The Libra Blockchain

Zachary Amsden, Ramnik Arora, Shehar Bano, Mathieu Baudet, Sam Blackshear, Abhay Bothra, George Cabrera, Christian Catalini, Konstantinos Chalkias, Evan Cheng, Avery Ching, Andrey Chursin, George Danezis, Gerardo Di Giacomo, David L. Dill, Hui Ding, Nick Doudchenko, Victor Gao, Zhenhuan Gao, François Garillot, Michael Gorven, Philip Hayes, J. Mark Hou, Yuxuan Hu, Kevin Hurley, Kevin Lewi, Chunqi Li, Zekun Li, Dahlia Malkhi, Sonia Margulis, Ben Maurer, Payman Mohassel, Ladi de Naurois, Valeria Nikolaenko, Todd Nowacki, Oleksandr Orlov, Dmitri Perelman, Alistair Pott, Brett Proctor, Shaz Qadeer, Rain, Dario Russi, Bryan Schwab, Stephane Sezer, Alberto Sonnino, Herman Venter, Lei Wei, Nils Wernerfelt, Brandon Williams, Qinfan Wu, Xifan Yan, Tim Zakian, Runtian Zhou *

Note to readers: This report was published before the Association released White Paper v2.0, which includes a number of key updates to the Libra payment system. Outdated links have been removed, but otherwise, this report has not been modified to incorporate the updates and should be read in that context.

Abstract. The Libra Blockchain is a decentralized, programmable database designed to support a low-volatility cryptocurrency that will have the ability to serve as an efficient medium of exchange for billions of people around the world. We present a proposal for the Libra protocol, which implements the Libra Blockchain and aims to create a financial infrastructure that can foster innovation, lower barriers to entry, and improve access to financial services. To validate the design of the Libra protocol, we have built an open-source prototype implementation — *Libra Core* — in anticipation of a global collaborative effort to advance this new ecosystem.

The Libra protocol allows a set of replicas — referred to as validators — from different authorities to jointly maintain a database of programmable resources. These resources are owned by different user accounts authenticated by public key cryptography and adhere to custom rules specified by the developers of these resources. Validators process transactions and interact with each other to reach consensus on the state of the database. Transactions are based on predefined and, in future versions, user-defined smart contracts in a new programming language called *Move*.

We use Move to define the core mechanisms of the blockchain, such as the currency and validator membership. These core mechanisms enable the creation of a unique governance mechanism that builds on the stability and reputation of existing institutions in the early days but transitions to a fully open system over time.

1 Introduction

The spread of the internet and resulting digitization of products and services have increased efficiency, lowered barriers to entry, and reduced costs across most industries. This connectivity has driven economic empowerment by enabling more people to access the financial ecosystem. Despite this progress, access to financial services is still limited for those who need it most — impacted by cost, reliability, and the ability to seamlessly send money.

Revised May 1st, 2020

This paper presents a proposal for the Libra protocol, which supports the newly formed Libra ecosystem that seeks to address these challenges, expand access to capital, and serve as a platform for innovative financial services. This ecosystem will offer a new global currency — the Libra coin — which will be fully backed with a basket of bank deposits and treasuries from high-quality central banks. All of these currencies experience relatively low inflation, and thus the coin mechanically inherits this property as well as the advantages of a geographically diversified portfolio of assets. The Libra protocol must scale to support the transaction volume necessary for this currency to grow into a global financial infrastructure and provide the flexibility to implement the economic and governance policies that support its operations. The Libra protocol is designed from the ground up to holistically address these requirements and build on the learnings from existing projects and research — a combination of novel approaches and well-understood techniques.

A key prerequisite for healthy competition and innovation in financial services is the ability to rely on common infrastructure for processing transactions, maintaining accounts, and ensuring interoperability across services and organizations. By lowering barriers to entry and switching costs, the Libra protocol will enable startups and incumbents to compete on a level playing field, and experiment with new types of business models and financial applications. Blockchain technology lends itself well to address these issues because it can be used to ensure that no single entity has control over the ecosystem or can unilaterally shape its evolution to its advantage [1].

