/2018).

[49] 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).

[50] Jang, Y., Lee, J., Lee, S., and Kim, T. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution* (2017), ACM, p. 5.

[51] Johnson, N. M., Near, J. P., and Song, D. X. Practical differential privacy for SQL queries using elastic sensitivity. *CoRR abs/1706.09479* (2017).

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

[53] Kosba, A., Miller, A., Shi, E., Wen, Z., and Papamanthou, C. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 839–858.

[54] Kumaresan, R., and Bentov, I. Amortizing secure computation with penalties. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 418–429.

[55] Kumaresan, R., Moran, T., and Bentov, I. How to use bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 195–206.

[56] Kuvaiskii, D., Oleksenko, O., Arnautov, S., Trach, B., Bhatotia, P., Felber, P., and Fetzer, C. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 205–221.

[57] Kwon, J. Tendermint: Consensus without mining. 2017.

[58] Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., and Kang, B. B. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security* (2017), pp. 523–539.

[59] Lee, S., Shih, M.-W., Gera, P., Kim, T., Kim, H., and Peinado, M. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security* (2017), pp. 16–18.

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

[61] Lichman, M. UCI machine learning repository, 2013.

[62] Lindell, Y., and Pinkas, B. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality 1*, 1 (2009), 5.

[63] Liu, C., Wang, X. S., Nayak, K., Huang, Y., and Shi, E. Oblivm: A programming framework for secure computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2015), SP '15, IEEE Computer Society, pp. 359–376.

[64] Luu, L., Teutsch, J., Kulkarni, R., and Saxena, P. Demystifying incentives in the consensus computer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 706–719.

[65] Matetic, S., Ahmed, M., Kostiainen, K., Dhar, A., Sommer, D., Gervais, A., Juels, A., and Capkun, S. ROTE: rollback protection for trusted execution. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* (2017), E. Kirda and T. Ristenpart, Eds., USENIX Association, pp. 1289–1306.

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

[67] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy - HASP '13* (2013), pp. 1–1.

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

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

[70] Miers, I., Garman, C., Green, M., and Rubin, A. D. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (2013), IEEE Computer Society, pp. 397–411.

[71] Moghimi, A., Irazoqui, G., and Eisenbarth, T. Cachezoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems* (2017), Springer, pp. 69–90.

[72] Möser, M., and Böhme, R. The price of anonymity: empirical evidence from a market for bitcoin anonymization. *Journal of Cybersecurity* (2017).

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

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

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

[76] Pass, R., Shi, E., and Tramer, F. Formal Abstractions for Attested Execution Secure Processors. Cryptology ePrint Archive, Report 2016/1027, 2016. https://eprint.iacr.org/2016/1027.

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

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

[79] Ron, D., and Shamir, A. Quantitative analysis of the full bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security* (2013), Springer, pp. 6–24.

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

[81] Sajda, P. Machine learning for detection and diagnosis of disease. *Annu. Rev. Biomed. Eng. 8* (2006), 537–565.

[82] Schoenmakers, B. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Annual International Cryptology Conference* (1999), Springer, pp. 148–164.

[83] Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., and Russinovich, M. VC3: Trustworthy data analytics in the cloud using SGX. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 38–54.

[84] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., and Mangard, S. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2017), Springer, pp. 3–24.

[85] Seo, J., Lee, B., Kim, S., Shih, M.-W., Shin, I., Han, D., and Kim, T. Sgx-shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA* (2017).

[86] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA* (2017).

[87] Shinde, S., Chua, Z. L., Narayanan, V., and Saxena, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016), ACM, pp. 317–328.

[88] Shokri, R., Stronati, M., Song, C., and Shmatikov, V. Membership inference attacks against machine learning models. In *Security and Privacy (SP), 2017 IEEE Symposium on* (2017), IEEE, pp. 3–18.

[89] Sinha, R., Rajamani, S., and Seshia, S. A. A compiler and verifier for page access oblivious computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 649–660.

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

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

[92] Teutsch, J., and Reitwiessner, C. A scalable verification solution for blockchains.

