CCF: A Framework for Building Confidential Verifiable Replicated Services

Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak*, Olga Ohrimenko, Felix Schuster*, Roy Schuster, Alex Shamis, Olga Vrousgou, Christoph M. Wintersteiger

Microsoft Research & Microsoft Azure

April 2019

***Abstract*—We present CCF, a framework to build permissioned confidential blockchains. CCF provides a simple programming model of a highly-available data store and a universally-verifiable log that implements a ledger abstraction. CCF leverages trust in a consortium of governing members and in a network of replicated hardware-protected execution environments to achieve high throughput, low latency, strong integrity and strong confidentiality for application data and code executing on the ledger. CCF embeds consensus protocols with Byzantine and crash fault-tolerant configurations. All configurations support strong service integrity based on the ledger contents. Even if some replicas are corrupt or their keys are compromised, they can be blamed based on their signed evidence of malicious activity recorded in the ledger. CCF supports transparent, programmable governance where the power of the consortium members is tunable and their activity is similarly recorded in the ledger for full auditability.**

We are developing an open-source implementation of CCF based on SGX-enabled Azure Confidential Compute, built on top of the Open Enclave SDK. Experimental results show that this implementation achieves throughput/latency tradeoffs up to 3 orders of magnitude better than previous confidential blockchain designs. Its code and documentation are available at `https://github.com/Microsoft/CCF`.

I. Introduction

Current blockchain designs do not meet the confidentiality and performance requirements of many applications [17], [70]. For example, Bitcoin [52] requires about an hour [11] to yield high confidence that a transaction has committed, and the public Ethereum network [67] processes only 10 transactions per second. Meanwhile, Visa averages 2,000 transactions per second [1]. In addition, their smart contracts and their full transaction history are in the clear for anyone to see. Several recent systems [2], [25], [28], [39], [40], [48] propose more efficient permissionless blockchain designs based on variants of Byzantine agreement [16], [44], [50], [57], but they do not offer confidentiality guarantees. While some designs leverage zero-knowledge proofs to hide transaction contents [59], they involve expensive proof generation procedures (e.g., over one minute on a consumer machine [17]) and often rely on an undesirable trusted set-up to bootstrap the system.

Permissioned blockchains gain efficiency by leveraging a stable consortium of members to manage their governance. They enable much larger sets of users to submit transactions; for example, a consortium of banks may implement a service for millions of customers. Some of their designs do not address confidentiality [5], [23] while others provide confidentiality but relatively low performance (e.g., about 4 transactions per second [17]).

In this paper, we present CCF, a framework that addresses these limitations: it provides both confidentiality and high-performance for consortium-based blockchains. CCF replicates its blockchain operations using a network of hardware-protected trusted execution environments (TEEs). This yields high throughput, high availability, and low latency, while at the same time protecting the integrity and confidentiality of application data and code running on the ledger. Although CCF may leverage any TEE, our initial implementation uses Intel SGX (Software Guard Extensions) enclaves [49]. Enclaves are hardware-protected memory regions that allow trustworthy execution of code even on untrusted host computers. Some recent blockchains designs also use enclaves [47], [71], [72], [78] but they do not provide confidentiality guarantees.

In CCF, the members of a consortium may not necessarily trust one another, and need only agree on the service they intend to run, including its application code, its governance, and its initial configuration. The configuration comprises the member credentials and a set of TEE-enabled hosts to replicate the blockchain. The hosts may be in one or more cloud data centers, or in the members' enterprise data centers. Recall that all hosts support TEEs, irrespective of their locations. The members' agreement is consolidated in the hashed digest of the service code to run within each TEE, which includes implementations of the CCF protocols, the application logic, and their supporting runtimes and libraries. Following previous design patterns [61], we strive to keep TEE code small and application-specific in order to minimize our software TCB. Additionally, CCF could harden code inside the enclaves [12], [27], [31], [53], [62], [63] against exploits of side-channels or memory safety defects.

CCF provides a simple programming model of a key-value store and a universally verifiable log that implements a ledger abstraction. CCF developers can write application logic (also known as smart contracts) in several languages by configuring CCF to embed one of several language runtimes on top of the key-value store. Developers can also express flexible policies as smart contracts that enforce access control. For example, they can express permissive policies that authorize a specific participant to perform audit requests over the full transaction

* Work done while at Microsoft Research

graph, and restrictive policies that allow only the parties involved in a transaction to access its confidential details.

CCF supports consensus protocols with both Byzantine and crash fault-tolerant configurations, while ensuring that all configurations support strong service integrity (provided the application code records adequate information in the ledger) even if some nodes or their keys are compromised, and specifically ensuring that compromised nodes can be identified through evidence of their malicious activity recorded in the ledger.

To allow evolving the service and the rules of its consortium over time [29], CCF supports a model of programmable governance running on top of the ledger abstraction. Hence, the dynamic service configuration—including its members, replicas, users, application code, and governance rules—is represented as key-value pairs in the data store, and all governance operations are recorded on the ledger. In particular, CCF enables governance-based rule updates, software upgrades, and service recovery in the face of a disaster that may cause all replicas to fail.

As a precaution against various compromise scenarios, CCF carefully restricts the privileges of the TEEs trusted to run its service and the members trusted to run its governance. To this end, we consider a variety of attacker models and, anticipating on the next sections, we address them as follows:

- High performance and availability depend on the otherwise untrusted TEE hosts, their storage, and their communications network. This risk may be mitigated by replicating the service between widely distributed TEEs. Pragmatically, CCF also provides an optional mechanism to recover from major failures of the replication protocol itself, subject to governance.
- Confidentiality depends on TEE hardware protection, combined with application code review and remote attestation to ensure that no other code may gain access to confidential transactions within TEEs. The risk of hardware compromise is somewhat reduced by using key rotation, separation, and erasure. In contrast, transactions are encrypted in the ledger and remain confidential even if the hosts, the network, and most of the members are compromised.
- Governance relies on the service members reaching consensus before modifying the configuration. For transparency and auditability, the governance process is recorded in plaintext in the ledger.
- For long-term ledger integrity, we consider a stronger attacker model where all TEEs may eventually be compromised. To resist such powerful attacks, CCF maintains a Merkle tree [58] over the whole contents of the ledger and systematically records signed evidence in the ledger. This enables the replay of the service protocols and the detection of malicious activity by its members and replicas. In addition to this *trust, but check* approach, which is always enabled, CCF's replication protocol can be configured to *prevent* active attacks from less than a third of the TEEs, using Byzantine fault-tolerance techniques.

We implemented several blockchain applications with CCF. In some of the applications, we aimed for functional compatibility with Ethereum and ran an implementation of the Ethereum Virtual Machine (EVM) that stores state in CCF's data store. Experimental results from running these applications on SGX-enabled VMs in Microsoft Azure show that they achieve two orders of magnitude more throughput than the current public Ethereum network, while providing confidentiality. For another application, we did not set any compatibility requirements and aimed for simplicity and high-performance. Experimental results show that it achieves throughput that is orders of magnitude better than previous confidential blockchain designs.

Contents. Section II provides background on TEEs and replication protocols, and also sets up notations. Section III gives an overview of the protected service, describing its store, its programming model, and its governance. Section IV presents our TEE service protocols. Section V details the ledger format for integrity and confidentiality. Section VI presents our replication protocols. Section VII discusses our implementation. Section VIII evaluates it using sample applications. Section IX discusses related work on blockchains. Section X concludes.

II. Background

A. Trusted Execution Environments (TEEs)

TEEs provide a means for the attestable, privacy-preserving, and integrity-protected execution of sensitive programs within otherwise untrustworthy systems. A range of TEE architectures has been proposed [22], [26], [56] and commercially deployed [6], [36], with Intel SGX [21], [49] being one of the most comprehensive and prevalent commercial implementations. Hence, we focus on SGX as TEE provider.

SGX is a recent addition to the x86-64 instruction set architecture. It enables the dynamic creation of enclaves within the virtual address space of user mode processes. An enclave can be initialized with arbitrary code and data. The initial state (i.e., code and data) and configuration of an enclave is recorded by the CPU into a cryptographic digest called a *measurement*. Once an enclave is initialized, the CPU prevents all other software (including the hypervisor, operating system, and host process) from accessing the enclave's memory range. However, enclave code can still access its host's entire address space, enabling efficient communication with the outside world. To protect against hardware attackers, the CPU transparently encrypts and integrity-protects all enclave memory before writing it to RAM.

An enclave exposes one or more fixed entry points that can be invoked by the untrusted host. While executing enclave code, system calls are not available. Thus, the enclave returns control to the host for performing any network and file I/O.

Two essential features of TEEs are sealing and remote attestation [4]. For these, each SGX-enabled CPU is provisioned with root hardware secrets, used in particular to derive a signing key for its unique platform identity (written q below) and separate keys based on code identity (written C below).

Sealing allows an enclave to protect private data using authenticated encryption under a key derived from a hardware secret and its code identity. The resulting ciphertext may then be passed to the untrusted host and written to disk. It can be unsealed only by an enclave running the same code on the same platform (possibly after a power-down cycle). We write $\mathbf{Seal}_q[\mathcal{C}](m)$ for sealing data m on platform q for code identity $\mathcal{C}$ and $\mathbf{Unseal}_q[\mathcal{C}](v)$ for the corresponding unsealing of ciphertext v. The latter operation may return a decryption error. Remote attestation allows an enclave to convince a party that it (i) runs on an SGX-enabled platform q, (ii) has code identity $\mathcal{C}$, and (iii) produced message m. For this, abstractly, platform q issues a signed certificate $\mathbf{QSig}_q[\mathcal{C}](m)$, which binds q, $\mathcal{C}$, and m and is also referred to as *quote*. Quotes are used in particular to establish secure communication channels with mutual authentication between enclaves.

(As explained in Section IV, CCF security crucially depends on the remote attestation of the nodes running the service, but it depends on sealing only for reporting node failures.)

Weaknesses. A single vulnerability, e.g., a memory corruption, in enclave code can void all privacy. As mitigation, variants of software fault isolation have been proposed [61], [63]. Further, recent research [13], [15], [45], [68], [69] finds the SGX architecture susceptible to different side-channel attacks and proposes mitigations [12], [27], [31], [53], [62]. Such vulnerabilities in enclave code and side-channel attacks are out of scope in this report.