The Libra Blockchain will be decentralized, consisting of a collection of validators that work together to process transactions and maintain the state of the blockchain. These validators also form the membership of the Libra Association, which will provide a framework for the governance of the network and the reserve that backs the coin. Initially, the association (and validators) will consist of a geographically distributed and diverse set of Founding Members. These members are organizations chosen according to objective participation criteria, including that they have a stake in bootstrapping the Libra ecosystem and investing resources toward its success. Over time, membership eligibility will shift to become completely open and based only on the member's holdings of Libra. The association has published reports outlining its vision [2], its proposed structure [3], the coin's economics [4], and the roadmap for the shift toward a permissionless system [5].

This paper is the first step toward building a technical infrastructure to support the Libra ecosystem. We are publishing this early report to seek feedback from the community on the initial design, the plans for evolving the system, and the currently unresolved research challenges discussed in the proposal. Thus, the association has established an open-source community [6] for the discussion and development of the project.

The Libra protocol. The Libra Blockchain is a cryptographically authenticated database [7, 8, 9] maintained using the Libra protocol. The database stores a ledger of programmable resources, such as Libra coins. A resource adheres to custom rules specified by its declaring module, which is also stored in the database. A resource is owned by an account that is authenticated using public key cryptography. An account could represent direct end users of the system as well as entities, such as custodial wallets, that act on behalf of their users. An account's owner can sign transactions that operate on the resources held within the account.

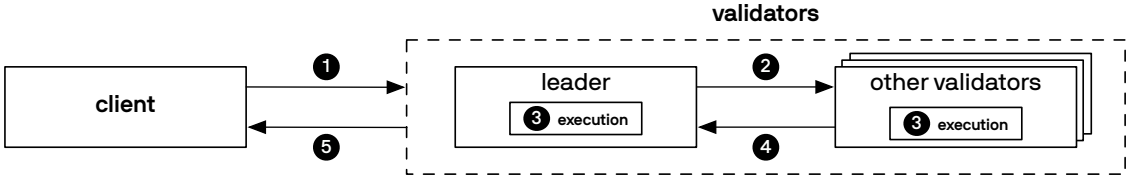

Figure 1: Overview of the Libra protocol.

Figure 1 shows the two types of entities that interact using the Libra protocol: (1) validators, which maintain the database and (2) clients, which perform queries on the database and submit transactions to modify it.

Validators maintain the database and process transactions submitted by clients for inclusion in the database (❶). The validators use a distributed consensus protocol to agree on an ever-growing list of transactions that have been committed to the database as well as the results of executing those transactions. This consensus protocol must be reliable even in the presence of malicious or erroneous behavior by a minority of validators. Validators take turns driving the process of accepting transactions. When a validator acts as a leader, it proposes transactions, both those directly submitted to it by clients and those indirectly submitted through other validators, to the other validators (❷). All validators execute the transactions (❸) and form an authenticated data structure that contains the new ledger history. The validators vote on the authenticator for this data structure as part of the consensus protocol (❹). As part of committing a transaction T_i at version i, the consensus protocol outputs a signature on the full state of the database at version i — including its entire history — to authenticate responses to queries from clients (❺).

Clients can issue queries to a validator to read data from the database. Since the database is authenticated, clients can be assured of the accuracy of the response to their query. As part of the response to a read query, a validator returns a signed authenticator for the latest version i of the database known to the validator.

In addition, a client can optionally create a replica of the entire database by synchronizing the transaction history from the validators. While creating a replica, a client can verify that validators executed transactions correctly, which increases accountability and transparency in the system. Other clients can read from a client that holds a replica in the same way they would read from a validator to verify the authenticity of the response. For the sake of simplicity, the rest of this paper assumes that clients query a validator directly rather than a replica.

