

ACE: Asynchronous and Concurrent Execution of Complex Smart Contracts

Karl Wüst

Department of Computer Science
ETH Zurich

Sinisa Matetic

Department of Computer Science
ETH Zurich

Silvan Egli

Department of Computer Science
ETH Zurich

Kari Kostianen

Department of Computer Science
ETH Zurich

Srdjan Capkun

Department of Computer Science
ETH Zurich

ABSTRACT

Smart contracts are programmable, decentralized and transparent financial applications. Because smart contract platforms typically support Turing-complete programming languages, such systems are often said to enable arbitrary applications. However, the current permissionless smart contract systems impose heavy restrictions on the types of computations that can be implemented. For example, the globally-replicated and sequential execution model of Ethereum requires gas limits that make many computations infeasible.

In this paper, we propose a novel system called ACE whose main goal is to enable more complex smart contracts on permissionless blockchains. ACE is based on an off-chain execution model where the contract issuers appoint a set of service providers to execute the contract code independent from the consensus layer. The primary advantage of ACE over previous solutions is that it allows one contract to safely call another contract that is executed by a different set of service providers. Thus, ACE is the first solution to enable off-chain execution of interactive smart contracts with flexible trust assumptions. Our evaluation shows that ACE enables several orders of magnitude more complex smart contracts than standard Ethereum.

1 INTRODUCTION

After Bitcoin [30] became the first permissionless cryptocurrency and popularized Blockchain technology, Ethereum [36] extended the concept to smart contracts. Smart contracts are programs that allow *contract participants* to load blockchain-based currency to a contract-controlled account. The contract’s code defines rules and conditions under which its funds will then be transferred out of the contract, typically to one of the contract participants.

It is commonly argued that smart contracts provide significant advantages over traditional financial instruments. One advantage is their *generality*: since smart contracts are programmable, they should enable arbitrary financial applications on blockchains. The second is improved *transparency*: smart contract code and execution is verifiable by anyone from the public blockchain, unlike in traditional business applications. And the third is strong *liveness*: contract execution is not controlled by one or few entities, but rather by a large permissionless system.

While these properties are indeed an attractive offering for building novel financial applications, the currently available smart contract systems fail to realize them fully. For example, Ethereum, in principle, allows arbitrary contracts through a Turing complete language, but in practice it heavily limits contract complexity. Such limits are necessary, because Ethereum is based on *sequential* and

globally-replicated execution model, where every miner should execute all contract calls of the latest block before finding the next.

To prevent excessive delays, Ethereum uses a metric called *gas* to measure execution complexity. If a contract call surpasses a specified limit, its execution is aborted which effectively limits the types of computation that can be implemented. For example, the simple task of sorting 256 integers with selection sort requires 17M gas while the current limit for each block is 8M. Quick sort hits the limit after 2,000 elements. More complex applications, such as decentralized blockchain oracles [33], become quickly infeasible.

The main goal of our work is to lift such execution limits and to enable safe execution of *more complex* smart contracts for permissionless blockchain systems like Ethereum, while maintaining smart contracts’ transparency and good liveness.

Previous work. Because Ethereum’s execution model is both expensive and slow, recent research has explored alternative ways to execute contracts on permissionless blockchains.

YODA [14] proposes a *randomly-sampled* model that supports asynchronous execution of contract calls. For each contract call, a subset of miners are chosen randomly to execute the contract code independently of the mining process and return the results in form of a new transaction. Arbitrum [25] suggests a model where the contract creator *appoints* a small set of verifiers who should check execution integrity off-chain. Thus, Arbitrum replaces the sequential and globally-replicated execution model of Ethereum with one where different contracts can be executed (and thus verified) asynchronously by only few parties who may have an interest in the contract’s integrity. Ekiden [11] uses Trusted Execution Environments (TEEs), namely SGX enclaves, to execute smart contracts. The main motivation of Ekiden is to enable *confidential* contracts, rather than complex ones, but since execution is decoupled from consensus, Ekiden can serve this purpose as well.

To draw an analogy to distributed database systems, in the above solutions, contract calls are transactions that are executed in separate *partitions* that are either standard PCs or TEEs, chosen based on random sampling or by appointment. However, none of the above works addresses safe and concurrent execution of transactions that cross partitions, i.e., cases where a contract executed in one partition calls a contract managed by another partition.

Many current Ethereum contracts call other contracts and change their state. Executing such contracts without *concurrency control* can leave the contracts in an inconsistent state, as is well-known from distributed databases [5]. Thus, none of the previous solutions enable safe execution of many current Ethereum contracts.

Our solution. In this paper, we propose a novel system that we call ACE (for *Asynchronous and Concurrent Execution of Complex Smart Contracts*). ACE combines elements from previous systems. As in YODA, ACE executes contracts asynchronously off-chain, decoupled from the consensus process. Similar to Arbitrum, execution is performed by a set of service providers that are appointed by the contract’s issuer. And, like in Ekiden, ACE runs contracts inside SGX enclaves to increase trust on each service provider.

Such model allows execution of complex contracts without slowing down the consensus process and enables flexible trust assumptions and liveness guarantees. Contract issuers can choose an appropriate set of service providers for each contract separately and users are free to choose which service providers they trust for contract safety and liveness. This execution model allows to use a digital currency provided by a permissionless blockchain, with its transparency and integrity guarantees, while benefiting from higher efficiency and more flexibility from the off-chain execution.

ACE has also noteworthy differences to previous systems. Unlike Arbitrum, where service providers must reach unanimous agreement on the execution results (or the system falls back to an expensive on-chain verification), we enable more flexible verification, where execution results are accepted if at least t out of n service providers report the same result. In contrast to Ekiden, ACE does not require that all TEEs are fully trusted for integrity, and our model is resistant to compromise of individual SGX platforms which has been shown to be a relevant threat [9, 10]. And finally, unlike previous solutions, ACE supports cross-partition transactions, and thus enables safe and efficient execution of contracts that interact across service provider boundaries.

Concurrency control has been studied extensively in the context of database systems. Classical solutions include two-phase locking [16] and optimistic concurrency control [27]. Recent research has proposed *deterministic* alternatives [35], where transactions are pre-ordered before execution. The main benefit of such systems is that they avoid the need for expensive distributed commit protocols that require multiple rounds of communication between the involved partitions. Our observation is that the deterministic approach provides also a good basis for off-chain smart contract execution and we tailor such schemes for our purpose.

In particular, we augment the typical block structure with separate *ordering* and *result sections*. When a miner creates a new block, it pre-orders transactions for execution. Service providers examine new blocks and if the ordering section contains transactions with calls to their contracts, they execute them off-chain. Typically, concurrency control solutions require expensive synchronization between partitions, like distributed commit protocols. We avoid this cost by leveraging the pre-assigned order and broadcasting not yet committed transaction execution results to the peer-to-peer network. Once a transaction is fully executed, service providers sign a *state-change transaction*. Miners accept state changes signed by at least t service providers and include it in the result section of one of the next blocks which commits the transaction.

Main results. We implemented a prototype of ACE that has two main components: the concurrency-control protocol and EVM implementation that runs inside SGX enclaves.

The primary goal of ACE is to enable more complex contracts. To evaluate this aspect, we simulated the asynchronous off-chain execution of ACE and the sequential and globally-replicated execution used in Ethereum. Our results show that ACE enables contract calls that take *several orders of magnitude* longer to execute (e.g., minutes instead of milliseconds) with similar system throughput. Therefore, ACE significantly improves the first main advantage of smart contracts, the ability to implement arbitrary financial applications on blockchain currencies.

The other two common advantages of smart contracts are improved transparency and liveness. ACE preserves the transparency, because contracts are executed by SGX enclaves and any entity who examines the chain can verify enclave code integrity through SGX attestation that is publicly recorded on the chain. Naturally, ACE cannot provide similar liveness guarantee as Ethereum, since its contracts are executed by a set of pre-named entities instead of a permissionless network. However, ACE still provides a strong and adjustable liveness guarantee, where contract calls are guaranteed to complete, as long as at most $n-t$ service providers are unavailable.