B. Crash-Fault and Byzantine-Fault Tolerant Replication

CCF implements its data store across multiple computing nodes using a replication protocol [42], which may be configured to withstand a range of failures and attacks:

- Crash fault-tolerant replication (CFT) protocols assume that each node runs the protocol correctly, and ensure safety when nodes are unresponsive or the network is lossy. They also ensure liveness whenever a (strict) majority of nodes communicate with one another, under a weak synchrony assumption [16].
- Byzantine-fault-tolerant (BFT) protocols provide additional protection against misbehaving replicas [16]. They ensure safety and liveness in a stronger attacker model whereby (less than a third of) the nodes are compromised. These guarantees come at the cost of additional replicas and additional checks during protocol execution.

These protocols may support dynamic re-configuration, using subprotocols for adding nodes and removing nodes [43].

CCF uses a form of state machine replication [60] that can be implemented using any replication protocol that provides the API specified below. Section V explains how we strengthen the protocol to enable signature-based universal verifiability and auditing. Section VII describes the current implementation of the replication protocol.

Replication API. We define a local interface within TEEs between the replicas and the rest of the code running service protocols. (In contrast, more abstract presentations usually provide a single abstract interface that hides the transient local state of the replicas.)

From this interface, the visible state of each replica consists of a log of transactions μ and a commit index c within that log. The *stable part* of the log is its prefix up to the commit index. In state (μ, c) with $c \leq |\mu|$,

- any node can *propose* an extension, written $\mu \rightsquigarrow \mu; \rho$.
- any replica can update its local state, in three ways: *extend* the log, by applying any prior proposal that extends μ; *truncate* an uncommitted suffix of the log, as long as $c \leq |\mu|$; or *advance* the commit index, as long as all stable logs are consistent (i.e, the logs of all replicas with commit index $c > n$ agree on their first n transactions.).

This specification captures abstract integrity and consistency properties, while hiding all messages and internal protocol details. For example, a correct replication protocol will ensure that proposed extensions are sufficiently replicated before advancing the commit index, so that they can reliably be restored after a crash.

A further simplification would be to hide any uncommitted transactions, and provide read access only to the stable part of the log. However, this may be somewhat misleading for confidentiality (as clients may observe the effect of transactions before they are committed) and inefficient (as nodes would have to wait for commitment between transactions).

The replication protocol starts with a single node with local state $(\emptyset, 0)$. Its interface supports *reconfiguration*: one can add nodes (e.g., with an empty log, or by cloning the state of another replica) or remove nodes (keeping the stable part of their state for consistency). It also enables some form of verifiability. For example, every node may embed their signature in each of their proposals, so that anyone may later check the log and (potentially) blame bad transactions on their proposers. The logged transactions may include additional evidence, such as signed messages of the replication protocol, so that the ledger may also enable verification of commitment, consistency, and protocol reconfiguration. Section VI presents our protocol and its verifiability properties in more detail.

III. A Trusted Service and Key-Value Store

CCF runs a service on top of a replicated key-value store, aiming to provide the same security guarantees as an ideal trusted third-party server. We present below the service functionality and programming model at the application level. The supporting protocols are described next in Sections IV–VI.

The service is defined by application code that programs a set of *commands* to operate on the key-value store. It accepts connections from clients and, once authenticated, executes commands on their behalf. The service implements the commands as atomic transactions over the key-value store; it ensures their serializability and it persists their effects in the store. One may reason about the security of the service by reviewing its application code. For instance, one may verify that the commands correctly enforce access control based on the client identity before reading or updating the store, and that they all preserve the application invariants.

In the following, we use two application examples: a basic Bitcoin-like service for transferring money between accounts and a more advanced Ethereum-like service that runs stateful EVM smart contracts.

A. The Clients

We consider clients in two roles. *Users* can run the application commands of the service. For example, for the simple Bitcoin-like payment system, they can read the balance of their account, and they can transfer money from their account provided it remains in credit. Users may be compromised, and may thus leak or modify application data, subject to the access granted by the application.

Members can run privileged commands to manage the service using its governance module. They are jointly responsible for endorsing the service and managing its configuration (e.g., its users and members) through votes. Members and users are usually distinct actors. The same governance module is included in every CCF service, with some governance configuration. It is independent from the user application. Members may also be compromised, and may collude with users. However, some security guarantees of the service will depend on a quorum of its members being honest.

B. The Store

The store is a collection of tables. Each table has a unique name and records key-value pairs with fixed datatypes. We write $k \to v$ for a pair with key k and value v.

The store is divided into *application tables* and *governance tables*, the latter including primitive tables that record the current configuration of the underlying service protocols. For instance, a NODE table keeps track of trusted TEEs.

Application tables. Each application defines its own collection of tables. For example, our simple Bitcoin-like application uses an ACCOUNT table to keep track of user-balance pairs of the form $u \to x$. In an Ethereum-like application, a more complex table maps account numbers to smart-contract code and state.

Governance tables. These tables are the same for all services. They are explained throughout the paper, with a summary of all tables given in Table A.1. We begin with tables that define the current clients of the service:

USER stores $u \to \mathsf{cert}_u, \mathsf{rights}_u$ for each active user, where u is a unique user identifier, cert_u is her identity certificate, and rights_u represents her access rights, such as the application commands she may call.

MEMBER stores $m \to \mathsf{cert}_m, \mathsf{ksk}_m, \mathsf{status}_m$ for every service member, where m is a unique member identifier, cert_m is her identity certificate, ksk_m is her keyshare encryption key (see Section IV-G), and status_m is her current status. Entries in this table are never deleted to ensures member identifiers remain consistent for the lifetime of the service. Instead, their status may be updated from accepted (proposed by their peers) to active (authorized to run all governance commands) to retired.

Confidentiality. Depending on the application, the service definition also sets the confidentiality of each table.

Tables are either *public* or *private*. Public tables can be read by any client; the service protects the integrity of their contents. Private tables are annotated with a confidentiality label; the service keeps their contents encrypted using separate keys, and provides access only via its programmed commands. Typically, most application data is private, whereas most governance data is kept public.

C. The Ledger

To enable replayability of the service, CCF persists the history of the store in a tamper-proof ledger. Consider for example a frequently-updated table. The application only gives access to the current values of each key (kept in protected memory by the TEEs) whereas the ledger also keeps tracks of any overwritten values. This lower-level view of the store matters mostly for security and auditing.

Each command produces a sequence of key-value updates in various tables; as its transaction completes, this sequence is appended to the ledger. We let τ range over transactions (a sequence of key-value updates that records the effect of a command) and μ, ρ range over sequences of transactions. Updates to private tables are encrypted in the ledger, whereas updates to public tables are only integrity protected.

We will use μ to represent the whole ledger contents, and ρ for ledger updates passed to the replication protocol interface as described in Section VI. Hence, the state of the store $\mathcal{S}$ is a function of μ. The ledger format and its security properties are described in Section V.

D. Public-Key Infrastructure

CCF provides native support for public-key certificates and signatures. To this end, the service embeds a certificate store that records certificate-chain validation policies, authorized roots and intermediate certificates, and recent certificate-revocation lists in the governance tables CERTCONFIG, CERT and CRL, respectively. CCF validates all certificates and signatures against their contents in the store. This ensures their processing can be replicated and replayed.

Although some applications may choose to record simply self-signed certificates, or even just public keys, it is convenient to rely on existing PKI mechanisms for certificate issuance, management, and revocation. This approach also enables simple integration with the credentials used by TLS for mutual authentication between CCF clients and nodes.

In the following, we write $\{u : \mathsf{request}\}$, $\{m : \mathsf{request}\}$, and $\{n : \mathsf{msg}\}$ for signature tags issued by users, members, and nodes using the key identified by the certificate currently stored in the USER, MEMBER, and NODE governance tables. All requests signed by clients implicitly include a service identifier, to prevent any ambiguity between different services.

E. Remote Procedure Calls

We now describe commands in more detail. Clients communicate with the service via several RPC interfaces, always relying on TLS 1.2 to establish a secure channel with mutual authentication and forward secrecy. The client's certificate chain is checked against the contents of the USER table (or the MEMBER table, for governance commands) and the current state of the certificate store, whereas the server presents a TEE-quote certificate endorsed by the service key (see Section IV).

```
App.Transfer(caller: target, amount) {
  // the service authenticates the caller
  assert(exists(ACCOUNT[target]))
  assert(amount <= ACCOUNT[caller]);

  ACCOUNT[caller] -= amount
  ACCOUNT[target] += amount
}
App.GetBalance(caller) { return ACCOUNT[caller] }
```

Listing 1. Pseudocode for Bitcoin-like application RPCs.

```
App.SignedTransfer(caller: target, amount, sig) {
  // the service verifies the caller 's signature
  assert(exists(ACCOUNT[target]))
  assert(amount <= ACCOUNT[caller]);

  TRANSFER[] = sig
  ACCOUNT[caller] -= amount
  ACCOUNT[target] += amount
}
```

Listing 2. A variant illustrating signed persisted requests.

The service API is largely application-specific. It can be statically programmed in the TEE code (using e.g. C++), but it can also rely on scripts supported in that code (using e.g. Lua or EVM). Static code may be easier to review, whereas scripts offer more flexibility: they can be stored by the service and updated as part of its governance, or even passed as command parameters. In this presentation, we simply rely on pseudocode for all programming examples.

A first command example. A possible (partial) implementation of our Bitcoin-like example is given in Listing 1. The store uses an ACCOUNT table with user identifiers as keys and positive amounts as values. The service implicitly authenticates the caller as one of its currently-registered users. By convention, the authenticated client identifier is passed as the first argument of every command.

The command for transfers takes a target account and an amount as additional arguments; it checks that the source account has enough credit, then it updates both accounts. Its transaction semantics ensures that these checks and updates are atomic. The second, read-only command just returns the current balance of an account.

Signed Requests. In the transfer command of Listing 1, the ledger records two updated key-value pairs. By replaying it, one can verify its correctness (checking that all transactions are indeed valid transfers) but not user authentication.

To this end, the service may additionally require that the command requests be signed—by convention, the signature tag `sig` is then passed as the last argument of the command. The service then also implicitly verifies this signature over the other arguments of the request using the stored credentials of the caller before executing the command.

Continuing with our example in Listing 2, the received signature `sig` is verified as `{caller : target, amount}` then recorded in an auxiliary table TRANSFER. This table does not have a key, hence its value is overwritten by every transfer, but the resulting transaction in the ledger now records both the account updates and the supporting signature.

Signed requests involve additional cryptographic processing for the service and its clients (which may otherwise rely on long-lived connections for issuing many commands). Depending on the application workload, they may be required only for selected commands. Note also that the command code is responsible for storing the signature value together with sufficient context (so that it can be re-verified) within tables at an adequate level of confidentiality, subject to client privacy considerations.