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

[94] Tramèr, F., Zhang, F., Lin, H., Hubaux, J. P., Juels, A., and Shi, E. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)* (April 2017), pp. 19–34.

[95] Van Saberhagen, N. Cryptonote v 2. 0, 2013.

[96] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., and Gunter, C. A. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 2421–2434.

[97] Weichbrodt, N., Kurmus, A., Pietzuch, P., and Kapitza, R. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security* (2016), Springer, pp. 440–457.

[98] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), IEEE Computer Society, pp. 640–656.

[99] Zhang, F., Eyal, I., Escriva, R., Juels, A., and Renesse, R. V. REM: Resource-efficient mining for blockchains. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 1427–1444.

[100] Zheng, W., Dave, A., Beekman, J. G., Popa, R. A., Gonzalez, J. E., and Stoica, I. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 283–298.

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

A IDEAL FUNCTIONALITY $\mathcal{F}_{\text{Ekiden}}$

The ideal functionality. We specify the security goals of Ekiden in the ideal functionality $\mathcal{F}_{\text{Ekiden}}$ defined in Figure 7. $\mathcal{F}_{\text{Ekiden}}$ 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 [23] and parameterize $\mathcal{F}_{\text{Ekiden}}$ with a leakage function $\ell(\cdot)$. We use the standard *delayed output* terminology [23] 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, i.e. a smart contract. 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 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.

$\mathcal{F}_{\text{Ekiden}}(\lambda, \ell, \{\mathcal{P}_i\}_{i\in[N]})$

Parameter: leakage function $\ell : \{0,1\}^* \to \{0,1\}^*$
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)
  notify $\mathcal{A}$ of (cid, ℓ(st′), ℓ(outp), eid)
  wait for "ok" from $\mathcal{A}$ and halt if other messages received
  update Storage[cid] := (Contract, 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 $i \in [N]$:
  (_, st) := Storage[cid]; abort if not found
  send ℓ(st) to $\mathcal{P}_i$
  if $\mathcal{P}_i$ is corrupted: send ℓ(st) to $\mathcal{A}$

Figure 7: The ideal functionality of Ekiden.

Session ID (SID).. In UC [23], each functionality instance is associated with a unique session ID (SID). The SID is essential for the composition theorem, as it ensures that concurrent instances of protocols are kept separate from each other. To reduce clutter, we omit the handling of SIDs in $\mathcal{F}_{\text{Ekiden}}$.

Corruption model. $\mathcal{F}_{\text{Ekiden}}$ adopts the standard corruption model of [23]. $\mathcal{A}$ 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.

Formal security and privacy guarantees. $\mathcal{F}_{\text{Ekiden}}$ encapsulates the following security and privacy properties. First, query execution correctly reflects the code provided by the contract creator, Second, output and new states are delivered *atomically*, i.e. output is revealed if and only if the new state is committed. We discuss implementation of this property in Section 4.3.

$\mathcal{F}_{\text{Ekiden}}$ provides privacy in the sense that neither other parties nor the adversary learns the secret input of an honest party more than allowed leakage ℓ. A client interacting with a contract learns no more than its input and output. Contract states are kept secret from all parties, $\mathcal{A}$ included, unless intentionally revealed through the output. However, contract code is revealed publicly so that users can examine it before using it. We leave supporting private contract code (e.g. by employing a similar technique as in [83]) for future work.

B SUPPLEMENTARY FORMALISM

B.1 Ideal Blockchain

$\mathcal{F}_{\text{blockchain}}$[succ]

Parameter: successor relationship 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

Figure 8: 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.