Contributions. To summarize, this paper makes the following contributions:

- *ACE system.* We propose novel a system called ACE for off-chain execution of complex smart contracts on permissionless blockchains like Ethereum. The key features of ACE are its flexible trust model and safe concurrency control protocol.
- *Evaluation.* We show that ACE enables several orders of magnitude more complex contract calls compared to Ethereum.

The rest of this paper is organized as follows. Section 2 provides background and Section 3 explains our problem. Section 4 gives an overview of ACE and Section 5 explains it in detail. Section 6 contains the security analysis and Section 7 performance evaluation. Section 8 provides discussion, Section 9 reviews related work and Section 10 concludes the paper.

2 BACKGROUND

In this section we provide background information on smart contracts, Intel SGX, and concurrency control.

2.1 Ethereum Smart Contracts

The Ethereum protocol [37] defines a system for a decentralized state machine by using the blockchain paradigm. It allows participating nodes to write and execute contracts that are a collection of functions and global storage variables. Whenever a function is called, the system’s state may change. This can be achieved through transactions that define the target contract and the transaction sender. Each block defines a specific system state and the transactions it contains represent the transition from the state of the block’s predecessor to its current state.

The Ethereum state consists of two types of account objects: *external accounts* controlled by a private key and *contract accounts* controlled by the contract code. An external account can send messages to other accounts in the form of transactions. If a contract account receives a message call, its code is executed which might read or write to the account’s state, send messages to other contracts or create new contracts.

Miners collect transactions from a peer-to-peer network and add them to a new block that is distributed back to all miners who *execute all included transactions to verify them*. To keep verification fast, every used resource (computation, storage) defines a price, measured in *gas*, and every transaction has a gas limit. The gas used during execution is subtracted from the account balance of the transaction sender and added to the block miner’s account.

Ethereum contracts can be written in high-level languages, such as Solidity [1], that are compiled into a low-level, stack-based, Turing-complete language called EVM (Ethereum Virtual Machine) Code. Executing the EVM Code is handled by an EVM interpreter.

Ethereum’s trust assumptions follow the general security model of PoW blockchains [18]. It is assumed that no malicious entity controls a majority of the mining power and message dissemination works sufficiently well (e.g., eclipse attacks are not possible [21]). For contracts, Ethereum’s trust model is *contract-specific*, i.e., users can freely choose which contracts they decide to trust. For example, if a user decides to send money to a particular smart contract, then with this action the user *implicitly* agrees with the conditions and the logic specified in that contract’s code. A user that decides to trust one contract does *not* have to trust other contracts that run in the same system.

2.2 Intel SGX

Trusted execution environments (TEEs) enable protected applications on untrusted computing platforms. Instead of having to trust the entire platform that typically consists of millions of lines of code, the main goal of TEEs is to enable services whose security relies only on the protected application code that can be much smaller.

Intel’s SGX [13, 23] is a TEE architecture that is widely deployed to commodity PCs. In SGX, TEEs are realized as *enclaves* that run with user-level privileges, but their data and execution is isolated from any privileged code on the same platform, such as the untrusted OS. The two main security properties of SGX are *enclave code integrity* and *data confidentiality*. Data confidentiality is achieved through access control checks that are implemented inside the CPU allowing only the enclave’s code to access its data. The CPU encrypts enclave data whenever it’s moved outside the processor.

Enclave code integrity is realized through remote attestation [24]. Each SGX CPU has an attestation key, installed at the time of manufacturing, and it uses this key to produce a signed attestation statement containing measurements of enclave’s initialization sequence, code, and issuer. The enclave can generate another key and attach the public part of this key to the attestation statement which allows remote entities to establish secure connections to the attested enclave and verify data signed by the attested enclaves.

Attestation is an interactive protocol that involves three entities: the attested SGX platform, the remote verifier and Intel’s attestation service (IAS). The attestation protocol is based on group signatures and it is anonymous such that the remote verifier cannot identify the attested hardware platform. However, the remote attestation protocol supports a *linkable mode* of attestation that allows the remote verifier to check if the attested enclave code has been previously attested on the same CPU [2].

2.3 Concurrency Control

Distributed database systems process transactions using multiple database servers or partitions [12]. Transactions consist of multiple operations (reads and writes) and the execution order of these operations is called *schedule*. If two transactions access the same resource simultaneously, they may conflict. To ensure safe concurrent processing, the following ACID properties are often required [20]:

- *Atomicity* means that a committed transaction is either completed in full or aborted (either all or none of the reads and writes are applied).
- *Consistency* implies that every executed transaction can only change the state of the database from one consistent state to another.
- *Isolation* defines how the changes made by one transaction become visible to other transactions. *Serializability* is a strong level of isolation that ensures that the outcome of parallel-executed transaction is equal to one where transaction had been executed sequentially [5].
- *Durability* guarantees that all successfully committed transaction withstand possible database crashes.

Several concurrency protocols are known. Two-phase locking is a classical solution that enables serializable schedules [16] (strong isolation). To ensure atomicity and durability, the transaction results must be committed either on all or none of the involved resources that may cross partition boundaries. The involved partitions run a *distributed commit protocol* that requires multiple rounds of communication (two rounds are required for crash faults and three rounds for Byzantine faults). Such distributed commit protocol is typically considered the primary cost of distributed transaction processing.

Another classical concurrency control solution are *optimistic* protocols [27]. In such systems, distributed transactions are first executed without locking and afterwards the transaction managers examine if conflicts took place. If this is the case, all execution results must be rolled back which incurs a significant cost. If none of the executed operations were in conflict, the execution results can be committed (using a commit protocol).

Recently, *deterministic* concurrency control has been proposed as an alternative [32]. In deterministic systems, transactions are first ordered by a (centralized or distributed) sequencing layer followed by a simple and deterministic locking protocol. Executions must follow this order regardless of possible crashes. The primary benefit of deterministic concurrency control solutions is that they avoid the high cost of multi-round distributed commit protocols.

A transaction schedule is called *recoverable* if abort of one transaction never leaves other transactions in inconsistent state. A transaction schedule is called *cascading* or *cascadeless* recoverable depending if abort of one transaction requires rollback of other (not yet committed) transactions or not.

3 PROBLEM STATEMENT

In this section, we motivate our work, discuss limitations of previous solutions, and define requirements for our solution.

3.1 Motivation

In Ethereum, all contracts are executed one after another by all miners that participate in the consensus process. Such a *sequential*

and *globally-replicated* execution model ensures the integrity of the blockchain’s state that includes the state of all contracts, assuming that the trust assumptions of Ethereum’s consensus mechanism hold (see Section 2.1).

Execution integrity requires that all miners re-execute each transaction code to verify the correctness of the result. This has two main problems. First, it represents a high cost for the miners in terms of both the power consumption and time.¹ Second, it inherently limits the achievable *throughput* of the system and the *types of contracts* that can be implemented. To provide good throughput, the block interval must be kept small, and therefore the execution time of Ethereum contracts is limited. Such time limits have a direct effect on the allowed computational complexity of contracts.

Currently, the gas limit for each Ethereum block is approximately 8M gas. To put this into perspective, even the simple task of sorting 256 integers using insertion sort requires twice as much. Thus, although Ethereum is based on a Turing-complete language that in principle allows the execution of arbitrary smart contracts, the *arbitrary* part is severely limited in practice. For example, contracts that perform heavy cryptographic operations (e.g., for data feeds like TLS-N [33]) or use machine learning models are infeasible.

Our main goal is to provide a solution that allows safe execution of *more complex* smart contracts in permissionless blockchain systems like Ethereum. With “complex” we mean contracts where a single call may take seconds or minutes to complete, in contrast to Ethereum where calls can take few milliseconds at most. We argue that this would enable various new types of contracts, such as ones that perform cryptographic operations that are not natively supported by EVM. We assume that heavier computations that take hours or days to complete, are rarely needed in smart contracts.

3.2 Limitations of Previous Solutions

Recent research has explored alternative ways to execute smart contracts. In this section we summarize the main limitations of these approaches. Section 9 surveys additional related work.