Early Results and Confirmations. By design, the service immediately returns the result of each RPC as the command completes, with the important caveat that this result may still be rolled back if the TEE that executed the command crashes. (The result implicitly includes the tentative serialization index of the command in the ledger.) If the command updated the store, the server runs in parallel the underlying replication protocol to propagate, persist, and eventually commit the resulting transaction (see Section VI).

Clients can later query the commit state of their transactions using a separate `GetConfirmation` RPC. Optionally, this RPC also returns a signed receipt for committed transactions, providing independent evidence that the command was indeed executed by the service at a given index in the ledger and produced the previously-returned results (see Section V).

Handling RPC failures. RPCs may fail to return a result for a variety of reasons, such as client crashes, node crashes, connection failures, or timeouts. In such cases, the client may still need to determine whether its command has been executed or not before issuing its next command. To this end, the application command should record unambiguous information in the transaction, such as a sequence number or unique identifier provided by the client.

The client may also track the current state of the service. When it connects, and before submitting any command, it receives the current view (described in Section VI) and position in the ledger. When it submits a request, the resulting transaction may be recorded only in this view and after this position. The optional client signature may cover the view and position as well as the request to ensure verifiability of this restriction.

If the RPC fails, the client reconnects (possibly to a different node): if the same view is still running, then a lookup based on the identifier will return up-to-date information about the command based on the local store. Otherwise, the client may wait for at least one transaction committed in the next view, so that the ledger includes a full record of the only view that may have recorded the client command.

F. Governance

To further illustrate our programming model, we describe the main commands used for governing the service.

Members have access to the governance RPC interface, whose code is set in $\mathcal{C}$ and parameterized by scripts stored in the RULE table. The interface comprises commands `Read`,

```
Gov.Read(caller: table, keys) {
  assert(MEMBER[caller].status in {accepted, active})
  assert(isGovTable(table))
  return multiRead(table, keys)
}
Gov.Propose(caller: action, sig) {
  assert(MEMBER[caller].status = active)
  assert(wellFormedProposal(action))
  pid = size(PROPOSAL) - 1
  PROPOSAL += (caller, action, sig, status=open)
  return pid
}
Gov.Vote(caller: pid, condition, sig) {
  assert(MEMBER[caller].status = active)
  assert(wellFormedCondition(condition))
  assert(PROPOSAL[pid].status = open)
  VOTE[caller, pid] := (condition, sig)
}
Gov.Complete(caller: pid) {
  assert(PROPOSAL[pid].status = open)
  tally = 0
  foreach(v in VOTE, v.pid = pid)
    // check status and condition
    if (MEMBER[v.voter].status = active &&
      execReadOnly(v.condition())) tally++

  assert(RULE["quorum"](pid,tally));
  PROPOSAL[pid].status := passed
  exec(PROPOSAL[pid].action)
}
Gov.Ack(caller: i, rho, sig) {
  assert(MEMBER[caller].status <> retired)
  assert(rho = ledgerRoot(i))
  if (exists(MEMBERACK[caller]))
    assert(MEMBERACK[caller].i < i)

  MEMBERACK[caller] = i, sig
  if (MEMBER[caller].status = accepted)
    MEMBER[caller].status := active
}
```

Listing 3. Pseudocode for parts of the governance RPC interface.

```
RULE["quorum"] :=
lambda (pid, tally) {
  voters := 0;
  foreach(m in MEMBER, m.status = active) voters++
  if (touches(RULE, PROPOSAL[pid].action))
    // changes to rules require unanimity
    return tally = voters
  if (touches(MEMBER, PROPOSAL[pid].action))
    // changes to members require super-majority
    return tally > 2 * voters / 3
  (...)
  else
    return tally > voters / 2
}
```

Listing 4. Sample governance rule defining vote quorums.

```
  foreach(m in MEMBER, m.status <> retired) members++
  if (getTxIndex()) > 11054442 &&
      PROPOSAL[5543].status = passed &&
      members < 10 )
    return true
  else
    return false
```

Listing 5. Sample pseudocode for a vote script

Propose, Vote, Complete, and Ack, whose pseudocode is given in Listing 3.

The Read command allows members with status at least *accepted* (see Section IV-C) to review the governance tables, for instance to obtain the current list of members and their credentials. (The interface may also support the read-only execution of member scripts on these tables.)

The other commands enable transparent governance of the service, by reaching joint decisions and implementing them as updates on its primitive tables. For verifiability, all these commands require signed requests from *active* service members. They operate on the following tables:

PROPOSAL records $p \rightarrow m, \mathit{action}, \{m : \mathit{action}\}, \mathit{status}$ when member m proposed to run *action*. Actions are restricted scripts that update selected governance tables after performing various checks. Proposals are implicitly keyed by their insertion index p. The status of each proposal evolves from open to either passed or withdrawn. Except for their status, proposals are never updated or deleted.

VOTE records $m, p \rightarrow \mathit{condition}, \{m : p, \mathit{condition}\}$ when member m voted on proposal p. Each vote includes a read-only script to support conditional voting. Votes may be updated as long as their proposal is open.

MEMBERACK records $m \rightarrow i, \{m : i, \rho_i\}$ when member m acknowledged witnessing the service running at transaction index i with Merkle-tree root ρ_i (see Section V). These member signatures are stored to track the liveness of the service members and their endorsement of relatively recent states of the ledger, which may be useful input for governance and auditing.

RULE records *name* → *condition* to define named auxiliary scripts executed as part of governance. As an example, Listing 4 outlines pseudocode for a rule to determine if a quorum of members is met to pass a proposal, depending on its proposed action.

Members call Propose to record their new proposed action. As shown in Listing 3, the command first checks that the proposer is an active member and that the script is well-formed, enforcing that it reads and updates tables within the scope of governance. For example, an action may add a new entry to the MEMBER only with a fresh member identifier, valid credentials, and status accepted; and it may update an existing MEMBER record only by setting its status to retired. A single action may include multiple such updates, ensuring their atomic execution. The command then records the signed action as an open proposal at the next available index *pid* in the table and returns that index.

Members call Vote to record (or modify) their vote on open proposals. The vote consists of a read-only script that returns true if the member approves the proposed action. In general, the script includes additional checks on the state of the store—this mechanism enables the service to check that all conditions set by the proposer and the voters are met at the time the action is executed. For example, when voting on a proposal to add two new members, the voter condition may require that (1) the decision is made before reaching a given transaction index; (2) another proposal has been passed; (3) the current number of accepted and active members is less than 10. The corresponding script is outlined in Listing 5.

Any member may call Complete to tally the votes and determine if they suffice to pass a proposal. This command checks that the proposal is still open, runs all its recorded preconditions, then calls the script stored at RULE "quorum" to determine the number of positive votes required, depending on the contents of the proposed action. If all these steps succeed, the action is executed and the proposal status is updated from open to passed. Thus, each proposal is executed at most once. (We omit here a command for updating proposals that fail to pass from open to withdrawn.) Note that all past and present proposals and votes are persisted in the store, enabling their detailed public auditing.

Finally, members regularly use `Ack` to confirm their participation in a service and endorse its state with a recorded signature. (The command checks that the member is not retired, that the signed endorsement correctly authenticates the ledger up to index i, and that it does not overwrite any previously-recorded endorsement.) In particular, accepted members need to explicitly acknowledge their participation to become active and thus gain access to `Propose`, `Vote`, and `Complete`.

The scripts stored in RULE may themselves be changed through a proposal. Listing 4 illustrates that such updates may require a super-majority. As shown in the next section, momentous decisions, such as the creation of a new service, may even require member unanimity.

IV. Service Protocols

Next, we describe our protocols for creating a node, for starting the service, for enrolling nodes in a service and removing them again, for rekeying the service, and for recovering the service after a major outage.

A. *Creating a Node*

The (untrusted) host for platform q creates a CCF node n by initializing a TEE with static code $\mathcal{C}$ and arguments α that include the unique identifier id_x of the intended service. The code $\mathcal{C}$ follows the state-machine depicted in Figure 1. When invoked in its initial *booting* state, it runs the following steps:

1) generate a signing key-pair $\mathsf{pk}_n, \mathsf{sk}_n$ for the node;
2) obtain a quote $\mathsf{qt}_n = \mathbf{QSig}_q[\mathcal{C}](\alpha \parallel \mathsf{pk}_n)$ from q;
3) output $\mathsf{rp}_n = \mathbf{Seal}_q[\mathcal{C}](\{n : \texttt{retire}\ \mathsf{id}_x\})$;
4) transition to state *created*.

The public key pk_n identifies n; it is used for TLS server authentication and for verifying protocol messages signed by this node; the corresponding signing key never leaves its TEE instance. The quote qt_n attests that n runs in a TEE on platform q with code $\mathcal{C}$, arguments α, and key pk_n. The sealed signed message rp_n enables an accelerated node-replacement procedure described in Section IV-D.

In state *created*, the node is not yet part of a service. It accepts a single call from a TLS client, exposing an interface with functions to either `Start` or `Join` a service, described in Sections IV-B and IV-D, respectively. It then transitions to state *trusted*, or *pending* in case it does not have the service secrets yet, or *down* if the call fails. Hence, a node can only become part of a single service throughout its lifetime.

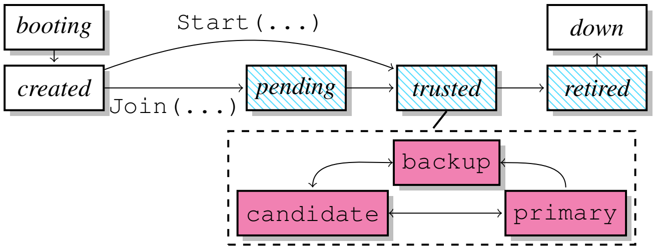

Fig. 1. State diagram for a node; states while part of a CCF service are blue (hatched). While in state *trusted*, a node is also part of the service's replication protocol (see Section VI). The corresponding sub-states are pink (solid). In case of a fatal error, each state directly transitions to *down*.

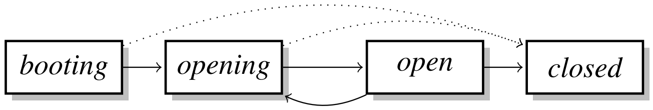

Fig. 2. State diagram for a service; in state *opening*, only the RPC interfaces for members and nodes are available; in state *open*, in addition, the user RPC interface is available. In case of a fatal error, each state directly transitions to *closed*.

B. *Starting a Service*

The `Start`ing protocol involves primitive tables that track the service configuration:

SERVICE stores $r \to \mathsf{cert}_x, \mathsf{status}_x$ where r is a recovery index (starting at 0), cert_x is the public-key certificate of the service, and status_x its current status. The table holds the full history of the service's long-term signing keys. Except for the status updates given in Figure 2, its entries are never modified or deleted. The entry with the highest index is active.