Organization. This paper discusses the components of the Libra protocol:

- **Logical Data Model** (Section 2) describes the logical data model that organizes the decentralized database visible to validators and clients.
- **Executing Transactions** (Section 3) describes the use of Move [10], a new programming language that is used to define and execute database transactions.
- **Authenticated Data Structures and Storage** (Section 4) describes the mapping of the logical model into authenticated data structures based on Merkle trees [11].
- **Byzantine Fault Tolerant Consensus** (Section 5) describes the LibraBFT [12] variant of the HotStuff protocol [13], which allows a network with potentially malicious validators to maintain a single, consistent database by executing transactions with Move and coming to agreement on their execution using the authenticated data structures.
- **Networking** (Section 6) describes the protocol that enables validators to communicate with each other securely, as required for consensus.

Subsequently, we present the open-source Libra Core prototype [6]. Section 7 discusses how Libra Core combines the components of the Libra protocol to process a transaction. Section 8 discusses performance considerations.

Finally, we explain how the protocol is being used to support the economic stability and governance policies of the Libra ecosystem. Section 9 shows how we use the Move language to implement the low-volatility, reserve-backed Libra coin and a validator management system that mirrors the governance of the Libra Association.

Section 10 concludes the paper with a discussion of future plans and ongoing challenges for the Libra ecosystem.

2 Logical Data Model

All data in the Libra Blockchain is stored in a single *versioned* database [14, 15]. A version number is an unsigned 64-bit integer that corresponds to the number of transactions the system has executed. At each version i, the database contains a tuple (T_i, O_i, S_i) representing the transaction (T_i), transaction output (O_i), and ledger state (S_i). Given a deterministic execution function Apply, the meaning of the tuple is: executing transaction T_i against ledger state S_{i-1} produces output O_i and a new ledger state S_i; that is, $\mathsf{Apply}(S_{i-1}, T_i) \rightarrow \langle O_i, S_i \rangle$.

The Libra protocol uses the Move language to implement the deterministic execution function (see Section 3). In this section, we focus on the versioned database, which allows validators to:

1. Execute a transaction against the *ledger state* at the latest version.
2. Respond to client queries about the *ledger history* at both current and previous versions.

We first explain the structure of the ledger state stored in a single version and then discuss the purpose of the versioned ledger history view.

2.1 Ledger State

The ledger state represents the ground truth about the Libra ecosystem, including the quantity of Libra held by each user at a given version. Each validator must know the ledger state at the latest version in order to execute new transactions.

The Libra protocol uses an account-based data model [16] to encode the ledger state. The state is structured as a key-value store, which maps *account address* keys to *account values*. An account value in the ledger state is a collection of published Move *resources* and *modules*. The Move resources store data values and modules store code. The initial set of accounts and their state are specified in the genesis ledger state (see Section 3.1).

Account addresses. An account address is a 256-bit value. To create a new account, a user first generates a fresh verification/signature key-pair (vk, sk) for a signature scheme and uses the cryptographic hash of the public verification key vk as an account address $a = \mathsf{H}(vk)$.[1] The new account is created in the ledger state when a transaction sent from an existing account invokes the `create_account`(a) Move instruction. This typically happens when a transaction attempts to send Libra to an account at address a that has not yet been created.

Once the new account is created at a, the user can sign transactions to be sent from that account using the private signing key sk. The user can also rotate the key used to sign transactions from the account without changing its address, e.g., to proactively change the key or to respond to a possible compromise of the key.

The Libra protocol does not link accounts to a real-world identity. A user is free to create multiple accounts by generating multiple key-pairs. Accounts controlled by the same user have no inherent link to each other. This scheme follows the example of Bitcoin and Ethereum in that it provides pseudonymity [19] for users.

[1] Concretely we instantiate hash functions with SHA3-256 [17] and digital signatures with EdDSA using the edwards25519 curve [18].