In YODA [14] a subset of consensus participants is *randomly sampled* to execute and thus verify each contract call. The main limitation of systems like YODA is that each smart contract call needs multiple rounds of execution by separate subsets and the sampled subsets need to be relatively large (e.g., hundreds of nodes) to reduce the probability of cheating. Such approaches can reduce the required degree of execution replication compared to standard Ethereum (e.g., from thousands of nodes to hundreds), but they still require a great amount of redundancy, a multi-round execution process, large communication overhead, and the probability of integrity violation is not negligible. In addition, systems like YODA require unbiased and distributed random beacon. State-of-the-art beacon protocols, like RandHound [34], require highly expensive initialization routines (that need to be repeated when new participants join or leave) and significant communication for every periodic random value.

In Arbitrum [25] the execution of smart contracts is performed asynchronously off-chain by a set of managers that are *appointed* by the contract creator. Execution results are accepted by miners, if

¹Miners could be spent the same time to mine a new block and acquire monetary benefits. The only direct benefit verification is “the common good” of the cryptosystem itself, and therefore miners are often tempted to skip the execution and verification, accepting the result as given, which is often referred to as “Verifier’s Dilemma” [28].

all appointed managers signed the same results. If fewer managers sign results, the transaction does not immediately get accepted and a challenge period is entered instead. The signing managers and the disputing challenger are required to post a deposit and then one of them proves correctness of his results using a protocol that is logarithmic in the length of the execution trace. During the challenge period the contract cannot make progress.

Ekiden [11] provides *confidentiality* for smart contracts and decouples contract execution from the consensus process by executing contract off-chain in SGX enclaves. While this approach also enables execution of more complex contracts, Ekiden requires that all enclaves are trusted, since compromising one enclave enables the adversary violate contract integrity arbitrarily. Recent research has shown that TEE compromise is a practical threat [8, 9, 17, 19, 29, 38].

Each of the above solutions executes contracts in separate *partitions*. However, none of the solutions supports safe cross-partition transactions, i.e., contract calls from one execution partition to another. Since many Ethereum contracts make calls that write to another contract, these solutions therefore cannot execute them safely.

ChainSpace [3] increases the transaction throughput through sharding. The backbone infrastructure is divided into shards and every smart contract author can designate a trusted shard to execute its smart contract, thus parallelizing execution. However, to maintain integrity across cross-shard executions all shards need to be honest and have mutual trust.

3.3 Requirements

Given the above limitations of previous approaches, we define two main requirements for our solution.

R1: Flexible trust model. Our first requirement is a practical and flexible trust model for contract execution. We want to maintain the trust model of Ethereum where each user can decide which contracts he trusts. We want to avoid systems like Ekiden, where each entity that executes the contract code needs to be fully trusted, or solutions like ChainSpace, where all partitions need to fully trust each other, because such solutions offer poor security in the permissionless setting. We also want to avoid solutions like Arbitrum where all entities need to agree on contract execution results, because such approaches provide poor availability and performance.

R2: Efficient concurrency control. Our second requirement is that we want to enable contracts that can make safe cross-partition calls. In particular, our solution should ensure the typical ACID-properties of distributed database systems with serializability (to mimic the sequential execution of Ethereum). Our goal is to avoid expensive distributed commit protocols between partitions and enable recoverable schedules. Finally, we want to optimize the common and benign case, so that transaction execution is safe with minimal communication overhead in the absence of errors and attacks (recovery from error conditions can be more expensive).

4 ACE OVERVIEW

In this section, we provide an overview of our solution ACE.

4.1 System Model

Figure 1 shows our system model. ACE is based on a permissionless setting where any entity can freely take one of the following roles.

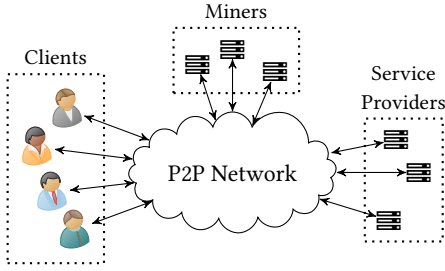


Figure 1: ACE system model. Clients deploy and call contracts by broadcasting transactions. Miners organize transactions into blocks. Service Providers execute contracts and post results to miners that will add them to new blocks.

Clients are equivalent to users in systems like Ethereum. Clients can call smart contracts by broadcasting a signed transaction that specifies the called contract. Client identities are pseudonyms, i.e., identifiers that clients can freely choose and change. Clients can also deploy new smart contracts as *contract creators* by broadcasting a transaction that contains the contract code and its specification. The specification defines the service providers who should execute contract calls and acceptance criteria for execution results.

Miners are responsible for collecting and ordering transactions to blocks, similar to systems like Ethereum.² The primary difference between ACE miners and Ethereum miners is that the former do not execute contract calls. Instead, they only add state changes from already executed contract calls to new blocks.

Service providers are responsible for executing contract calls using SGX enclaves. After execution is complete, service providers communicate the execution results back to miners. Service providers must register on the chain, before they can execute smart contracts. The identity of each service provider is their enclave public key.

Network. We assume asynchronous communication between the above entities (i.e., message delivery may get delayed and messages may arrive out of order). However, network-level attacks like node eclipsing [21] are not possible.

4.2 Trust Assumptions

We consider *clients* fully untrusted, similar to most permissionless blockchain systems. Regarding *miners*, we follow the standard trust assumptions of Ethereum and other PoW systems (see Section 2.1). Any individual miner can be malicious, but miners are collectively trusted for consensus and forks in the chain remain shallow.

Service providers can be reputable entities such as well-known companies who run contracts in exchange for a small service fee or non-profit organizations who run contracts for public good. Alternatively, service providers can be private people who volunteer to run contracts as a community effort. Each service provider has a TEE, namely SGX processor. We consider the OS in all service providers untrusted, due to their large complexity and possibility of insider attacks like malicious administrators. In addition, we assume that some individual TEEs may get compromised, as is shown

²We focus on Ethereum that uses Proof-of-Work consensus. Therefore, we use the term *miner* to refer to a consensus participant, although our solution is orthogonal to the used consensus scheme. ACE could be used also with other permissionless consensus schemes like Proof of Stake.

to be possible by recent SGX attacks (see Section 2.2). We assume that Intel runs the SGX attestation service (IAS) correctly.

While these assumptions suffice to ensure integrity of smart contract execution, to ensure contract availability, we additionally assume that some of the service providers remain responsive (see Section 6.2 for details).

4.3 Execution Model

The starting point of our solution is an *off-chain* contract execution model that is similar to Arbitrum [25]. At the time of deploying a new contract, the contract creator appoints n service providers who should execute each call of that contract. The contract creator specifies the identities (enclave public keys) of these service providers in the transaction that creates the contract.

Instead of requiring that all service providers have to agree on the execution results, we follow a different approach and allow the contract creator to define a *quorum* of t service providers that are required for acceptable execution result. The identities of the n service providers and the security parameter t together define the *trust level* of the contract.

When a contract call is executed, if at least t out of the specified n service providers report the same execution result, the state change will be accepted by the miners. All service providers execute the smart contract calls inside attested SGX enclaves. Enclave attestation takes place during service provider registration and both the enclave code and the attestation evidence is recorded on the chain so that they can be publicly verified.

It may seem counterintuitive to allow the contract creator to choose the trust level for each contract separately. However, we emphasize that this design decision is inline with the existing trust model of Ethereum, where contract participants implicitly agree with the contract *specification*, defined by its code, when they start interacting with it (see Section 2.1). In ACE, this specification additionally includes the identities of n service providers and the quorum size t . Similar to Ethereum, any party that does not use a particular ACE contract is not affected by any results of that contract.

Our model can be seen as a *hybrid* of permissionless and permissioned systems. The decoupled off-chain execution provides more efficiency and allows for a flexible trust model in which a permissioned set of validators is appointed for each contract separately. The smart contracts that are executed off-chain can directly utilize the underlying digital currency that is implemented by the permissionless blockchain which provides transparency and double-spending protection. The trust model is flexible as it allows various applications with separate security requirements to co-exist in the same system. For example, even if all validators of one contract are fully compromised, only the funds of that contract are affected and the integrity of the currency is still guaranteed for all other parties. Additionally, using a public blockchain to connect clients to service providers enables simple service discovery.