NODE stores $n \to \mathsf{pk}_n, \mathsf{qt}_n, \mathsf{netinfo}_n, \mathsf{sc}_n, \mathsf{status}_n$ where n is a unique node identifier (integers counting from 0), pk_n is its signature-verification key, qt_n is its remote attestation quote, $\mathsf{netinfo}_n$ is its hostname and port, sc_n is an optional shutdown certificate for a prior node on platform q (see Section IV-E), and status_n is its status, ranging over pending → trusted → retired (see Figure 1). Similarly, the table holds the history of past and present nodes for the service, and only its status field is mutable.

CODEID stores $v \to \mathbf{H}(\mathcal{C}_v)$ where v is a code version number and $\mathbf{H}(\mathcal{C}_v)$ is the corresponding code digest. The entry with the highest version is active. The table records the digests used for verifying quotes from nodes as they join the service. It enables code update, subject to governance.

The `Start` command takes as argument the initial state of the key-value store as transaction τ_1 that initializes the governance tables MEMBER, USER, RULE, CERTCONFIG, CERT, CRL, and any application tables. It runs the following steps:

1) generate the service key-pair pk_x,sk_x, its intermediate certificate cert_x, and its first data-encryption secret s_d.
2) validate τ_1, for instance checking that all initial MEMBER entries have valid credentials and an *accepted* status;
3) transition to state *trusted* and start the replication protocol with itself as (single) primary replica: as detailed in

Section VI, this writes a first transaction in the ledger that initializes the replication protocol-specific tables with a single entry NODE $0 \to \mathsf{pk}_n, \mathsf{qt}_n, \mathsf{netinfo}_n, \mathsf{sc}_n, \mathsf{trusted}$ for the starting node. Until more replicas join the service, all transactions commit immediately.

4) write initial entries SERVICE $0 \to \mathsf{cert}_x, \mathsf{opening}$ and CODEID $0 \to \mathbf{H}(\mathcal{C})$ followed by τ_1;
5) start the service.

C. Opening a Service

The service uses its signing key and intermediate certificate to issue certificates for the public keys pk_n of each of its *trusted* nodes—initially just the starting node. This enables clients to authenticate any of these nodes as part of the service before issuing commands.

In state *opening*, the service accepts governance commands to complete its bootstrapping. Its initial members (defined by MEMBER records in τ_1) can connect to the primary, check the certificates it presents, call the `Ack` command to upgrade their status from *accepted* to *active*, and start governing the service. The service also accepts connections from nodes attempting to join the service, as described in Section IV-D. Conversely, users cannot yet connect to the service.

Once the members are satisfied that the service is sufficiently replicated and that its configuration is trustworthy, they can use its governance commands to propose, vote, and complete the update SERVICE$[0].\mathsf{status} = \mathsf{open}$. This fully opens the service and enables its application commands.

D. Adding a Node to a Service

The `Join` command takes as arguments the service certificate cert_x and the network address of a node already trusted to run the service. The joining node n connects to the service with mutual authentication, presenting its fresh qt_n as client credentials and verifying the quote-certificate and its endorsement by the service certificate cert_x presented by the server. Then, using this secure channel, it sends $\mathsf{netinfo}_n$ and sc_n to the service; within a transaction, the service validates qt_n against the latest code digest recorded in CODEID and the state of its certificate store, and then adds a NODE entry for n with state *pending*. In this state, the node begins receiving state updates from the service, enabling it to replicate its ledger, but is not yet part of the replication protocol, and does not yet have access to the service secrets.

The join protocol resumes when the new node gains status *trusted* through governance. This authorizes the service to share the service signing key sk_x and encryption secrets s_d with the new node over their secure channel and, finally, to reconfigure the replication protocol with the joining node as active replica.

The join protocol can be simplified in several cases, enabling new nodes to join as *trusted* after validating their quote but without a full round of governance.

- In state *opening*, the service can immediately accept new nodes, since the members will review them before opening the service.
- The new joiner may provide additional evidence that it is replacing a node on the same platform q, for instance after a shutdown of its host or a code update. As long as the prior node is currently *trusted* by the service and signed that evidence, it can be seamlessly replaced by its successor.
  This additional evidence may be either a shutdown certificate sc_n (Section IV-E) or a retire certificate rp_n (Section IV-A), necessarily unsealed by a node with code $\mathcal{C}$ created later on platform q. In both cases, the transaction that records the new node as *trusted* also records the evidence and marks its predecessor as *retired*.

E. Removing a Node from a Service

A node can be removed from a service through governance by updating its status to *retired* in NODE, as a result of a planned shutdown or a crash. In all these cases, the node attempts to complete the following steps:

1) transition to state *retired*;
2) create a shutdown certificate $\mathsf{sc}_n = \{n : \texttt{shutdown}\ \mathsf{id}_x\}$;
3) erase its state and return sc_n to the host.

The service accepts calls to report shutdowns and, after verifying sc_n against NODE$[n]$, it records the evidence and updates the node status to *retired*. Keeping track of shutdown certificates accelerates node replacement, and also yields some forward secrecy, as a retired node cannot leak any secret.

F. Rekeying a Service

To limit the scope of key compromise, the service supports updates of its data-encryption master secret s_d. As part of governance, such an update may be triggered after a service reconfiguration, so that only the nodes trusted after the reconfiguration have access to newly-encrypted data. Updates are tracked by an auxiliary governance table:

SECRET stores $d \to (\mathbf{aenc}[\mathsf{k}^n_r](d, s_d))_{n\ \mathit{trusted}}$ where d is the rekeying index (counting from 1) and, for each currently-trusted node n, $\mathbf{aenc}[\mathsf{k}^n_r](d, s_d)$ encrypts the dth data-encryption secret under a key established between n and the current primary r. The entry with the highest index indicates which secret to use for data encryption in the ledger after the transaction including this record.

To update the key, the current primary increments d, samples a fresh secret s_d, encrypts it for each currently-trusted backup, and writes the resulting entry to SECRET. The replication protocol ensures that, as the rekeying transaction commits, all trusted nodes get access to the new secret and start using it from the same transaction index. (The ledger is used here as a public reliable transport.) Wrapping this secret by authenticated encryption under ephemeral keys separately established between nodes provides additional protection against long-term TEE compromise.

G. Recovering a Service

A service that suffers more node outages than its replication protocol can withstand safely stops to make progress, and requires privileged intervention of its consortium to resume.

We describe an optional protocol that allows members to recover a service from a ledger, even if none of a service nodes are live. The protocol depends on the availability of a recent copy of the ledger, and may cause the loss of transactions not recorded in any available copies. It also supposes that the members confirm that the service has actually stopped before enabling its recovery.

The protocol uses an additive key-sharing scheme, enabling a key-wrapping key k_z to be split into multiple shares—one for each active member—so that a quorum of shares yields back k_z and the shares otherwise leak no information about k_z. (The size of the quorum can be configured as part of governance.) The protocol relies on an additional public table:

SHARE stores
$z \rightarrow \mathbf{aenc}[\mathsf{k}_z](s_0, \dots, s_d), (\mathbf{share}_m(\mathsf{id}_x, z, \mathsf{k}_z^m))_{m\, active}$
where z is the keyshare index, k_z is the zth shared key-wrapping key, d is the rekeying index, $s_0, \dots, s_d$ are the secrets used for encrypting the ledger so far, and, for every entry $m \rightarrow \mathsf{cert}_m, \mathsf{ksk}_m, \mathit{active}$ in MEMBER, $\mathbf{share}_m(\mathsf{id}_x, z, \mathsf{k}_z^m)$ encapsulates m's share of k_z under her public key ksk_m.

If key-sharing is statically enabled in $\mathcal{C}$, every time the members complete a proposal to share the service encryption secrets, the service (1) increments z, (2) generates a fresh key k_z, (3) encrypts the relevant secrets s_d under k_z, (4) splits k_z into shares k_z^m for the currently active members, (5) encrypts those shares under their recorded public keys, and (6) records the resulting new SHARE. As for rekeying, the ledger provides reliable delivery of encrypted keying materials.

The recovery protocol is a variation of the starting protocol: as described in Section IV-B, the service starts and generates fresh keying materials, including a new public-private keypair, intermediate certificate, and data encryption secret, and it starts accepting both new joining nodes and governance commands. The key differences are as follows:

- Instead of starting with an empty ledger, the service resumes from an existing ledger, reads it, validates it, reconstructs the state of its public tables, increments the recovery index r, and writes a resumption entry SERVICE $r \rightarrow (\mathsf{cert}_x^r, \mathsf{opening})$ followed by τ_1^r.
- The current state of the ledger indicates in particular (the indexes of) the prior data encryption secrets necessary for completing the recovery and the acceptable encrypted keyshares that enable it.
- Once the service is *opening*, the members selected for recovery (as defined by MEMBER in the ledger updated with τ_1^r) connect to the service and, using its governance commands, review the proposed state for recovery. Once they are satisfied with its configuration, they release their expected share to the service and vote for re-opening it.
- Trusted with a quorum of shares, the service can then decrypt the prior encryption secrets, decrypt and process encrypted entries in the ledger, reconstruct the state of its private tables, and finally update its status to *open*.

By design, the recovery attempt fails unless a quorum of members actively participate and contribute their key shares. Also, the service resumes with fresh nodes and a fresh public-key service certificate, preventing client confusion.

(A variation of this recovery protocol does not require members to be trusted with key shares, but it requires instead at least one surviving replica to run the recovery and share the prior service secrets with new replicas.)

V. Ledger Encryption and Verifiability

Next, we present our mechanisms for auditability and universal verifiability, to support strong service integrity even if some of the nodes or their keys eventually get compromised.

As usual with blockchains, these guarantees stem from reaching a consensus on (a stable prefix of) the ledger. To this end, replicas exchange and store signatures over the whole ledger contents, represented as the root of the binary Merkle tree for all its serialized transactions—see also Section VI.

We use a Merkle tree [58] instead of a linear chain of hashes to enable logarithmic access to past transactions. As a first approximation, the tree is never updated, only extended with new transactions added to the ledger (see Section II-B). Using a tree is convenient in our setting because most clients are not expected to keep a full copy of the ledger. Instead, they can select their own level of abstraction, keep a local copy of the transactions they care about, together with intermediate hashes in the tree and recently-signed roots of the tree, and use them as independent evidence of the correct execution, ordering, and commitment of these transactions in the ledger.

A. Ledger Encryption and Transaction Hashes

Recall that confidentiality is configured as part of the service definition, enabling application trade-offs between privacy and transparency: each table is optionally labelled with a confidentiality level ℓ, and is otherwise considered public.