B.2 Contract TEE wrapper $\widehat{\text{Contract}}$

Contract TEE wrapper $\widehat{\text{Contract}}$

On input ("create") :
  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, , $_0$, $\text{pk}^{\text{in}}_{\text{cid}}$)
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
  $\text{st}_{\text{prev}} := \mathcal{SE}.\text{Dec}(\text{k}^{\text{state}}_{\text{cid}}, \text{st}_{\text{ct}})$
  $\text{st}_{\text{new}}, \text{outp} :=$ Contract(st_{prev}, inp, spk_i)
  $\text{st}'_{\text{ct}} := \mathcal{SE}.\text{Enc}(\text{k}^{\text{state}}_{\text{cid}}, \text{st}_{\text{new}})$
  // initiate atomic delivery
  $\text{k}^{\text{out}}_{\text{cid}} :=$ keyManager("output key")
  $\text{outp}_{\text{ct}} := \mathcal{SE}.\text{Enc}(\text{k}^{\text{out}}_{\text{cid}}, \text{outp})$
  let $h_{\text{inp}} := \text{H}(\text{inp}_{\text{ct}})$, $h_{\text{prev}} := \text{H}(\text{st}_{\text{ct}})$, $h_{\text{outp}} = \text{H}(\text{outp}_{\text{ct}})$
  return (("atom-deliver", h_{inp}, h_{prev}, st'_{ct}, h_{outp}, spk_i), outp_{ct})
On input ("claim output", st'_{ct}, outp_{ct}, σ, epk_i):
  parse σ as (σ_{TEE}, h_{inp}, h_{prev}, h_{outp}, spk_i)
  assert $\text{H}(\text{outp}_{\text{ct}}) = h_{\text{outp}}$
  send ("$\in$", cid, (st'_{ct}, σ)) to $\mathcal{F}_{\text{blockchain}}$
  receive true from $\mathcal{F}_{\text{blockchain}}$ or abort
  $\text{k}^{\text{out}}_{\text{cid}} :=$ keyManager("output key")
  $\text{outp} := \mathcal{SE}.\text{Dec}(\text{k}^{\text{out}}_{\text{cid}}, \text{outp}_{\text{ct}})$
  return ("output", $\mathcal{AE}$.Enc(epk, outp))

Figure 9: Contract TEE wrapper. Subroutine keyManager is defined in $\mathbf{Prot}_{\text{KM}}$ (Figure 10).

B.3 Protocol for Key Managers

$\text{Prot}_{\text{KM}}(\lambda, \{\mathcal{KM}_i\}_{i\in[N]}, \{\mathcal{E}_i\}_{i\in[M]})$

Key manager $\mathcal{KM}_i$:
On input ("init") from $\mathcal{Z}$:
  if found an entry ("km list", KM) on $\mathcal{F}_{\text{blockchain}}$
    send ("sync") to $\mathcal{KM}$ for all $\mathcal{KM} \in$ KM
    if any key manager replies with k
      set $\text{k}_{\text{master}} :=$ k
      try to store ("km list", KM $\cup \{\mathcal{KM}_i\}$) on $\mathcal{F}_{\text{blockchain}}$ // overwrite
      if receive "reject" from $\mathcal{F}_{\text{blockchain}}$: restart current "init" call
  else : k $\leftarrow\$ \{0,1\}^\lambda$; store ("km list", $\{\mathcal{KM}_i\}$) on $\mathcal{F}_{\text{blockchain}}$
    if receive "receipt" from $\mathcal{F}_{\text{blockchain}}$: set $\text{k}_{\text{master}} :=$ k
    else : restart current "init" call // retry on race condition
On input ("get-key", type) from $\mathcal{E} \in \{\mathcal{E}_i\}_{i\in[M]}$:
  if found an entry ($\mathcal{E}$, type, k_{ct}) on $\mathcal{F}_{\text{blockchain}}$:
    send Dec(k_{master}, k_{ct}) to $\mathcal{E}$
  else generate a key as follows:
    k $\leftarrow\$$ KGen(1^λ, keyType);
    store ($\mathcal{E}$, type, Enc(k_{master}, k)) on $\mathcal{F}_{\text{blockchain}}$
    if receive "receipt" from $\mathcal{F}_{\text{blockchain}}$: send k to $\mathcal{E}$
    else : restart current "init" call // retry on race condition
On input ("sync") from $\mathcal{KM} \in \{\mathcal{KM}_i\}_{i\in[N]}$: send k_{master} to $\mathcal{KM}$

Contract TEE $\mathcal{E}_i$:
internal subroutine keyManager(type): // called in fig. 2
  if found an entry ("km list", KM) on $\mathcal{F}_{\text{blockchain}}$:
    randomly choose a key manager $\mathcal{KM} \leftarrow\$$ KM
    send ("get-key", type) to $\mathcal{KM}$;
    wait for k with timeout T; if timeout: restart current "get-key" call
    return k
  else : **return** $\perp$

Figure 10: The protocol for a key manager. Communication between key managers and contract TEEs is implicitly **encrypted and authenticated** via secure channels established through remote attestations.