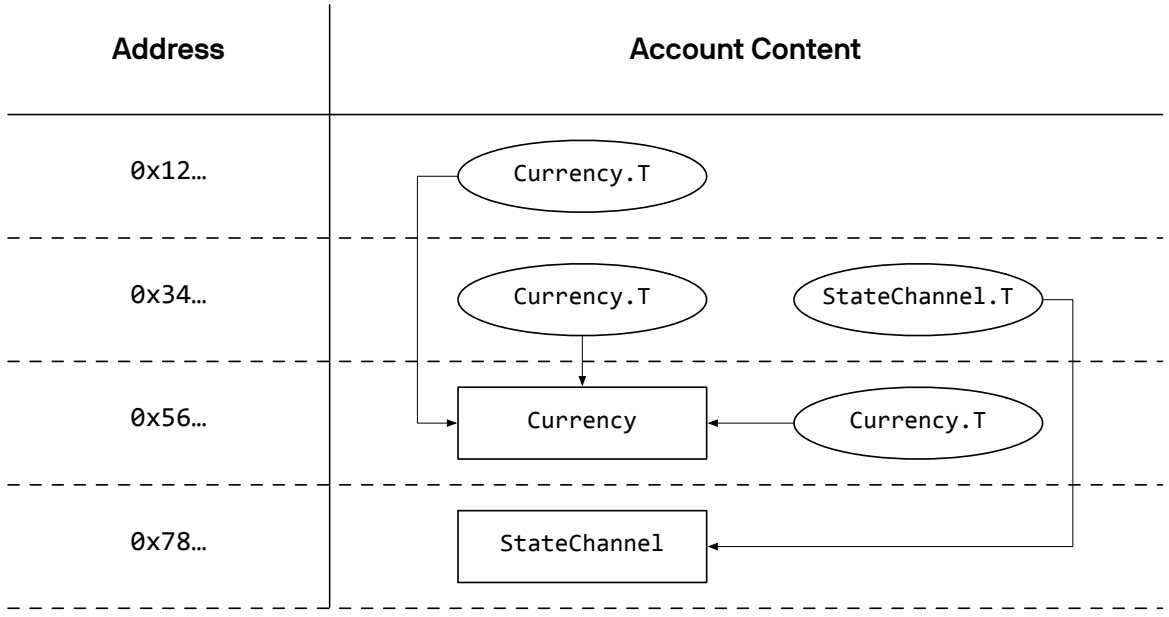

Figure 2: An example ledger state with four accounts. In this diagram, ovals represent resources and rectangles represent modules. A directed edge from a resource to a module means that the type of the resource was declared by that module. The account at address `0x12` contains a `Currency.T` resource declared by the `Currency` module. The code for the `Currency` module is stored at address `0x56`. The account at address `0x34` contains both a `Currency.T` resource and a `StateChannel.T` resource, which is declared by the module stored at address `0x78`.

Resource values. A resource value, or *resource*, is a record that binds named fields to simple values — such as integers — or complex values — such as other resources embedded inside this resource.

Every resource has a type declared by a module. Resource types are nominal types [20] that consist of the name of the type and the name and address of the resource's declaring module. For example, the type of the `Currency.T` resource in Figure 2 is `0x56.Currency.T`. Here, `0x56` is the address where the `Currency` module is stored, `Currency` is the name of the module, and `Currency.T` is the name of the resource.

To retrieve the resource `0x56.Currency.T` under account address `0x12`, a client would request `0x12/resources/0x56.Currency.T`. The purpose of this design is to let modules define a predictable schema for top-level account values — that is, every account stores its `0x56.Currency.T` resource under the same path. As such, each account can store at most one resource of a given type. However, this limitation is not restrictive, since programmers can define wrapper resources that organize resources in a custom way (e.g., `resource TwoCoin { c1: 0x56.Currency.T, c2: 0x56.Currency.T }`).

The rules for mutating, deleting, and publishing a resource are encoded in the module that created the resource and declared its type. Move's safety and verification rules prevent other code from making modifications to the resource.

Module values. A module value, or *module*, contains Move bytecode that declares resource types and procedures (see Section 3.4 for more details). Like a resource type, a module is identified by the address of the account where the module is declared. For example, the identifier for the `Currency` module in Figure 2 is `0x56.Currency`.