4.4 Contract Execution Overview

Next, we provide an overview of contract execution in ACE. Similar to Ethereum, clients broadcast transactions to the peer-to-peer network. We assume that every ACE transaction contains at least one write operation, as read-only computations are possible simply

by examining the blockchain without issuing a new transaction. The broadcasted transactions are ordered and collected into blocks by miners. If a transaction does not include any contract call (i.e., it only transfers money from one account to another), the miners immediately execute the transaction *on-chain* and record it to a *result section* of the block.

If a transaction calls a contract, the miners perform basic checks (e.g., correct transaction format) and then include the transaction in the *ordering section* of a block, but importantly do not execute the contract call. The new block is broadcasted to the network. Once the service providers appointed for this contract receive it, they execute the contract call code inside attested enclaves. After executing the call, enclaves sign so called *state change transactions* and publish them back to the network. Once the miners receive signatures from t of the appointed service providers on the same set of state changes, they include the state changes and the corresponding signatures in the *result section* of a future block.

In distributed systems, when transactions are executed by multiple entities, the entities typically need to agree on the order of execution and whether execution was successfully completed on all nodes before committing the results. In ACE, the service providers do *not* run an agreement protocol, because transaction ordering is already performed by the miners. Similarly, the miners ultimately decide whether a transaction is committed (by including it to the next block), and therefore the service providers do not have to run a multi-round commit protocol with Byzantine agreement. Thus, ACE can be seen as a variant of deterministic concurrency control (see Section 2.3), where transactions are pre-ordered to avoid expensive distributed commit protocols.

In distributed databases, transactions are typically committed immediately after their execution to avoid so called *dirty reads* where one transaction changes the state of a resource and another transaction reads the changed state before the first transaction is committed. If the first transaction cannot commit (e.g., due to a crash), the second transaction must be rolled back to preserve consistency.

In off-chain contract execution transaction commits have inherent latency, due to the infrequent block generation rate of permissionless blockchains. Thus, dirty reads are in practice unavoidable.³ ACE ensures consistency in the presence of potential dirty reads as follows. When miners include signed state changes to a new block (i.e., commit transactions), they need to ensure that the state changes are added *per contract* in the same order as they were listed in the ordering section previously. That is, a transaction that appears before another transaction in the ordering section needs to either have its state changes committed or be invalidated (see Section 5.7) before state changes of the second transaction can be committed to a block if the two transactions affect the same contract state.

4.5 Concurrency Control

A common way to execute cross-partition transactions in distributed systems is that transaction managers (service providers) execute those operations that access their resources (contract storage variables) and they would send a request to another partition when

³In principle, it would be possible to design a system, where each executed transaction is committed to a new block before the execution of the next transaction is allowed to proceed. However, such system could process, in the worst case, only one transaction per block, and be completely impractical.

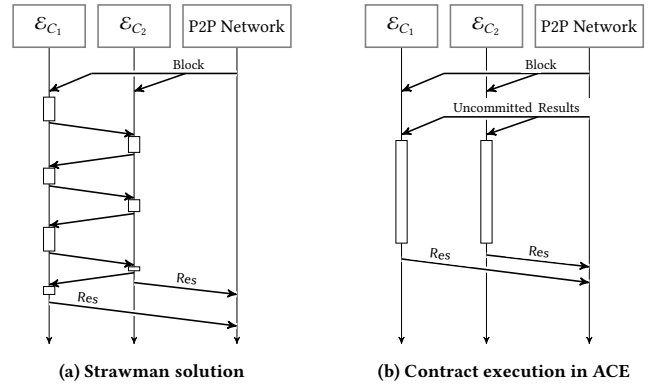


Figure 2: On the left we have the strawman solution for concurrent execution where each subcall requires communication between partitions. On the right we have our replicated model where all involved service providers execute the complete transaction preceded by an initial update of the other contracts tentative state based on uncommitted results.

the executed transaction reads or write to a resource that is controlled by the other partition. The contacted service provider would then execute the requested operations and send back a result. This common approach is illustrated in Figure 2 on the left. However, a significant drawback of this approach is that it may require lots of communication. A single smart contract call may involve several subcalls to many other contracts which would in turn mean many rounds of communication between the involved service providers.

In ACE, we follow a different approach. Instead of forcing the service providers to communicate before and after every subcalls, all involved service providers execute the full transaction and publish their results afterward, as illustrated on the right in Figure 2. To enable such cross-partition calls, the involved service providers only need to *update tentative states* of involved contracts before executing the call. This is necessary since some results of previous contract calls may not have been included in the blockchain yet.

To do this, service providers listen to state change results that are broadcast to the P2P network. If the required state updates (results for relevant preceding transactions) are not received within a specified timeout, the service provider can issue an abort for the transaction (see Section 5.7). Otherwise, the service provider can start executing the transaction. We use the P2P network for exchanging the tentative contract states, because it makes deployment of ACE easier. In particular, the service providers do not need to establish direct communication links between each other which gives them the freedom to migrate, change IP addresses etc.⁴

Such cross-partition contract call model requires that every transaction has to list all possibly involved contracts. In ACE, clients pre-run transactions as a *reconnaissance step* to determine the set of contracts and service providers involved and attach this information to the broadcast transaction (see Section 5.4). This enables service providers to know which contract calls they need to execute and with which other service providers they need to exchange state.

⁴If some service providers communicate often, they can ensure that they are well connected within the P2P network for reduced latency, similar to what large mining pools do to ensure low latency.

Based on this tentative state, all of the involved service providers execute the full call chain.

After the execution has finished, each service provider enclave signs the results (list of state changes) and sends them to the P2P network. Once miners have received the threshold number of signatures on the results for each of the involved smart contracts, they will include them in the results section of the next block (again, following the per-contract order from a previous ordering section).

5 ACE DETAILS

In this section we describe the ACE system in detail.

5.1 Block Structure

ACE blocks consist of three sections: The *Registration section* is used to add new service providers, the *Ordering section* is used to list and order transactions before execution, and the *Result section* lists the state changes of executed smart contract calls.

5.2 Service Provider Registration

To register, the service provider runs the SGX remote attestation protocol with its own enclave that runs the EVM. The attestation is performed using the linkable mode of attestation that prevents the same SGX processor from being registered multiple times (see Section 2.2). The enclave generates a new key pair and includes the public part of this key in the attestation which enables external parties to verify data signed by the enclave.

The service provider includes the attestation result to the registration transaction, and the miners include it to a registration section of a new block. This allows any client who wants to participate in a smart contract to verify the remote attestation of each involved service provider enclave. Later on, in order to prove the correct execution of smart contracts, the enclave signs the contract code running in the EVM along with the result with its private key, thus proving honest execution of a particular contract inside of the attested environment.

5.3 Contract Deployment

Deployment of new smart contract C is shown in Figure 3. ① Client chooses an executing set \mathcal{E}_C out of the registered service provider enclaves \mathcal{P} and sends a *deployment request* R_D to each service provider $P \in \mathcal{E}_C$ asking them to act as executors for this contract deployment. The request contains the contract code, parameters n and $t \leq n$, and the identities (registered enclave public keys) of the chosen service providers. ② Every service provider that agrees to be part of the executing set returns a signature $\sigma_P(R_D)$ on the deployment request. ③ The client creates a *deployment transaction* Tx_D , containing the list of service providers from \mathcal{E}_C , the deployment request R_D , and the set of signatures $\Sigma_{\mathcal{E}_C}(R_D) = \{\sigma_P(R_D) \mid P \in \mathcal{E}_C\}$ from all members of \mathcal{E}_C . The client sends the deployment transaction to the network, where miners verify the signatures and, if valid, include it in the ordering section of a new block. ④ Once the transaction is included, the members of \mathcal{E}_C execute the constructor of the contract like a normal contract call, as described below. ⑤ The service providers send the result Res to the network to be included in the result section of a future block.