C PROOF OF PUBLICATION

The proof of publication protocol (fig. 11) involves 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.

$\mathcal{E}$ 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 $t_2 - t_1 < n_c \times \tau \times \epsilon$. The above protocol is specified in fig. 11.

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, which affects the user experience of Ekiden.

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

Parameters:
- 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 checkpiont $CB = B_n$
- **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 $\mathcal{E}$

Figure 11: Proof of Publication

Table 1: Exemplary parameters for Proof of Publication.

p	n_c	ϵ	expected no. of hashes to forge	false reject rate
10%	30	2	2^{112}	2^{-17}
10%	60	2	2^{147}	2^{-31}
20%	60	1.7	2^{113}	2^{-19}
25%	80	1.6	2^{113}	2^{-19}

For an attacker controlling p fraction of the total mining power of the blockchain network, we provide exemplary concrete parameters for n_c and ϵ in table 1. 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[4], 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 [36].

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.

D PROOF OF MAIN THEOREM

Here we give our proof of Theorem 5.1, given in Section 5.

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 outputs a random ciphertext with length $|x|$. In particular, $\ell(\cdot)$ maintains a set L and $\ell(x)$ is evaluated as follows: let $\mathcal{C}$ be the ciphertext space. If $\exists(x, r) \in L$, $\ell(x)$ returns r; otherwise, returns $r \leftarrow\$ \{c \in \mathcal{C} : |c| = |x|\}$ and add (x, r) to L. In the protocol, $\ell(\cdot)$ is realized with IND-CPA encryption schemes.

Proof. Let $\mathcal{Z}$ be an environment and $\mathcal{A}$ be a "dummy adversary" [23] 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

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

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

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 $(\mathcal{P}_i, \text{cid}, \text{Contract})$ 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 $(\text{cid}, \mathcal{P}_i, \ell(\text{inp}))$ 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 $c_{\text{inp}} = \text{Enc}(\text{pk}^{\text{in}}_{\text{cid}}, \vec{0})$ with length $\ell(\text{inp})$, 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 $\ell(\text{st}')$ and $\ell(\text{outp})$ from $\mathcal{F}_{\text{Ekiden}}$, Sim computes $c = \text{Enc}(\text{k}^{\text{out}}_{\text{cid}}, \vec{0})$ and emulates a message $((\text{"atom-deliver"}, \text{H}(c_{\text{inp}}), \text{H}(s), \ell(\text{st}'), \text{H}(c), \text{spk}_i), \sigma_{\text{TEE}}, c)$ 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", $\text{Enc}(\text{epk}_i, \vec{0}), \sigma_{\text{TEE}}$) with length $|\text{outp}|$ 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, $\text{inp}_{\text{ct}}, s$), 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 $\text{inp}' = \text{Dec}(\text{sk}^{\text{in}}_{\text{cid}}, \text{inp}_{\text{ct}})$. Then Sim sends ("request", cid, inp′, eid) to $\mathcal{F}_{\text{Ekiden}}$ on $\mathcal{P}_i$'s behalf.
- Upon receiving $\ell(\text{st})$ and $\ell(\text{outp})$ from $\mathcal{F}_{\text{Ekiden}}$, Sim computes $c = \text{Enc}(\text{k}^{\text{out}}_{\text{cid}}, 0)$ and sends $((\text{"atom-deliver"}, \text{H}(\text{inp}_{\text{ct}}), \text{H}(s), \ell(\text{st}), \text{H}(c)), \sigma_{\text{TEE}}, c)$ 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, $(\text{st}'_{\text{ct}}, \text{outp}_{\text{ct}}, \sigma, \text{epk}_i)$), Sim emulates $\mathcal{G}_{\text{att}}$ as follows: Sim first checks that $\mathcal{G}_{\text{att}}$ has previously sent outp_{ct} to $\mathcal{P}_i$ and that $(\text{st}'_{\text{ct}}, \sigma)$ 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", $\text{Enc}(\text{epk}_i, \text{outp}), \sigma$) 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 $(\text{cid}, \ell(\text{st}'), \ell(\text{outp}), \text{eid})$ from $\mathcal{F}_{\text{Ekiden}}$, Sim replies with "ok" only if the corresponding "output" message from $\mathcal{G}_{\text{att}}$ is allowed by $\mathcal{Z}$.