For each transaction, all data with label ℓ is encrypted using a key specifically derived from the current service encryption secret: $k_{v,i}^{\ell} = \mathbf{KDF}[s_d](\ell \,\|\, v \,\|\, i)$ where **KDF** is a key derivation function, v is the current view, and i is the current transaction sequence number in the ledger. (The replication-protocol view is included in the derivation context to ensure that every key is used at most once for encryption.) After partial encryption, the transaction is hashed and inserted in the Merkle tree. Hence, the ledger authenticates the whole sequence of transactions while hiding parts of their contents. The main benefit of using separate keys is to enable release of data in the ledger at the granularity of single transactions. In particular, $k_{v,i}^{\ell}$ may be released to i's client so that it may independently verify the recorded outcome of its command.

For simplicity, we present the ledger format with a single confidentiality label. After formatting, every transaction consists of two bytestreams of key-value records in public and private tables, respectively:

$$\tau_i \quad = \quad \tau_i^0 \,\|\, \tau_i^{\ell}$$

where τ_i^0 and the length of τ_i^{ℓ} are public, whereas the content of τ_i^{ℓ} is confidential. In the ledger, the transaction record is

$$\tau_i' \quad = \quad \tau_i^0 \,\|\, \mathbf{encrypt}[k_{v,i}^{\ell}](\tau_i^{\ell})$$

Using encryption in counter-mode, τ_i and τ_i' have the same length, which enables efficient in-place processing.

In the Merkle tree, the transaction is logged by inserting at position i a hash tag computed in two steps:

$$\begin{aligned} h_i^\ell &= \mathbf{H}(\mathbf{bind}[k_{v,i}^\ell] \,\|\, \mathbf{encrypt}[k_{v,i}^\ell](\tau_i^\ell)) \\ h_i &= \mathbf{H}(\tau_i^0 \,\|\, h_i^\ell) \end{aligned}$$

where $\mathbf{bind}[k_{v,i}^\ell]$ cryptographically binds the key $k_{v,i}^\ell$ to ensure that ciphertext authentication implies plaintext authentication. Otherwise, a malicious primary might return to the client a forgery $k^\star \neq k_{v,i}^\ell$ and $\tau_i^{\ell\star} = \mathbf{decrypt}[k^\star](\mathbf{encrypt}[k_{v,i}^\ell](\tau_i^\ell)))$. In counter-mode, $\mathbf{bind}[k_{v,i}^\ell]$ may simply consist of the encryption of a constant block at counter 0, with the encryption of τ_i^ℓ starting at counter 1.

We assume our encryption scheme (including the binder) provides indistinguishability against 1-time fixed-sized chosen plaintext attacks. In combination with key binding, signature- and hash-based authentication, this yields authenticated encryption for the whole ledger.

B. Signed Receipts and Confidentiality

For every command to be independently verifiable, the application programmer must ensure that both the request and the response are computable from the resulting transaction τ_i. This can be trivially coded by recording them in auxiliary tables REQUEST and RESPONSE within the transaction, with more compact, application-specific encodings available for most commands (see e.g. Listing 2).

We complete the description of client RPCs of Section III-E by explaining the early results and signed confirmations returned by the service. As the primary completes the transaction, it immediately returns to the client its index i and partially-encrypted encoding τ_i' being recorded in the ledger. (In principle, it may also replace parts of the transaction not meant to be returned to the client with their hashes.) This binds τ_i to the (provisional) ledger contents, but does not yet grant the client access to its confidential parts. Later, once the service has committed the transaction, it returns to the client both signed evidence of its commitment in the ledger and the specific keys used to unseal the confidential parts of the response previously encrypted in τ_i'. Although the application might grant earlier access to the encrypted response, releasing confidential information only for committed transactions facilitates reasoning since the ledger records any such release.

The primary issues all command confirmations based on its latest signed root of the ledger, at some transaction index $j \geq i$, without recalling any details of past transaction processing. The most basic *signed confirmation* for a client command that produced τ_i thus consists of $(p, v, j, \{p : v, j, \rho_j\})$ where p, v, j are the current primary, view, and index and where ρ_j is the root of the tree including all transactions up to index j, together with $\log(j)$ intermediate hashes to recompute ρ_j given the leaf h_i. As detailed in the next section, a more comprehensive confirmation also includes indexes $k \in i..j$, signatures $\{b : v, k, \rho_k\}$, and intermediate hashes to verify roots signed by backups b, providing evidence that a quorum of replicas agree that τ_i committed in the ledger at index i.

Given τ_i and its key $k_{v,i}^\ell$, the client can first recompute h_i then, using the intermediate hashes, the roots ρ_k, and finally verify their signatures. Given τ_i^0, h_i^ℓ, and any such receipt, anyone can similarly recompute ρ_j and verify the signatures, thereby checking the public part and a commitment to the private part of the transaction are committed in the ledger. In particular, h_i^ℓ may be recomputed from the ledger, enabling the public verification of its integrity.

C. Checkpoints

As described in Section IV, the node-join and recovery protocols involve replaying the complete ledger to reconstruct the current state of the key-value store. In addition to the ledger that fully records its history for auditing purposes, the nodes may also periodically save the state of the store at a given index (partially-encrypted under keys derived from s_d) and execute a transaction to record the resulting authentication hash in a table. The checkpoint is considered complete once the enclosing transaction commits.

Joining nodes can then start replaying the ledger from any recent complete checkpoint provided by the host. Similarly, backups that are lagging behind the current consensus may use a complete checkpoint to catch up—this also helps bound the memory required to withstand Byzantine attacks.

After confirming that the ledger records a complete checkpoint, replicas may safely erase earlier encryption secrets $s_{d'<d}$. In combination with rekeying, this enables forward secrecy for application data that has been deleted or overwritten in the store.

VI. Replication Protocol

As outlined in Section II-B, the replication protocol persists key-value updates in various tables, grouped into transactions τ_i, then into batches (to amortize the cost of signatures) then into views (to support changes of primary). Within each view, all transactions are proposed by the primary, forwarded to the backups, then committed or discarded. Integrity means that the series of transactions follows both the protocol and application logics. Consistency means that all nodes observe prefixes of the same series of committed transactions. Safety is both integrity and consistency.

The replication protocol has two main configurations. We describe first a generic core protocol, then its configuration details. In its Byzantine configuration (Section VI-B), n replicas ensure (1) safety against $f < n/3$ corrupt replicas and any number of crashes; (2) confidentiality against any crashes; and (3) progress against $f < n/3$ corrupt replicas. In its simpler Crash-Fault configuration (Section VI-A), for which we present a performance evaluation, n replicas ensure progress against $f < n/2$ crashes, but they provide safety and progress only against TEE crashes.

In all configurations, the protocol persists a ledger that enables anyone either to verify that the protocol ran correctly or to blame corrupt nodes that contributed to faulty quorums. To this end, the ledger records the signed payload of protocol messages into primitive tables (omitting any parts of the message that can be recomputed from prior records in the ledger). We describe these tables below. Recall that we write $\{r : v\}$ for message v signed with the private key of node r.

LOG stores $\{p : \texttt{log}\, v, i, \rho_i\}$ when primary p in view v ends a transaction at index i with Merkle-tree root ρ_i. The contents of the ledger determine the signed values of p, v, i, and ρ_i. The primary for view v signs at most once for each transaction index i, and typically signs large batches of transactions jointly authenticated by ρ_i.

ACK stores $r \to i, \{r : \texttt{ack}\, v, p, i, \rho_i\}$ when backup r verified and recorded transactions proposed by primary p in view v ending with a LOG record at index i with root ρ_i. The record includes i and the signature, whereas the ledger determines the current values of p, v, and the root ρ_i being acknowledged.

The primary ends every batch of transactions with optional, signed ACKs received from its backups (referring to LOGs at prior indexes in the view) followed by its own signed LOG.

In CFT configurations, we say that a transaction j commits once the ledger records ACKs and a LOG for a quorum of replicas with the same view number, and any indices $i \geq j$ (where the Merkle tree with root ρ_i extends the one with root ρ_j). BFT configurations require a stronger condition for commitment: the ledger must record ACKs and a LOG for a quorum of replicas with indices greater than an index that satisfies the CFT condition for commitment.

Our next table supports view changes by recording the local state of replicas for prior views:

VIEW stores $r \to w, \mathcal{P}_{r\to w}, \{r : \texttt{view}\, w, \mathcal{P}_{r\to w}\}$ when replica r moved to view w after signing transactions in earlier views summarised in $\mathcal{P}_{r\to w}$. The details of the summary $\mathcal{P}_{r\to w}$ depend on the protocol configuration.

In the ledger, every view w starts with a 'view-change' transaction that includes a quorum of supporting VIEWs followed by a first LOG of the new primary that proposes to resume in view w at a given index and root. This first transaction must justify any resulting truncation of prior views.

The last table supports dynamic reconfiguration:

CONFIG stores $c \to N$ when c is the next configuration identifier and N is a subset of *trusted* replica identifiers, subject to governance. A configuration stored in CONFIG is *active* until the next configuration commits, with usually one or two active configurations at a time.

The initial transaction in the ledger is special, of the form

$$\begin{array}{ll} \textsc{node} & 0 \to k_0^+, \mathsf{trusted}; \\ \textsc{config} & 0 \to \{0\}; \\ \textsc{view} & 0 \to \{0 : \texttt{view}\, 0\}; \\ \textsc{log} & \{0 : \texttt{log}\, 0, 0, \rho_0\} \end{array}$$

We are now ready to describe the core replication protocol followed by (honest) replicas. Every replica in state pending or trusted maintains local state: a current view v; a well-formed ledger (with a branch for each recent view $w \leq v$); for each replica, any VIEW record last sent or received; for each recent view $w \leq v$, any j, ρ_j signed last. As a function of each branch of its local ledger, it also maintains current configuration set(s) N, view w, primary p, index i, root ρ_i, and commit index c. (Replicas in state retired are safe to shut down.) Their local state machine is as follows, starting as backup:

In any state:

Receive, verify, and record a batch of transactions that extends the ledger beyond its current commit index. If it includes the first transaction in view $w > v$, set $v := w$ and become backup.

Receive and verify a VIEW message for w. If $w > v$ (and received $f + 1$ VIEW messages for w in the BFT configuration), set $v := w$, update current VIEW message, and (depending on the protocol configuration) become backup. In all these cases, respond with current VIEW message.

Timeout, set $v := v + 1$, update current VIEW message, broadcast it, and (depending on the protocol configuration) become candidate.