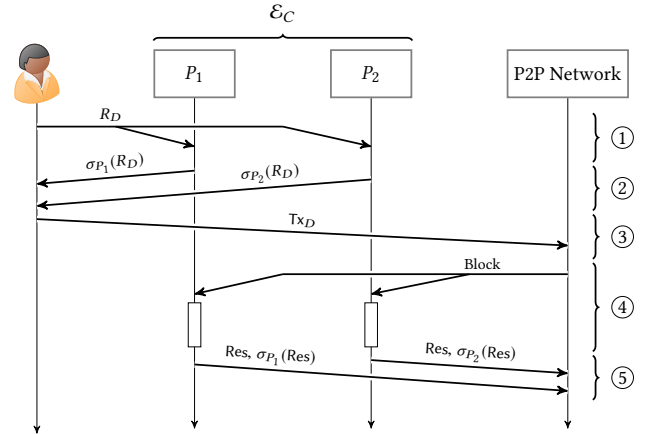


Figure 3: Contract deployment. The client first sends the deployment request R_D to the service providers chosen to be in the executing set \mathcal{E}_C ①, who then sign it and return it to the client ②. The client sends the resulting transaction to the network ③, where it gets included into a block. The service providers see this transaction in a block, execute the contract constructor ④ and send the signed results to the network to be included in a future block ⑤.

5.4 Contract Call

To call a smart contract, a client creates a transaction similar to Ethereum transactions. However, in addition to the standard components of an Ethereum transaction (nonce, gas, gas price, recipient, transfer value, input data, signature) the client also specifies the set of involved smart contracts which allows service providers to quickly determine whether they need to be involved in the execution of the smart contract. While the directly called contract is always specified for Ethereum transactions as well, the same does not apply to contracts that are indirectly called as subcalls from the executed contract.

To determine the set of involved contracts, the client can first execute the contract locally to get a set of dynamically called contracts. Such *reconnaissance step* is analogous to how gas costs are estimated in Ethereum. Similar to Ethereum, the contract state may change between local pre-execution that is based on the latest block and the final execution where new transactions may have been already applied. Such transaction execution may need to abort, and the client needs to send the transaction again with newly determined set of involved contracts (see Section 5.7). However, this is unlikely to happen for well-written smart contracts that are not purposefully designed to exhibit such behavior. Alternatively, static analysis can be used to over-approximate the involved smart contracts.

We note that clients cannot misuse this feature to violate contract safety. If the contract call has a subcall to some contract that is not included in the list, it will simply not be executed by the responsible executing set, i.e. no changes to the respective contract state can be committed. Other (non-compromised) executing sets involved in the transaction will simply abort it.

Once the client has sent the transaction to the network, the miners check its validity, i.e., they check whether the sending account

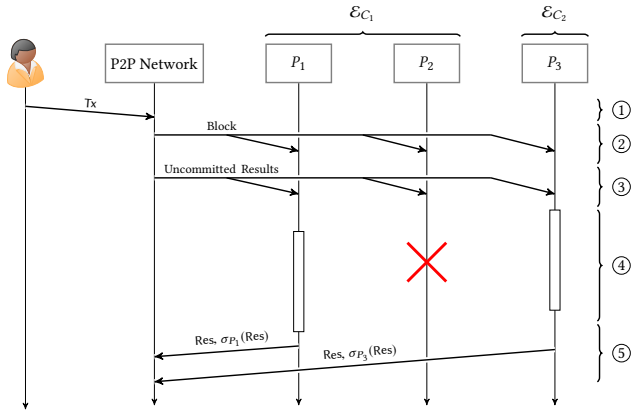


Figure 4: Cross-partition contract call. In this example, contract C_1 is executed with a 1-out-of-2 quorum by $\mathcal{E}_{C_1} = \{P_1, P_2\}$ and C_2 is executed with a 1-out-of-1 quorum by $\mathcal{E}_{C_2} = \{P_3\}$. The diagram shows the messages exchanged in a contract call involving C_1 and C_2 in which P_2 becomes unresponsive after the state exchange. Since P_1 is responsive, a quorum can still be reached and the results can be committed successfully.

has enough balance, the transaction nonce is correct, and the signature verifies correctly. They then include the transaction in the ordering section of the next block.

Service providers parse the blocks they receive and check whether they are involved in any contract call, i.e. whether some transaction has specified a contract, for which they are responsible. If this is the case, they add the transaction to their execution queue. Service providers can parallelize execution of distinct smart contracts, i.e. they only need to execute calls to the same contract sequentially and can thus use separate queues for each contract.

To execute a transaction that does not involve other smart contracts, the service providers simply execute the transaction, sign the state changes and send them to the network. The miners then check that the state transition is based on the most recent state and that the transaction has been signed by a quorum of t service providers in the executing set. If this is the case, they include the results in the result section of the next block, and thus commit the transaction.

5.5 Cross-Partition Contract Call

If the transaction does involve other smart contracts, the contract call is executed as described below. An example is shown in Figure 4.

- ① **Sending the Transaction.** The client creates transaction T_x and sends it to the network as described in Section 5.4. Miners check its validity and include it in the ordering section of the next block.
- ② **Fetching the Block.** The service providers fetch the new block including the transaction T_x from the P2P network and, if the transaction is addressed to (in the example, contract C_1 for P_1 and P_2) or lists (in the example, contract C_2 for P_3) a contract they are responsible for, they add the transaction to the execution queue of the respective contract.

- ③ **Tentative State Update.** The service providers continue to receive tentative results from other transactions from the P2P network that have not yet been included in the chain. Based on these results, the service providers update the tentative local state for other contracts. This allows them to use the current state of other service providers involved in the contract call during execution. In Figure 4 \mathcal{E}_{C_1} , i.e. service providers P_1 and P_2 , needs updates for the state of contract C_2 , and P_3 needs updates for the state of C_1 , if other transactions to one of these contracts is scheduled before T_x .

- ④ **Contract Execution.** Once the states have been updated and the transaction T_x is at the top of the execution queue for each contract, every service provider executes the full transaction, i.e., the contract call with all subcalls based on the current (tentative) state of the smart contracts. If a service provider does not yet have the contract code required for subcall execution, it can obtain this code from a previous block, as every contract deployment is recorded on the chain.

- ⑤ **Broadcast Result.** After the execution, each service provider signs the resulting state changes and sends the results (list of state changes) and their corresponding signature to the P2P network. The results also include a hash of the previous contract state to ensure that e.g. blockchain forks or inconsistent tentative states of other contracts cannot lead to inconsistencies. At least a quorum of t_i service providers in each involved executing set \mathcal{E}_{C_i} has to sign the state changes for the transaction to be accepted. In the example in Figure 4, P_2 becomes unresponsive and does not send its results. The protocol can still progress without abort, since P_1 is sufficient to achieve a quorum in this case ($t = 1, n = 2$).

Upon receiving state changes from service providers, miners check that previous state changes have already been received and applied, i.e., that for every involved smart contract, the state changes correspond to the next scheduled transaction in the list of transactions given by the ordering sections of the blocks in the chain. They further check the signatures on the state changes, ensuring that a quorum t_i has been reached for all involved smart contracts and then include the state changes in the result section of the next block.

5.6 Block Verification

To verify a block, verifiers need to check that it was formed correctly, the transactions listed in the ordering section are valid transactions, and that the results are valid. Further, they have to check that the block conforms to consensus rules, e.g. that it has a valid proof-of-work, and finally apply the state changes listed in the results section.

To check the validity of the ordering sections, the node checks for every transaction that the signature from the sender is valid and that the sender has sufficient balance for the call. For each result in the result section, the node checks that the hash of the previous state stored in the result is equal to the current state (before applying the state changes) of the contract, that the signatures from the service providers are valid and that a quorum from each executing set of involved contracts has been reached. Additionally, it checks that all results of previous transactions involving one of the involved smart contracts have been applied. That is, the previously defined per-contract transaction order is preserved. Nodes can keep FIFO queues for transactions per contract to easily keep track of which results are expected next for each smart contract.

To verify the registration section, the node checks that all the service provider enclaves registered to run a particular smart contract are from separate SGX processors. This is possible, since enclaves are attested using the linkable attestation mode.

5.7 Recovery from Errors

Unresponsive enclave. If u individual enclaves from one set become unresponsive, the transactions handled by that set of service providers can still complete, as long as $u \leq n - t$.

Unresponsive executing set. If $u > n - t$ service providers from the same set become unresponsive, we say that this set is unresponsive, and transactions involving that set of service providers can obviously not be processed, until some of the unresponsive service providers are fixed, e.g., restarted after a crash (in Section 8 we discuss how permanently unresponsive service providers can be replaced). The goal of ACE is to ensure that in such cases the unresponsive set cannot prevent *other* service providers and their contracts from proceeding.