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 $(\text{pk}_{\text{TEE}}, \text{sk}_{\text{TEE}})$ 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 ("install", $\widehat{\text{Contract}}$), then for all sequential "resume" calls, Sim records a tuple $(\text{outp}, \sigma_{\text{TEE}})$ where outp is the output of $\widehat{\text{Contract}}$ 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 $(\text{outp}, \sigma_{\text{TEE}}) \notin \Omega$ 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 outp_{ct} before outputting it. Whenever $\mathcal{A}$ sends to $\mathcal{G}_{\text{att}}$ a "claim output" message with a input outp'_{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 $\mathcal{H} = \{\text{H}(\text{outp}^i_{\text{ct}})\}_i$ and $\mathcal{O} = \{\text{outp}^i_{\text{ct}}\}_i$ from $\mathcal{G}_{\text{att}}$ through "request" calls. If $\mathcal{A}$ sends a "claim output" message with $\text{outp}_{\text{ct}} \notin \mathcal{O}$, $\mathcal{G}_{\text{att}}$ aborts unless a $\text{H}(\text{outp}_{\text{ct}}) \in \mathcal{H}$. 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 ("create", 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 ("atom-deliver", $h_{\text{inp}}, h_{\text{prev}}, s, h_{\text{outp}}, c$) sent from $\mathcal{G}_{\text{att}}$ to $\mathcal{P}_i$ with $s = \mathcal{SE}.\text{Enc}(\text{k}_{\text{cid}}^{\text{state}}, \text{st}')$ and $c = \mathcal{SE}.\text{Enc}(\text{k}_{\text{cid}}^{\text{out}}, \text{outp}))$ in H_4 is replaced with ("atom-deliver", $h_{\text{inp}}, h_{\text{prev}}, \ell(\text{st}'_{\text{ct}}), \text{H}(c'), c'$) where $c' = \text{Enc}(\text{k}_{\text{cid}}^{\text{out}}, 0^{|c|})$.
- If $\mathcal{P}_i$ is honest, any message ("request", cid, $\mathcal{AE}.\text{Enc}(\text{pk}_{\text{cid}}^{\text{in}}, \text{inp}), s$) from $\mathcal{P}_i$ to $\mathcal{G}_{\text{att}}$ is replaced with ("request", cid, c', s) where $c' = \text{Enc}(\text{pk}_{\text{cid}}^{\text{in}}, 0)$, and any message ("output", $\mathcal{AE}.\text{Enc}(\text{k}_{\text{cid}}^{\text{out}}, \text{outp}))$ sent from $\mathcal{G}_{\text{att}}$ to $\mathcal{P}_i$ is replaced with ("output", $\text{Enc}(\text{epk}_i, 0)$).

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: $\mathcal{F}_{\text{Ekiden}}$ **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 $\mathcal{F}_{\text{Ekiden}}$.

□

E 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. As we show in Section 7, the impact is significant, up to 200% better for write-heavy workloads. 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 $\textbf{Prot}_{\text{Ekiden}}^{\text{full}}$ in Figure 13. For now, we provide a high-level description of the insight and challenges involved in each application.

Using a write-ahead log: 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.

Our first observation is that the use of a write-ahead log can reduce this expense. We modify the protocol so that only the "diff" of the state, $\Delta\text{st}_{\text{ct}}$ is written to the blockchain. To determine the current state, the enclave must parse the entire diff sequence, starting from the initial state, and applying each patch. In the token application, each transaction touches a constant number of records, hence requiring $O(M + T)$ storage complexity for T transactions if there are M users, compared to $O(MT)$ in the simple protocol.