Exchange transactions with other replicas, answering at least requests for transactions signed by the replica.

backup (initial state): Issue signed ACKs for any transaction received in current view.

primary: Prepare and replicate transactions that extend the current view. Echo ACKs and issue LOGs to complete transaction batches and advance the commit index. Collect ACKs from other replicas.

candidate: Receive and verify VIEWs for v. Request any missing transaction that may not be truncated. Issue a view-change transaction and become primary.

Except for the embedding of signed messages into the ledger, the protocol is very close to RAFT in the CFT configuration and PBFT in the BFT configuration.

A. Crash-Fault Tolerant Configuration

We first implemented a simple protocol configuration based on RAFT [54]—we refer to their original paper for a detailed design discussion and analysis. In this configuration, a quorum consists of a strict majority of replicas: $f + 1$ out of $n = 2f + 1$. Compared with RAFT, CCFmessages are signed to allow blaming corrupt replicas during auditing.

View changes are randomized: any replica that times out becomes candidate for the next view, with thresholds such that, with high probability, a single candidate appears. For each replica r, the VIEW payload $\mathcal{P}_{r\to w}$ records the last view $v < w$, index j, and Merkle-tree root ρ_j signed by r in any LOG or ACK message. A valid view-change transaction must include messages from a quorum of replicas such that the contents w, i, ρ_i in the first signed LOG of the new primary proposes to continue the ledger from a state *at least as up-to-date as* as any v, j, ρ_j in the supporting VIEWs. This simple mechanism ensures that the new primary can immediately start processing new transactions, without the need to fetch batches stored at other replicas.

It is possible to configure CCF to have signatures only on LOGs but not on ACKs. This does not affect the safety and liveness guarantees of the protocol, but it may prevent blaming corrupt backups during auditing.

B. Byzantine-Fault Tolerant Configuration

This more defensive configuration is based on PBFT [16]. A quorum now consists of two thirds of the replicas: $2f + 1$ out of $n = 3f+1$, although a smaller quorum of $f+1$ replicas may suffice to confirm the correctness of a computation given its result.

The core replication protocol in its Byzantine configuration requires *two* roundtrips between the primary and its backups to record in the ledger signed evidence that a given transaction is committed—that is, a sequence of 6 message flights from the initial client request to the verifiable commit certificate from the primary. We outline below an efficient mechanism adapted from PBFT [16] to return the same evidence to clients in just 4 message flights. We leave its implementation and performance evaluation as future work.

- For each batch in view v and transaction index i, before signing its LOG or ACK message, replica r computes a nonce $n^r_{v,i} = \mathbf{KDF}[n^r_d](v \,\|\, i)$ using a local secret n^r_d, computes its hash $h^r_{v,i} = \mathbf{H}(n^r_{v,i})$, and appends it to the signed payload. The replica communicates $h^r_{v,i}$ with the signature to enable its immediate verification, but temporarily withholds $n^r_{v,i}$.
- Backups send their ACK messages and hashed nonce to all other replicas (not just to the primary).
- Replicas receive these ACKs and verify that their signatures match the message they sent. They release their nonce $n^r_{v,i}$ once they have collected $2f+1$ such messages including their own, and may immediately send it with their earlier signature to the clients of this batch.
- A client can gather signed evidence of commitment by collecting $2f + 1$ matching signatures with the corresponding nonces (together with auxiliary hashes to relate committed transactions to their signed root, as described in Section V). In combination, these signatures ensure that the transaction will be committed, since at least one honest signer that released the nonce will be part of any view change quorum.
- As before, the primary regularly records these signatures in the ledger, to further ensure that the commitment of these transactions become universally verifiable.

This protocol configuration supports robust view changes that guarantee safety and liveness if fewer than $1/3$ of the replicas are corrupt. The primary for every view is predetermined, using the next replica in round-robin according to their ordering in the NODE table. This is important to prevent malicious nodes from being primaries indefinitely.

The summary recorded in VIEW messages is adapted as follows: $\mathcal{P}_{r \to v'}$ records $v, i, \rho_{v,i}, P, w, j, \rho_{w,j}$ where replica r changed to view v', where v and i are the largest view and index within that view for which r released a nonce and $\rho_{v,i}$ is the corresponding Merkle-tree root, where P is a set with one signed LOG and $2f$ signed ACKs from different replicas for $(v, i, \rho_{v,i})$, and where j and w are the last view and index within that view for which r signed a LOG or ACK (with $j \geq i$ and $w \geq v$) and $\rho_{w,j}$ is the corresponding Merkle-tree root.

A valid view-change transaction to v' consists of a quorum of VIEW messages and a first LOG message from the primary for v' continuing the ledger from some $(v^0, i^0, \rho_{v^0,i^0})$ that (1) extends a $(v, i, \rho_{v,i})$ chosen from one of the VIEW messages such that it has the largest i from among those with the largest v, and (2) includes the longest prefix of the $(w, j, \rho_{w,j})$ that is shared by at least $f + 1$ included VIEW messages and extends $(v, i, \rho_{v,i})$.

The backups check the validity of the view-change transaction and send ACKs to the primary. Both the primary and the backups update their state to match $(v^0, i^0, \rho_{v^0,i^0})$. First, they discard transactions in forks from the ledger and undo the effects of those transactions on the key-value store. Then they add new transactions to the ledger and execute those transactions to reflect their effects in the key-value store.

The BFT configuration tolerates malicious failures but requires more replicas and more messages to tolerate the same number of failures. Both configurations return a commitment receipt to the client in two round trips and require a single signature per replica per batch of transactions, but the BFT configuration requires verifying more signatures. Additionally, BFT backups need to execute the commands in a transaction batch to verify the correctness of the write set generated by the primary before signing an acknowledgment.

VII. IMPLEMENTATION

We implemented CCF in C++17 using the Open Enclave SDK, targeting the Intel SGX TEE. Using CCF on other TEEs will be possible when Open Enclave supports them, with relatively little additional porting work.

The code is split between the trusted enclave and the untrusted host. The well-known CLOC tool counts roughly 14,000 lines of code (LOC) for the enclave, 1,900 LOC for the host, and 5,800 LOC shared by both. The numbers exclude the Open Enclave SDK, optional transactions engines (e.g., an EVM and the Lua interpreter) and third-party code, e.g. libuv.

The source code and its documentation are available at **`https://github.com/Microsoft/CCF`**. Our EVM implementation, eEVM (roughly 2,800 LOC), is available at **`https://github.com/Microsoft/eEVM`**.

A. Untrusted Host

The host is responsible for handling I/O for the enclave. If the host is compromised, it can cause a denial of service by not handling I/O. However, the data the host handles is always integrity protected, and, for confidential data, is encrypted, such that the host cannot alter communications or observe confidential communications.

Communication Between Host and Enclave. The host and enclave communicate via a pair of lock-free multi-producer, single-consumer ringbuffers rather than via enclave transitions (e.g., ECALL and OCALL on Intel SGX). This ringbuffer pair acts as a virtual network interface, requiring communication to be serialized and eliminating possible security bugs due to pointer dereferencing inside the enclave. The serialization performance cost is ameliorated by (i) not paying the performance cost of enclave transitions and (ii) communication

https://OpenEnclave.io

primarily being in the form of forwarding network traffic, which is already serialized.

Network Traffic. The host manages both inbound and outbound network connections. The enclave can request, over the ringbuffer, that the host listen on some port or connect to some address and port pair. The host makes the necessary system calls. When inbound data arrives, it is written to the ringbuffer with a connection identifier. When the enclave places outbound data in the ringbuffer, the host is responsible for writing it to the appropriate connection.

Storage. The enclave writes ledger updates to the ringbuffer and expects the host to extend the ledger on disk or other persistent storage. The ledger data emitted is integrity protected and the confidential portion is encrypted.

B. Trusted Enclave

The enclave is responsible for secure communications, command execution, maintaining the key-value store, and replication. Modules are placed inside the enclave when their malicious operation would compromise the integrity or confidentiality of the system.

Secure Channels. We ported the mbedTLS library to our enclave environment. We use it to terminate TLS connections inside enclaves and to manage public-key certificates. Hence, the host observes only TLS negotiation and encrypted traffic. We fix the choice of cryptographic ciphersuites to use ECDHE, ECDSA, AES256-GCM, and SHA2. The elliptic curves used are statically configurable. The TLS module authenticates the connection and forwards plaintext data and peer identity to a connection handling module, using the Server Name Indication extension (SNI) to select the correct command interface: (i.e., “members”, “users”, or “nodes”). This is used for both client-to-node connections and node-to-node connections when joining a service, as described in Section IV-D. Clients and nodes then communicate via JSON-RPC over TLS.

Cryptography. We use formally verified, side-channel resistant implementations of cryptographic algorithms from the Everest project [10]. We replaced some of the algorithms in mbedTLS with their Everest implementations, both for performance and for side-channel resistance. We also implemented and verified a custom library for binary Merkle trees based on the SHA2-256 compression function, with fast incremental updates (up to 3M tx/s) and caching of intermediate hashes.

Command Execution Engines. Commands received via JSON-RPC are handled by a command execution engine. Each type of command can be handled by a different engine. When building a service, the “nodes” and “members” engines are fixed, while the “users” engine is specified by the service. We implemented a C++ engine for “nodes” commands, and ported the Lua virtual machine to the enclave for “members” commands. A service can handle “users” commands using custom C++, Lua, an EVM we implemented specifically for use inside an enclave, or any other engine provided by the service developer atop the CCFexecution engine API.

Transactional Key-Value Store. The current state is reflected as a collection of key-value stores (i.e., tables) in enclave memory. These transactional key-value stores are implemented as a sequence of compressed hash-array mapped prefix trees [64], giving a compact representation of locally but not globally committed changes using structural sharing. Transactions can be processed in parallel, and commit locally when they have no conflicts. The key-value store maintains both opacity and strict serializability. A locally committed transaction receives a monotonic index, establishing a total order. These write sets are encrypted, hashed, inserted into the Merkle tree, and then sent, in order, both as deltas to the replication module and as ledger updates to the host.

Replication. Replication messages are integrity protected using keys derived from s_d, but are not encrypted and are sent over raw TCP. This is an optimization: when a node sends a sequence of (already encrypted and integrity protected) state deltas to another node, it does not reencrypt them. Instead, it sends an integrity protected header (containing the index range of deltas to be sent) to the host, which then appends the requested state deltas, as read from the local ledger. The receiving node verifies the state deltas on receipt, applies them to its key-value store and appends them to its local ledger.