The way this is achieved in ACE is that service provider enclaves can sign and broadcast an *abort* for a transaction. The enclaves will do this in the following two cases. First, they will broadcast an abort if they do not receive tentative state updates (with t signatures) for involved contracts in cross-partition transaction execution within a pre-defined time limit. Second, the enclaves can broadcast an abort if they do not receive enough signatures on the new state after execution within some time limit after they have finished its execution.

If the miners receive a signed abort from more than $n - t$ service providers out of an executing set \mathcal{E}_{C_i} of one of the involved contracts C_i , where $n = |\mathcal{E}_{C_i}|$ and t is the quorum threshold of C_i , they can include these aborts in the blockchain as result for the given transaction instead of state changes. This allows involved contracts to progress with the execution of future transactions, if the executing set of another contract is offline for an extended period of time.

Failed reconnaissance. If a client failed to correctly determine the set of contracts involved in a transaction, the service provider enclaves will notice this at the time of transaction execution and abort the execution, as explained above. The client can issue the same transaction again with newly estimated involved contracts.

Missing state change. Contract calls in ACE may involve dirty reads. If the state change from the previous transaction never reaches miners (e.g., due to a temporary networking issue or intentional attack), the following state changes cannot be committed, as the miners must follow the per-contract order for transactions. In such cases, the service providers will soon observe that their state changes are not committed by miners and they can simply re-send the state changes again. Since we assume that node eclipsing is not possible (see Section 4.1), eventually the missing state changes will reach the miners and the pending state changes can be committed.

6 SECURITY ANALYSIS

In this section we analyze the safety and liveness of ACE.

6.1 Safety

In ACE, smart contracts are executed inside TEEs that are SGX enclaves. The two main security properties of SGX are enclave code

integrity that is verified through attestation and enclave data confidentiality (see Section 2.2). For the purposes of this analysis, we say that a TEE is compromised if the adversary can violate either enclave code integrity or data confidentiality on that CPU.

We start our analysis by defining a basic security property of ACE for a restricted case where one set of service providers \mathcal{E} executes one contract.

LEMMA 1. Given the specification of a contract, which defines an executing set \mathcal{E} consisting of n enclaves and quorum t , the following holds: If fewer than t TEEs from \mathcal{E} are compromised, transaction atomicity, consistency and serializability hold for this contract in contract calls involving no other contracts.

PROOF. First, we need to show that if the adversary cannot compromise t or more *separate* TEEs (i.e., SGX processors), he cannot violate enclave code integrity or data confidentiality on t or more used enclaves. This follows from the fact that service provider enclaves are attested using the linkable mode of remote attestation. If multiple enclave instances of the same enclave running on the same SGX processor would be registered, this would be visible in the attestation evidence, and the miners would refuse to add such attestations to a registration section of a new block. An individual malicious miner can deviate from the protocol rules, but other miners will collectively ignore the invalid block, and therefore such service provider registrations cannot be part of the final and stable chain.

Since less than t of the specified enclaves can deviate from the attested enclave code and since less than t of the enclave signing keys may leak, the adversary cannot produce t or more signatures on state changes that are different from the correct enclave execution. Since all non-compromised enclaves follow the transaction order per contract, and execute their transactions sequentially one after another, such contract execution provides consistency and serializability for all transactions. Because transaction are either committed in full or not at all by miners, all transactions are also atomic. \square

Next, we extend our analysis and define a more general security property for the case where a call involving contract A executed by \mathcal{E}_A also involves other contracts.

THEOREM 2. Given the specification of contract A , which defines an executing set \mathcal{E}_A that consists of n_A enclaves and quorum t_A , the following holds: If less than t_A TEEs from \mathcal{E}_A are compromised, then transaction atomicity, consistency and serializability hold for contract A even in contract calls involving other contracts.

PROOF. We distinguish three possible cases:

1. No compromised executing sets. If for every other involved contract B with \mathcal{E}_B consisting of n_B enclaves and quorum t_B , less than t_B TEEs from \mathcal{E}_B are compromised the adversary cannot produce an acceptable and false state change for contract B and thus the adversary cannot produce a false tentative state of contract B that would be accepted by any of the non-compromised enclaves of \mathcal{E}_A . Because of this, t_A or more enclaves cannot execute the cross-partition contract call based on false initial state and the miners cannot accept such execution result. Thus, all executed transactions ensure consistency. Transaction serializability and atomicity are ensured due to the same reason as in Lemma 1.

2. Compromise during execution. If for all involved contracts, the tentative states are correct, but for an (or all other) involved

contract B with \mathcal{E}_B consisting of n_B enclaves and quorum t_B , at least t_B TEEs from \mathcal{E}_B are compromised *during the execution*, they can produce a false result for the execution of the contract with a quorum from \mathcal{E}_B . However, due to the same reasons as in Lemma 1, no quorum from \mathcal{E}_A can be achieved on these results, which ensures that they will not be accepted by the miners and thus consistency is ensured. Transaction serializability and atomicity are ensured due to the same reason as in Lemma 1.

3. Inconsistent tentative state. If for an (or all other) involved contract B with \mathcal{E}_B consisting of n_B enclaves and quorum t_B , at least t_B TEEs from \mathcal{E}_B are compromised *before the execution*, they can produce inconsistent results for the execution of previous calls to B , i.e. they can convince members of \mathcal{E}_A to use a false tentative state of contract B for the contract execution. Once the contract has finished executing, a quorum of \mathcal{E}_A may sign a set of state changes based on this false state. However, the signed results include a reference to the tentative state, and miners only include state changes in the result section of a block if this correctly references the previous state committed to the chain, i.e. the results will only be included if the tentative state used for the execution is in fact the canonical state committed to the chain, thus ensuring consistency for contract A . Transaction serializability and atomicity are ensured due to the same reason as in Lemma 1. \square

Finally, we note that the execution results (state changes) of smart contracts are only relevant to participants of the smart contract, as long as they do not enable double spends or affect the overall supply of the cryptocurrency. In ACE, the integrity of the underlying cryptocurrency is guaranteed independently of smart contract execution. In particular, a smart contract (or a compromised executing set) is not able to arbitrarily modify its balance. Transfers of money is always checked on the cryptocurrency consensus layer, i.e. a contract balance can only increase by amounts explicitly sent in transactions (or received from other smart contracts) and smart contracts can only initiate sending of funds that do not exceed their balance. This ensures that compromised executing sets can neither double spend funds nor create money out of nothing, even if an executing set is completely compromised.

6.2 Liveness

As shown above, ACE provides a strong safety guarantee, where the adversary has to compromise several SGX processors to violate transaction atomicity, consistency or serializability. This condition holds even if the adversary controls the OS of every registered service provider platform. The liveness guarantee of ACE is slightly weaker, since the above strong adversary could obviously prevent the non-compromised enclaves from running or communicating.

Given this observation, we define the liveness property for contract A that may or may not call other contracts:

THEOREM 3. Given the specification of contract A , which defines an executing set \mathcal{E}_A that consists of n_A enclaves and quorum t_A , the following holds: If the adversary is able to compromise at most c TEEs from \mathcal{E}_A and block enclaves on at most b platforms from \mathcal{E}_A , such that $t_A = n_A - c - b$, then contract A provides liveness.

PROOF. There are two possible cases to consider. First, we consider the case where contract A does not call any other contract B .

If the adversary compromises less than c enclaves and blocks enclaves on less than b platforms, then at least t_A non-compromised and non-blocked enclaves exist which enables the execution to complete for calls of contract A .

Second, we consider the case where contract A interacts with another contract B . If the above defined available enclaves of \mathcal{E}_A receive state changes from \mathcal{E}_B , they can complete the contract call. If the available enclaves of \mathcal{E}_A do not receive state changes from \mathcal{E}_B , they issue an abort which allows other transactions for the same contract to be completed. Similarly, if the available enclaves of \mathcal{E}_A receive inconsistent state changes from \mathcal{E}_B , the transaction will be aborted which allows other transactions for the same contract to be completed. \square

7 EVALUATION

In this section we analyze how complex contracts ACE can support and evaluate the performance of ACE.