A module must be uniquely named within an account — each account can declare at most one module with a given name. For example, the account at address `0x56` in Figure 2 could not declare

another module named `Currency`. On the other hand, the account at address `0x34` *could* declare a module named `Currency`. The identifier of this module would be `0x34.Currency`. Note that `0x56.Currency.T` and `0x34.Currency.T` are distinct types and cannot be used interchangeably.

In the current version of the Libra protocol, modules are immutable. Once a module has been declared under an account address, it cannot be modified or deleted, except via a hard fork. We are researching options for a scheme to enable safe module updates in future versions.

2.2 Transactions

Clients of the Libra Blockchain update the ledger state by submitting transactions. At a high level, a transaction consists of a *transaction script* (written in Move bytecode) and arguments to the transaction script (e.g., a recipient account address or the number of Libra to send). A validator executes the transaction by running the script with its arguments and the current ledger state as input to produce a completely deterministic transaction output. The ledger state is not changed until the transaction is committed during consensus (Section 5) by agreeing on a binding commitment (Section 4) to the output of the transaction. We discuss the structure and execution of transactions in more detail in Section 3.

Transaction output. Executing a transaction T_i produces a new ledger state S_i as well as the execution status code, gas usage, and event list (aggregated in output O_i). The execution status code records the result of executing a transaction (e.g., success, exited with an error e, ran out of gas, etc.), and the gas usage records the number of gas units used in executing the transaction (see Section 3.1 for the details of how gas is used to manage the fee associated with transaction processing).

Events. The *event* list is a set of side effects produced by executing the transaction.[2] Move code can trigger an event emission through an event structure. Each event is associated with a unique key, which identifies the structure through which the event was emitted, and a payload, which provides detailed information about the event. Once a transaction has been committed by the consensus protocol, events generated by the transaction are added to the agreed ledger history and provide evidence that the successful execution of a transaction resulted in a specific effect. For example, a payment transaction emits an event that allows the recipient to confirm that a payment has been received and confirm the amount of the payment.

At first glance, events might seem redundant. Instead of querying for the events emitted by a transaction T_i, a client could ask whether the transaction T_i has been included in the blockchain. However, this is error-prone because the inclusion of T_i does not imply successful execution (e.g., it might be interrupted after running out of gas). In a system where transactions can fail, an event provides evidence not only that a particular transaction has executed but also that it had successfully completed with the intended effect.

Transactions can only generate events — they cannot *read* events. This design allows transaction execution to be a function only of the current state, not historical information, such as previously generated events.

[2] Events serve a purpose similar to the concept of logs and events in Ethereum [16]. However, the mechanics of events in the Libra protocol are quite different.

2.3 Ledger History

The ledger history stores the sequence of committed and executed transactions as well as the associated events they emitted. The purpose of the ledger history is to keep a record of how the latest ledger state was computed. There is no concept of a *block* of transactions in the ledger history.[3] The consensus protocol batches transactions into blocks as an optimization and to drive the consensus protocol (see Section 5 for details on consensus). However, in the logical data model, the transactions occur in sequence without distinction as to which block contained each transaction.

Although a validator does not need to know the ledger history to execute new transactions, the client can perform authenticated queries against the ledger history and use the ledger history for auditing the transaction execution.

Responding to client queries. Validators can use the ledger history to answer client queries about previous ledger states, transactions, and outputs. For example, a client might ask about the ledger state at a specific version (e.g., "What was the balance in the account at address x at version 30?") or the history of events of a certain type (e.g., "What payments did the account at address y receive in the last 20 minutes?").

Auditing transaction execution. A client can check that the ledger state is correct by re-executing each transaction T_i in the history and comparing the computed ledger state to the corresponding ledger state S_i and transaction output O_i in the versioned database. This mechanism allows clients to audit the validators to ensure that transactions are being executed correctly.

3 Executing Transactions

In the Libra protocol, the only way to change the blockchain state is by executing a transaction. This section outlines the requirements for transaction execution, defines the structure of transactions, explains how the Move virtual machine (VM) executes a transaction, and describes the key concepts of the Move language.