Command Forwarding. Commands can be submitted to any node configured to run the service. If the node is not the current primary, it verifies the command signature if one is present, it authorizes the command, and, if successful, it detects whether the command is read-only or may write to the key-value store. If the command is read-only, it executes it locally. Otherwise, it forwards it for execution on the primary on a secure channel. This allows clients to be unaware of which node is the current primary, and enables us to distribute the signature verification workload and the read-only command workload across nodes.

VIII. Evaluation

We now evaluate the performance of our implementation of CCF. We measure a crash-fault-tolerant configuration in which only primary signatures are turned on (see Section VI). To enable comparison with related work, we selected the following realistic benchmarks: (1) Small Bank [3] was originally proposed as a SQL benchmark. It involves three tables and is designed to mirror a small set of simple banking operations. We use Small Bank in three configurations: (A) the database fits into enclave memory; (B) the database does not fit and the client uses random accounts to maximize page faults; (C) the database fits and the client explicitly signs every command in addition to sending it over TLS and the service checks and stores every signature. (2) ERC20 is a standard Ethereum smart contract for token transfer. We invoke its `Get` and `Transfer` functions. (3) CryptoKitties is a popular game about breeding cats implemented as a complex Ethereum smart contract. We invoke the `mixGenes` function, which is

https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
https://www.cryptokitties.com

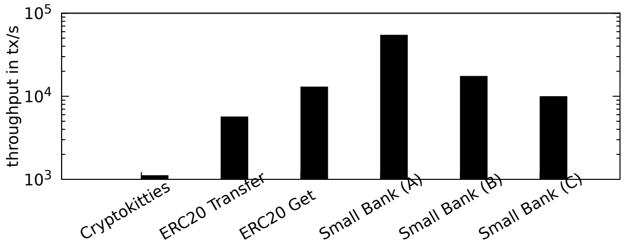

Fig. 3. Throughput measured for 9 CCF nodes and 1 client from our VM pool; all values are averaged over 5 runs with 100,000 transactions each. The primary issues signed `log` messages for batches of 10,000 transactions.

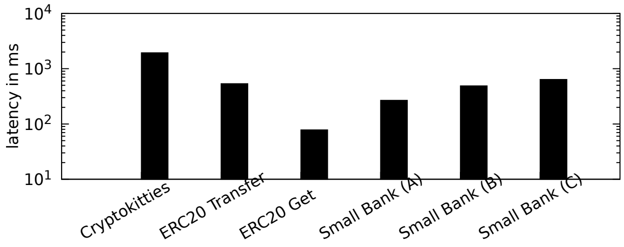

Fig. 4. Global commit latency of CCF; same setting as before

significantly more memory and compute intensive than the functions in ERC20.

We run all experiments in Microsoft Azure on SGX-enabled VMs with four cores of an Intel Xeon E-2176G 3.70GHz CPU, 16GB RAM, and Ubuntu 18.04.2 LTS. We use 10 VMs—9 replica nodes and 1 client—with 4 servers on the east coast of the United states and 5 in western Europe. In all experiments, we compose our VM pool evenly from both locations. This is the most durable configuration, but it also exhibits the worst performance characteristics.

Figures 3 and 4 show the throughput and latency characteristics of CCF for the described experiments. Latency is averaged over all transactions in an experiment and counts the time from sending a command on the client to receiving a global commit confirmation. For smaller number of nodes, performance characteristics are similar. The comparably low performance of Small Bank (B) is not unexpected, as SGX page-faults are very expensive [55].

To put these numbers into perspective, we now compare them with related work. (We give a general overview on related work in Section IX.) Dinh et al. [24] use Small Bank with 8 servers and 8 clients in one data center to compare (1) Hyperledger Fabric [5] version *0.6.0-preview*, private Ethereum deployments using (2) the geth client version *1.4.8* and (3) the Parity client version *1.6.0*, and (4) an unspecified version of H-Store [34], a state-of-the-art sharded database. Each of their servers is equipped with an Intel Xeon E5-1650 3.5GHz CPU with 32GB RAM. On this configuration, Hyperledger Fabric is reported to achieve a throughput of 1,122 transactions per second (tx/s), geth 1,122 tx/s, Parity 46 tx/s, and H-Store 21,596 tx/s. FastFabric [30] improves the throughput of Hyperledger Fabric. It achieves 19,112 tx/s running transactions that transfer funds between two accounts on one client and 14 servers under the same switch. These systems do not provide confidentiality for smart contract execution.

Other than the aforementioned systems, the Ekiden system provides confidentiality for smart contract execution, using Intel SGX. For a setup with one compute enclave running on an Intel Core i7-6500U CPU with 8GB RAM, periodically submitting state updates to a four-node Tendermint [14] consensus network which are all in the same data center, Cheng et al. [18] report ca. 1,000 tx/s throughput and ca. 0.6s latency for ERC20 `Get` and ca. 600 tx/s and ca. 0.6s latency for ERC20 `Transfer`. Solidus [17] is a confidential permissioned ledger that achieves consensus among a large number of users through a small group of banks. Solidus hides transaction values and the identities of transaction entities. The use of cryptographic primitives such as Oblivious RAM and Schnorr signatures results in a throughput of < 9 tx/s.

In comparable single data-center setups, CCF achieves 57,641 tx/s throughput and 153ms latency for Small Bank with eight servers and 17,708 tx/s throughput and 57ms latency for ERC20 `Get` and 5,963 tx/s throughput and 276ms latency for ERC20 `Transfer` with five servers. While all discussed systems aim at different use cases and offer different guarantees in terms of fault tolerance and security, we believe that these numbers demonstrate the efficacy of CCF.

IX. Related Work

Bitcoin [52] created a cryptocurrency using a proof-of-work protocol to ensure that users agree on a set of transactions. Bitcoin generates a new block every 10 minutes and users typically wait for 6 blocks before a transaction is considered committed, resulting in confirmation latencies in the order of an hour. Many other cryptocurrencies adopted consensus based on proof-of-work. Ethereum [67] also uses proof-of-work, but supports general transactions.

Several recent systems propose higher-performance blockchains. Honey Badger [50] proposes using a Byzantine fault-tolerance protocol to build a cryptocurrency using a designated set of servers. RScoin [23] proposes a centrally banked cryptocurrency based on a variant of the two-phase commit protocol. In the permissioned setting, Hyperledger Fabric [5] decomposes the tasks of executing client commands and ordering the corresponding transactions. The ordering service may be a single trusted server or a CFT or BFT network of nodes. Bitcoin-NG [25] proposes using proof-of-work to elect a leader that then publishes blocks of transactions. Hybrid consensus [57] can periodically select a group of nodes using proof-of-work and then runs Byzantine agreement amongst those nodes to confirm transactions, until a new group is chosen. The permissioned SBFT [32] proposes a variant of PBFT [16] that scales to larger consensus groups. The unpermissioned Byzcoin [39] also builds on PBFT and dynamically forms consensus groups. Algorand [28] uses a mechanism based on Verifiable Random Functions to select the nodes that participate in Byzantine agreement in a private and non-interactive way, which is important to prevent an adversary from targeting committee members. Elastico [48], Omniledger [40] and Chainspace [2] support sharding to further improve performance. Several systems also propose replacing proof-of-work with proof-of-stake protocols [8], [28], [38] to improve performance. While all of these systems provide higher performance than proof-of-work-based blockchains, unlike CCF, none of them provide strong confidentiality guarantees.

Some other blockchain designs either only provide confidentiality for limited scenarios or do not achieve high transaction throughput [5], [17], [41], [59], [73], [75]–[77].

Zerocash [59] provides privacy for financial transactions while Hawk [41] provides for generic privacy-preserving smart contracts. However, both of these works use complex cryptographic primitives which reduce performance. CCF provides confidentiality and throughput that is several orders of magnitude higher than these systems, albeit by trusting TEEs.

Several systems have proposed using TEEs to improve the efficiency of Byzantine fault tolerance protocols [7], [19], [20], [33], [35], [46], [65], [66]. Recently, TEEs have also been used in blockchains, but to achieve goals different from CCF's. REM [72] proposed a new blockchain mining framework that uses SGX enclaves, but still relies on inefficient proof of (useful) work for consensus. Hyperledger Sawtooth [74] proposes using a proof-of-elapsed-time (PoET) protocol based on running code that idles for a random amount of time inside an SGX enclave to elect a consensus leader, but does not provide any confidentiality guarantees. Teechan [47] proposes to implement payment channels that perform off-chain transactions in TEEs and only add a summary of transactions on Bitcoin. Proof of Luck [51] is a permissionless blockchain protocol which relies on Intel SGX for random leader election but otherwise uses a proof-of-work protocol [52]. Tesseract [9] uses a TEE to provide a cross-chain cryptocurrency exchange service. Town Crier [71] proposes using enclaves to authenticate data feeds from websites. All of these systems use TEEs to achieve desirable properties in a blockchain, but they do not provide confidentiality guarantees. Ekiden [18] runs smart contracts off-chain in stateless TEEs, which, after each execution, post encrypted state updates to an immutable ledger that provides *proof of publication*. The keys used for the encryption of state updates are managed by a permissionless network of key management TEEs. In a similar vein, Kaptchuk et al. [37] propose to persist TEE state on ledgers that provide proof of publication. To prevent rollbacks, they eliminate non-determinism and require that each query is committed to the blockchain in advance.

X. Conclusion

We presented the design and implementation of CCF, a new framework for consortium-based blockchains built on top of a network of hardware-protected TEEs. CCF enables confidential replicated services with dynamic reconfiguration and recovery. It accounts for a variety of attacker models, ranging from crashes to hardware compromise. It carefully restricts the privileges of TEEs that run its service and members that manage its consortium, and it records sufficient signed evidence to blame TEEs that deviate from its protocols. Experimental results show that CCF supports diverse blockchain applications and achieves performance orders of magnitude better than other blockchains with confidentiality guarantees. We are developing an open-source implementation of CCF to enable developers to build ledgers that achieve availability, confidentiality, verifiability, high performance, and flexible governance.

Acknowledgments. We thank Antoine Delignat-Lavaud, Samer Falah, Chris Klepper, Li Li, Zhen Li, Marc Mercuri, and Mark Novak for their contributions and their feedback on this report.

Appendix

References

[1] Visa inc. at a glance. https://usa.visa.com/dam/VCOM/download/corporate/media/visa-fact-sheet-Jun2015.pdf (Accessed on 11/30/2018).

[2] Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyny, D., and Danezis, G. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778* (2017).

[3] Alomari, M., Cahill, M., Fekete, A., and Rohm, U. The cost of serializability on platforms that use snapshot isolation.

[4] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative technology for CPU based attestation and sealing. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[5] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *European Conference on Computer Systems (EuroSys)* (2018).