7.1 Contract Complexity

We compare the average computational complexity of contract calls supported in the ACE execution model to the sequential execution model used in Ethereum. We modeled our system in Python to simulate its throughput for a large number of nodes. This allowed us to determine the complexity of contracts supported given a target throughput and different dependencies between the contracts.

Simulation details. To simulate ACE, we first set parameters such as the block size, the total number of service providers, the total number of smart contracts, and the mean and standard deviation for contract execution time. The block size puts an upper bound on the throughput and the number of service providers and contracts affects how interdependent the smart contract executions are.

We generate the set of contracts parameterized in their execution time and the number of involved contracts (number of sub-calls). For simplicity we assume that each contract call to the same contract results in the same amount of sub-calls and the execution time is fixed. The execution time is sampled from a normal distribution with given mean and standard deviation while the involved contracts are sampled uniformly from the list of all contracts. Furthermore, each contract is assigned a set of (between 1 and 6) responsible service providers, chosen uniformly from fix-sized list of providers.

For each block, we then sample the transactions for its ordering section. Each transaction calls a contract sampled uniformly at random from our set of generated contracts.

Measurements. For each configuration, we measured the throughput in transactions per second (Tps) for ACE for different mean execution times of the contracts. This allows us to determine how computationally intensive smart contract executions can be, given a target throughput (i.e. we want to exhaust throughput of an underlying consensus layer that can achieve k Tps). For our experiments, we fixed a block interval of 15 seconds and simulate up to 15000 transactions per block (1000 Tps). Note that this upperbound is not achievable in permissionless systems given current network throughputs, i.e. in practice the throughput will be capped at a lower number. Given the current average Ethereum block size, the throughput would be limited to 10 Tps. The total number of service providers in our experiments is set to 100.

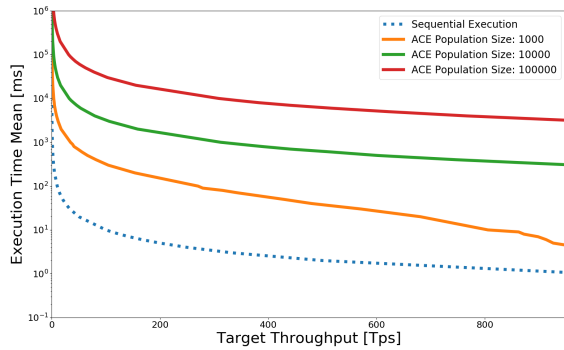


Figure 5: Average execution time per transaction given a target throughput. Sampling size refers to the number of different contract in the system from which we sample our transactions. The number of additionally involved contracts per contract call are sampled uniformly from the range [0, 5].

In order to compare to the theoretical maximum execution complexity in Ethereum and similar systems for a given target throughput, we also show the execution time supported by a sequential transaction execution model. This gives an absolute upper bound, since it assumes an instant propagation of the block and transactions and no other overheads, whereas in reality the average execution time has to be much lower to achieve a given target throughput.

Results. Figure 5 shows the results for different numbers of contracts from which we sample transactions. The maximum number of involved contracts is fixed at 5 which is currently the 99th percentile in Ethereum’s mainnet. The impact of the sampling size is clearly visible from our simulation results. When sampling from a small set of 1000 different contracts, ACE allows supporting execution times of one to two orders of magnitude higher than sequential execution, because the dependency between transactions is high. For a set size of 100’000 contracts, much higher execution times can be supported for a given target throughput. For example, for a target throughput of 10 Tps (currently achievable by Ethereum), transactions can have a mean execution time of 5 minutes, whereas the theoretical upper bound for sequential execution is at 0.1 seconds.

In Figure 6, we show the results for varying numbers of other contracts involved in a contract call. For these measurements, contracts are sampled from a set of 10’000 contracts. The numbers listed in the graph denote the upper bounds of the range from which we uniformly sample the number of additionally involved contracts for a contract call. We see that going from transactions involving zero additional contracts to transactions involving up to 5 additional contracts adds a lot of dependency and therefore supports a much lower average computational complexity. However, this difference gets much smaller with more and more additionally involved contracts.

Our results show that ACE is capable of handling contracts with execution times in the order of seconds or minutes. Clearly, ACE can support much longer execution times when contracts have a low interdependence, e.g. if many distinct (e.g. 10’000) smart contracts exist in the system or contracts only involve a low number of other smart contracts. Also in our simulations we used 100 service providers. If more service providers are available, ACE can support even longer execution times.

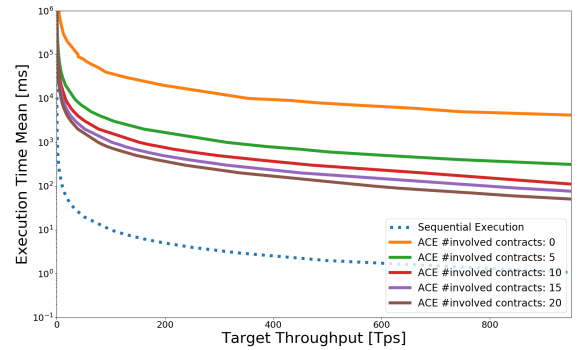


Figure 6: Average execution time per transaction given a target throughput. The sampling size is fixed to 10’000 different contracts while the upper limit of the range from which we sample the number of additionally involved contracts varies from 0 to 20.

Summary. Overall, we observe that ACE can process transactions with execution times that are 2-4 orders of magnitudes larger than what can be theoretically handled by sequential execution in the optimal case (instant block propagation). Since sequential execution is much slower due to block propagation delays, the difference would be even larger in practice.

7.2 ACE Performance

Next, we evaluate the computation, storage and communication cost of ACE.

Implementation. We implemented ACE based on Ethereum. Our implementation supports the execution of smart contracts that are compiled to EVM bytecode and it is based on the EVM implementation from the Parity Ethereum client for which we migrated the relevant parts to SGX, resulting in 30000 lines of enclave code.

Computation cost. The first performance metric that we evaluate is computation, measured as the time required for verification of contract call results (state changes). This verification task needs to be performed by all miners (and all other nodes that verify blocks) and if verification takes too long, then the savings of off-chain contract execution may be reduced or eliminated.

Verifying the transaction result consists of the verification of the service provider signatures, deserialization of the encoded results and finally application of the state changes. As the time used for hashing (needed for signature verification) and deserialization both depend on the size of their input we measured the verification time relative to the size of a transaction result in terms of number of storage changes, i.e. the number of variables changed during the contract call, which is independent of the number of sub-calls or the quorum size. One variable corresponds to 64 Bytes (key and value).

We conducted our experiments on a machine with a 4 Core Intel(R) Core(TM) i7-7700 CPU @ 3.60 GHz, 2 threads, CPU family 6 and 16 GB RAM. As expected, the verification time is proportional to the result size. The throughput for verifying and applying state changes is roughly 160 MB/s. Each service provider signature contributes with 0.1 ms to the total verification time but this could be parallelized for multiple signatures.

Tx/Block	Changes/Tx	Quorum Size	Storage	Verification
60	4	5	35kB	0.03s
60	10	5	58kB	0.03s
60	25	10	134kB	0.06s
60	50	10	230kB	0.06s
60	100	10	422kB	0.06s
60	200	20	845kB	0.12s

Table 1: On-chain storage cost and total verification time for the result section of blocks with 60 contract calling transactions per block (current Ethereum throughput), given different average quorum sizes and number of state changes per transaction.

Table 1 shows some examples for block verification time. We can see that even for large quorum sizes of 20 service providers and with an average number of 200 state changes per transaction, total verification cost is small at 0.12 seconds per block.

Storage cost. The second performance metric that we evaluate is storage, measured as the on-chain storage due to execution results. Execution result storage scales linearly with the number of state changes. For each updated 32 byte word, 64 bytes are needed (32 bytes for the location, 32 for the new value). In addition we need to store an ECDSA signature of 64 bytes for each member of the quorum. Note that the storage cost is independent of the execution time, i.e. even long-running contracts may only make very few state changes.