The encryption of the diff $\Delta\text{st}_{\text{ct}}$ may leak information about which query was invoked. The token application has constant-time queries, but in general applications, it may be necessary to bound the size of queries and pad the ciphertext. Finally, 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 unpadded queries.

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 the write-ahead log grows large.

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.

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 $\Delta\text{st}_{\text{ct}}$ 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.

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.

E.1 Sample Contract Source

```
pub struct TokenContract {
  balances: HashMap<Addr, uint>,
}

impl TokenContract {
  ...
  fn transfer(&mut self, sender: &Addr, to: &Addr, value:
        uint) ->
      Result<(), Error> {
    let from_acct = self.balances.get(sender)?;
    let to_acct = self.balances.get(to)?;
    if from_acct < value {
      return Err(Error::new("low balance"));
    }
    let from_acct = from_acct - value;
    let to_acct = to_acct + value;
    self.balances.insert(sender, from_acct);
    self.balances.insert(to, to_acct);
    return Ok(());
  }
  ...
}
```

Figure 12: Token contract code.

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

Clients $\mathcal{P}_i$:

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

On input ("create", Contract) from environment $\mathcal{Z}$:
- cid := create(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 $\text{pk}^{\text{in}}_{\text{cid}}$ from $\mathcal{F}_{\text{blockchain}}$
- let $\text{inp}_{\text{ct}} := \mathcal{AE}.\text{Enc}(\text{pk}^{\text{in}}_{\text{cid}}, \text{inp})$
- $\sigma_{\mathcal{P}_i} := \text{Sig}(\text{ssk}_i, (\text{cid}, \text{inp}_{\text{ct}}))$
- $(\Delta\text{st}_{\text{ct}}, \text{outp}_{\text{ct}}, \sigma) := \text{query}(\text{cid}, \text{inp}_{\text{ct}}, \sigma_{\mathcal{P}_i})$
- parse σ as $(\sigma_{\text{TEE}}, h_{\text{inp}}, h_{\text{prev}}, h_{\text{outp}}, \text{spk}_i)$
- assert σ verifies
- assert $\exists n\ s.t.\ h^n_{\text{inp}} = \text{H}(\text{inp}_{\text{ct}})$
- $o := \text{claim-output}(\text{cid}, \Delta\text{st}_{\text{ct}}, \text{outp}_{\text{ct}}, \sigma, \text{epk}_i)$
- // 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 $(\text{outp}'_{\text{ct}}, \sigma_{\text{TEE}})$
- assert $\Sigma_{\text{TEE}}.\text{Vf}(\text{pk}_{\text{TEE}}, \sigma_{\text{TEE}}, \text{outp}'_{\text{ct}})$ // $\text{pk}_{\text{TEE}} := \mathcal{G}_{\text{att}}.\text{getpk}()$
- output $\mathcal{AE}.\text{Dec}(\text{esk}_i, \text{outp}'_{\text{ct}})$

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", $\widehat{\text{Contract}}$) to $\mathcal{G}_{\text{att}}$, wait for eid
- send (eid, "resume", ("create")) to $\mathcal{G}_{\text{att}}$
- wait for ((Contract, cid, st_0, $\text{pk}^{\text{in}}_{\text{cid}}$), σ_{TEE}) from $\mathcal{G}_{\text{att}}$
- send ("write", cid, (Contract, cid, st_0, $\text{pk}^{\text{in}}_{\text{cid}}$)) to $\mathcal{F}_{\text{blockchain}}$
- receive ("receipt", cid) from $\mathcal{F}_{\text{blockchain}}$ and **return**

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

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

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

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

On input ("create")
- 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")
- $\text{st}_0 := \mathcal{SE}.\text{Enc}(\text{k}^{\text{state}}_{\text{cid}}, \vec{0})$
- Cache[cid] = st_0 // cache state locally
- **return** (Contract, cid, st_0, $\text{pk}^{\text{in}}_{\text{cid}}$)

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

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

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

Figure 13: Enhanced Ekiden Protocol. $\text{diff}(\cdot, \cdot)$ is a function that takes in two states and output the difference.