[6] ARM Ltd. Building a secure system using TrustZone technology, April 2009. PRD29-GENC-009492C.

[7] Behl, J., Distler, T., and Kapitza, R. Hybrids on steroids: SGX-based high performance BFT. In *European Conference on Computer Systems (EuroSys)* (2017).

[8] Bentov, I., Gabizon, A., and Mizrahi, A. Cryptocurrencies without proof of work. In *Financial Cryptography and Data Security (FC)* (2016).

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

[10] Bhargavan, K., Bond, B., Delignat-Lavaud, A., Fournet, C., Hawblitzel, C., Hritcu, C., Ishtiaq, S., Kohlweiss, M., Leino, R., Lorch, J., et al. Everest: Towards a verified, drop-in replacement of HTTPS. *SNAPL* (2017).

[11] BitcoinWiki. Confirmation. https://en.bitcoin.it/wiki/Confirmation.

[12] Brasser, F., Capkun, S., Dmitrienko, A., Frassetto, T., Kostiainen, K., Müller, U., and Sadeghi, A.-R. DR.SGX: Hardening SGX enclaves against cache attacks with data location randomization. *arXiv preprint arXiv:1709.09917* (2017).

[13] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. Software Grand Exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies (WOOT)* (2017).

[14] Buchman, E. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, 2016.

[15] Bulck, J. V., Weichbrodt, N., Kapitza, R., Piessens, F., and Strackx, R. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security Symposium* (2017).

[16] Castro, M., and Liskov, B. Practical byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (1999).

[17] Cecchetti, E., Zhang, F., Ji, Y., Kosba, A., Juels, A., and Shi, E. Solidus: Confidential distributed ledger transactions via PVORM. *Cryptology ePrint Archive 317* (2017).

[18] Cheng, R., Zhang, F., Kos, J., He, W., Hynes, N., Johnson, N., Juels, A., Miller, A., and Song, D. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. *arXiv preprint arXiv:1804.05141* (2018).

[19] Chun, B.-G., Maniatis, P., Shenker, S., and Kubiatowicz, J. Attested append-only memory: Making adversaries stick to their word. In *ACM SIGOPS Operating Systems Review* (2007).

Name	Introduced in ...	Description
CERTCONFIG	Section III-D	Configuration of embedded certificate store
CERT	Section III-D	Certificates in use, including root certificates
CRL	Section III-D	Certificate revocation lists in use
MEMBER	Section III-A	Past and present members of the service
USER	Section III-A	Past and present users of the service
RULE	Section III-E	Configurable governance rules
PROPOSAL	Section III-E	Past and present member proposals
VOTE	Section III-E	Votes on member proposals
MEMBERACK	Section III-E	Signed ledger acknowledgements from members
CODEID	Section IV-B	Past and present enabled versions and digests of TEE code
SERVICE	Section IV-B	Past and present identity certificates of the service
NODE	Section IV-D	Past and present nodes of the service
SECRET	Section IV-D	Past and present data secrets (encrypted to the nodes)
SHARE	Section IV-G	Past and present recovery key shares (encrypted to the members)
REQUEST	Section V-B	Optional record of signed client request
RESPONSE	Section V-B	Optional record of service response
LOG	Section VI	Signed ledger `log` messages of the primary
ACK	Section VI	Signed ledger `ack` messages of the backups
VIEW	Section VI	Signed `view` messages

TABLE A.1
SUMMARY OF CONFIGURATION AND GOVERNANCE TABLES IN CCF

[20] Correia, M., Neves, N. F., and Verissimo, P. How to tolerate half less one Byzantine nodes in practical distributed systems. In *IEEE International Symposium on Reliable Distributed Systems* (2004).

[21] Costan, V., and Devadas, S. Intel SGX explained. *Cryptology ePrint Archive 86* (2016).

[22] Costan, V., Lebedev, I. A., and Devadas, S. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium* (2016).

[23] Danezis, G., and Meiklejohn, S. Centrally banked cryptocurrencies. In *Symposium on Network and Distributed System Security (NDSS)* (2016).

[24] Dinh, T. T. A., Wang, J., Chen, G., Liu, R., Ooi, B. C., and Tan, K.-L. BLOCKBENCH: A framework for analyzing private blockchains. In *ACM SIGMOD International Conference on Management of Data* (2017).

[25] Eyal, I., Gencer, A. E., Sirer, E. G., and van Renesse, R. Bitcoin-NG: A scalable blockchain protocol. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2016).

[26] Ferraiuolo, A., Baumann, A., Hawblitzel, C., and Parno, B. Komodo: Using verification to disentangle secure-enclave hardware from software. In *ACM Symposium on Operating Systems Principles (SOSP)* (2017).

[27] Fu, Y., Bauman, E., Quinonez, R., and Lin, Z. SGX-LAPD: thwarting controlled side channel attacks via enclave verifiable page faults. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2017).

[28] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *ACM Symposium on Operating Systems Principles (SOSP)* (2017).

[29] Goodman, L. M. Tezos - a self-amending crypto-ledger, 2014.

[30] Gorenflo, C., Lee, S., Golab, L., and Keshav, S. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. *arXiv preprint arXiv:1801.00910* (2019).

[31] Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., and Costa, M. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium* (2017).

[32] Gueta, G. G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M. K., Seredinschi, D.-A., Tamir, O., and Tomescu, A. SBFT: a scalable decentralized trust infrastructure for blockchains. *arXiv preprint arXiv:1804.01626* (2018).

[33] Jia, Y., Tople, S., Moataz, T., Gong, D., Saxena, P., and Liang, Z. Robust synchronous P2P primitives using SGX enclaves. *Cryptology ePrint Archive* (2017). https://eprint.iacr.org/2017/180.pdf.

[34] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. H-Store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow. 1*, 2 (2008).

[35] Kapitza, R., Behl, J., Cachin, C., Distler, T., Kuhnle, S., Mohammadi, S. V., Schröder-Preikschat, W., and Stengel, K. CheapBFT: resource-efficient Byzantine fault tolerance. In *European Conference on Computer Systems (EuroSys)* (2012).

[36] Kaplan, D., Powell, J., and Woller, T. AMD memory encryption, April 2016. white paper.

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

[38] Kiayias, A., Konstantinou, I., Russell, A., David, B., and Oliynykov, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. *Cryptology ePrint Archive 889* (2016).

[39] Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., and Ford, B. Enhancing bitcoin security and performance with strong consistency via collective signing. In *USENIX Security Symposium* (2016).

[40] Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., and Ford, B. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *IEEE Symposium on Security and Privacy (S&P)* (2018).

[41] Kosba, A., Miller, A., Shi, E., Wen, Z., and Papamanthou, C. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy (S&P)* (2016).

[42] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (1998).

[43] Lamport, L., Malkhi, D., and Zhou, L. Reconfiguring a state machine. *ACM SIGACT News 41*, 1 (2010).

[44] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4*, 3 (1982).

[45] 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 *USENIX Security Symposium* (2017).

[46] Levin, D., Douceur, J. R., Lorch, J. R., and Moscibroda, T. TrInc: Small trusted hardware for large distributed systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2009).

[47] Lind, J., Eyal, I., Pietzuch, P., and Sirer, E. G. Teechan: Payment channels using trusted execution environments. *arXiv preprint arXiv:1612.07766* (2016).

[48] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., and Saxena, P. A secure sharding protocol for open blockchains. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[49] Mckeen, F., Alexandrovich, I., Berenzon, A., Rozas, C., Shafi, H., Shanbhogue, V., and Savagaonkar, U. Innovative instructions and software model for isolated execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[50] Miller, A., Xia, Y., Croman, K., Shi, E., and Song, D. The honey badger of bft protocols. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[51] Milutinovic, M., He, W., Wu, H., and Kanwal, M. Proof of luck: An efficient blockchain consensus protocol. In *Workshop on System Software for Trusted Execution (SysTEX)* (2016).

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

[53] 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).

[54] Ongaro, D., and Ousterhout, J. K. In search of an understandable consensus algorithm. In *USENIX Anual Technical Conference* (2014).

[55] Orenbach, M., Lifshits, P., Minkin, M., and Silberstein, M. Eleos: ExitLess OS services for SGX enclaves. In *European Conference on Computer Systems (EuroSys)* (2017).

[56] Owusu, E., Guajardo, J., McCune, J., Newsome, J., Perrig, A., and Vasudevan, A. Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[57] Pass, R., and Shi, E. Hybrid consensus: Efficient consensus in the permissionless model. In *International Symposium on DIStributed Computing (DISC)* (2017).

[58] R.C.Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO* (1988).

[59] Sasson, E. B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., and Virza, M. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy (S&P)* (2014).

[60] Schneider, F. B. Timplementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22*, 4 (1990).

[61] 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 *IEEE Symposium on Security and Privacy (S&P)* (2015).

[62] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Symposium on Network and Distributed System Security (NDSS)* (2017).

[63] Sinha, R., Costa, M., Lal, A., Lopes, N., Seshia, S., Rajamani, S., and Vaswani, K. A design and verification methodology for secure isolated regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2016).

[64] Steindorfer, M. J., and Vinju, J. J. Optimizing hash-array mapped tries for fast and lean immutable JVM collections.

[65] Veronese, G. S., Correia, M., Bessani, A. N., and Lung, L. C. EBAWA: Efficient Byzantine agreement for wide-area networks. In *IEEE International Symposium on High-Assurance Systems Engineering (HASE)* (2010).

[66] Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C., and Verissimo, P. Efficient Byzantine fault-tolerance. *IEEE Transactions on Computers* (2013).

[67] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/Paper.pdf (accessed 16/10/2017). EIP-150 revision.

[68] Xiao, Y., Li, M., Chen, S., and Zhang, Y. Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In *ACM Conference on Computer and Communications Security (CCS)* (2017).

[69] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[70] Yang, D., Gavigan, J., and Wilcox-O'Hearn, Z. Survey of confidentiality and privacy preserving technologies for blockchains, 2016.

[71] Zhang, F., Cecchetti, E., Croman, K., Juels, A., and Shi, E. Town crier: An authenticated data feed for smart contracts. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[72] Zhang, F., Eyal, I., Escriva, R., Juels, A., and van Renesse, R. REM: Resource-efficient mining for blockchains. In *USENIX Security Symposium* (2017).

[73] Chain. https://chain.com.

[74] Hyperledger Sawtooth Core. https://github.com/hyperledger/sawtooth-core.

[75] Multichain. https://github.com/MultiChain/multichain.

[76] Quorum. https://github.com/jpmorganchase/quorum.

[77] Ripple. https://ripple.com.

[78] Sawtooth lake. https://sawtooth.hyperledger.org/docs/.