Table 1 shows examples for on-chain storage costs given different average number of storage changes per transaction and different quorum sizes. We assume the current throughput of Ethereum in terms of contract calling transactions per block (60). Depending on the number of state changes and size of quorum, the storage overhead ranges from tens to hundreds of kilobytes. Currently, Ethereum has an average of 4 state changes per block, i.e. for similar contracts, the result section would add an overhead of 35kB to each block, effectively doubling the block size. With a larger number of state changes, the overhead grows linearly. Recent proposals for block dissipation claim that 1MB blocks could be propagated to the majority of the Bitcoin network within 1.9 seconds given connection speeds of 56Mbps [26], thus making it feasible to support even contracts with a large number of state changes (e.g. 200) in ACE.

Communication cost. The primary communication cost of ACE is that it may increase block size as described above. All other messages involved in the system have no significant cost.

8 DISCUSSION

Changing service providers. For a long running smart contract, it may be desirable to have the ability to update the set of service providers. For example, one may want to replace a service provider that has become (permanently) unresponsive. A simple solution is to allow the contract creator alone to change the executing set. The main drawback of this approach is that a contract creator could initially appoint a set of service providers that is acceptable to the contract participants who would then load money into the contract. After that, the contract creator could then update the set of service providers such that it no longer meets the contract participants’

requirements. A better solution is that the initially deployed smart contract code includes a contract-specific *service provider update policy* that defines the conditions under which the service providers may be updated after the initial contract deployment.

Aggregate signatures. For large executing sets (e.g. 20 or more service providers), the required result signatures increase the the block size and verification time, as discussed above. If large executing sets are used, the block size and the verification time can be reduced by using an *aggregate signature* scheme such as BLS signatures [6]. In an ACE contract call (see Section 5.4), the client could collect individual result signatures from t out of n service providers and then aggregate them together before broadcasting the results to the miners. Such optimization is beneficial only for large execution sets, because the verification time of aggregate signatures is high compared to standard signatures like ECDSA.

Contracts with private state. Since ACE executes contract code inside TEEs (SGX enclaves), it could be adapted to enable confidential contracts. Here we outline two possible models.

First, if the creator of a smart contract wants to keep the state private, except for a predefined set of participants, he generates a new symmetric key sk that he shares with the participants. He then encrypts sk with the public keys of the service provider enclaves that he wants the contract to be deployed at. This encrypted key is stored on the chain together during contract deployment. To call the contract, participants encrypt the contract call inputs using sk .

Second, if the state of a contract should be completely private, i.e. not even known to the creator or contract participants, the service provider enclave listed first in the deployment request can generate a symmetric key and exchange it with the other service provider enclaves using their public keys. Contract participants can encrypt inputs to contract calls using the service provider enclaves’ public keys. In either of the two scenarios, the result of a contract call will not contain the state-diff of the contract account (that has to be private) but rather an encrypted hash of the updated account state.

9 FURTHER RELATED WORK

In Section 3.2 we already discussed closely related solutions and their limitations. In this section we review further related work.

Brandenburger et al. [7] have developed an extension to Hyperledger Fabric [4] that uses Intel SGX [22] to provide private state for smart contracts. Similarly, ShadowETH [39] uses enclaves to execute smart contracts privately for Ethereum. In contrast, ACE tackles the problem of contract complexity in blockchains such as Ethereum, by leveraging trusted computing to enable the execution of long-running smart contracts.

Plasma [31] is another approach for scaling on top of Ethereum by introducing so-called *child-chains*, which run their own consensus mechanism and process transactions for a specific purpose. This requires explicit transfer of assets between the main chain and a child-chain and complicates interaction between child-chains.

Other works focusing on concurrency of smart contracts exist [15, 40]. However, they only parallelize execution on a single node in order to use multiple processor cores for faster block validation and thus still suffer from the limitation that every node needs to execute every smart contract call.

10 CONCLUSION

Permissionless smart contract platforms like Ethereum are often expected to enable arbitrary financial applications. However, in practice such systems impose heavy restrictions on the types of computations that can be implemented. In this paper, we have described a novel system called ACE that enables several orders of magnitude more complex contracts using off-chain execution by appointed service providers. The key technical ingredient of our solution is a concurrency control protocol that allows contracts to call each other across service provider boundaries but does not require that all service providers must mutually trust each other.

REFERENCES

- [1] 2016. Solidity. (2016). <https://solidity.readthedocs.io>.
- [2] 2017. Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation. (2017). <https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf>.
- [3] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2018. Chainspace: A sharded smart contracts platform. *Symposium on Network and Distributed Systems Security (NDSS)* (2018).
- [4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30.
- [5] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. Concurrency control and recovery in database systems. (1987).
- [6] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 514–532.
- [7] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. 2018. Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric. *arXiv preprint arXiv:1805.08541* (2018).
- [8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiaainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*. USENIX.
- [9] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*. USENIX Association, Baltimore, MD. <https://www.usenix.org/conference/usenixsecurity18/presentation/bulck>
- [10] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *CoRR abs/1802.09085* (2018). [arXiv:1802.09085](http://arxiv.org/abs/1802.09085)
- [11] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2018. Eکیدen: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *arXiv preprint arXiv:1804.05141* (2018).
- [12] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.
- [13] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. In *Cryptology ePrint Archive, Report 2016/086*.
- [14] Sourav Das, Vinay Joseph Ribeiro, and Abhijeet Anand. 2018. YODA: Enabling computationally intensive contracts on blockchains with Byzantine and Selfish nodes. *arXiv preprint arXiv:1811.03265* (2018).
- [15] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 303–312.
- [16] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (1976), 624–633.
- [17] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Adi Shamir, and Eran Tromer. 2016. Physical key extraction attacks on PCs. *Commun. ACM* 59, 6 (2016).
- [18] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. 2016. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 3–16.
- [19] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *European Workshop on Systems Security*. ACM.
- [20] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)* 15, 4 (1983), 287–317.
- [21] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. 2015. Eclipse Attacks on Bitcoin’s Peer-to-Peer Network. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*.
- [22] Intel. [n. d.]. Intel Software Guard Extensions. ([n. d.]). <https://software.intel.com/en-us/sgx>.
- [23] Intel. 2017. Intel Software Guard Extensions - Developer Zone - Details. (2017). <https://software.intel.com/en-us/sgx/details>.
- [24] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. 2016. Intel SGX: EPID Provisioning and Attestation Services. (2016). <https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation>.
- [25] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. 2018. Arbitrum: Scalable, private smart contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 1353–1370.
- [26] Uri Klarman, Soumya Basu, Aleksandar Kuzmanovic, and Emin Gün Sirer. [n. d.]. bloXroute: A Scalable Trustless Blockchain Distribution Network WHITEPAPER. ([n. d.]).
- [27] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.
- [28] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. 2015. Demystifying incentives in the consensus computer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM.
- [29] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. Cachezoom: How SGX amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems*. Springer.
- [30] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).
- [31] Joseph Poon and Vitalik Buterin. 2017. Plasma: Scalable autonomous smart contracts. *White paper* (2017).
- [32] Kun Ren, Alexander Thomson, and Daniel J Abadi. 2014. An evaluation of the advantages and disadvantages of deterministic database systems. *Proceedings of the VLDB Endowment* 7, 10 (2014), 821–832.
- [33] Hubert Ritzdorf, Karl Wüst, Arthur Gervais, Guillaume Felley, et al. 2018. TLS-N: Non-repudiation over TLS Enabling Ubiquitous Content Signing. In *Network and Distributed System Security Symposium (NDSS)*.
- [34] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. 2017. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*. Lee, 444–460.
- [35] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 1–12.
- [36] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. (2014).
- [37] Gavin Wood. 2018. Ethereum: A secure decentralised generalised transaction ledger BYZANTIUM VERSION (e94ebda - 2018-06-05). <https://ethereum.github.io/yellowpaper/paper.pdf>. (2018). Accessed: 2018-08-30.
- [38] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *SP*. IEEE.
- [39] Rui Yuan, Yubin Xia, Haibo Chen, Binyu Zang, and Jan Xie. 2018. ShadowEth: Private Smart Contract on Public Blockchain. *Journal of Computer Science and Technology* 33 (2018), 542–556.
- [40] An Zhang and Kunlong Zhang. 2018. Enabling Concurrency on Smart Contracts Using Multiversion Ordering. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*. Springer, 425–439.