In the initial version of the Libra protocol, only a limited subset of Move's functionality is available to users. While Move is used to define core system concepts, such as the Libra currency, users are unable to publish custom modules that declare their own resource types. This approach allows the Move language and toolchain to mature — informed by the experience in implementing the core system components — before being exposed to users. The approach also defers scalability challenges in transaction execution and data storage that are inherent to a general-purpose smart contract platform. As we discuss in Section 10, we are committed to exposing the full programmability supported by the Move language.

3.1 Execution Requirements

Known initial state. All validators must agree on the initial, or *genesis*, ledger state of the system. Because the core components of the blockchain — such as the logic of accounts, transaction validation, validator selection, and Libra coins — are defined as Move modules, the genesis state must define these modules. The genesis state must also have sufficient instantiations of these core components so that transactions can be processed (e.g., at least one account must be able to pay fees for the

[3] This is in contrast with Bitcoin and Ethereum, in which the concept of a block and the maximum size of a block play important roles.

first transaction; a validator set must be defined so a quorum of the set can sign the authenticator committing to the first transaction).

To simplify the design of the system, the initial state of the system is represented as an empty state. The genesis state is then created through a special transaction T_0 that defines specific modules and resources to be created, rather than going through the normal transaction process. Clients and validators are configured to accept only ledger histories beginning with a specific T_0, which is identified by its cryptographic hash. These special transactions cannot be added to the ledger history through the consensus protocol, only through configuration.[4]

Deterministic. Transaction execution must be *deterministic* and *hermetic*. This means that the output of transaction execution is completely predictable and based only on the information contained within the transaction and current ledger state. Transaction execution does not have external effects (e.g., printing to the console or interacting with the network). Deterministic and hermetic execution ensures that multiple validators can agree on the state resulting from the same sequence of transactions even though transactions are executed independently by each validator. It also means that the transaction history of the blockchain can be re-executed from the genesis block onwards to produce the current ledger state (see Section 2.3).

Metered. In order to manage demand for compute capacity, the Libra protocol charges transaction fees, denominated in Libra coins.[5] This follows the *gas* model popularized by Ethereum [16]. We take the approach of selecting validators with sufficient capacity to meet the needs of the Libra ecosystem (see Section 8). The only intention of this fee is to reduce demand when the system is under a higher load than it was provisioned for (e.g., due to a denial-of-service attack). The system is designed to have low fees during normal operation, when sufficient capacity is available. This approach differs from some existing blockchains, which target validators with lower capacity and thus at times have more demand to process transactions than throughput. In these systems, fees spike during periods of high demand — representing a revenue source for the validators but a cost for the users.

The size of the fee is determined by two factors: gas price and gas cost. Each transaction specifies a price in Libra per unit of gas that the submitter is willing to pay. The execution of a transaction dynamically accounts for the computational power it expends in the form of a gas cost. Validators prioritize executing transactions with higher gas prices and may drop transactions with low prices when the system is congested. This reduces the demand for transactions under high load.

In addition, a transaction includes a maximum gas amount, which specifies the maximum number of gas units that the submitter is willing to pay for at the specified price. During execution, the VM tracks the number of gas units used. If the maximum gas limit is reached before execution completes, the VM halts immediately. None of the partial changes resulting from such a transaction are committed to the state. However, the transaction still appears in the transaction history, and the sender account is charged for the gas used.

As we discuss in this section, many parts of the core logic of the blockchain are defined using Move, including the deduction of gas fees. To avoid circularity, the VM disables the metering of gas during the execution of these core components. These core components must be defined in the genesis state and must be written defensively to prevent malicious transactions from triggering the execution of

[4] We are exploring the idea of using special transactions of this form to define hard forks cleanly. Clients could specify a preferred configuration, stating that a similar special transaction — which makes specific changes to modules and resources outside the standard transaction processing rules — should be appended to the ledger history at version i. Since many parts of the Libra protocol are expressed using Move, updates of this form can be expressive yet would only require a configuration change to client software.

[5] In Section 4.4, we discuss approaches to managing demand for storage capacity.

computationally expensive code. The cost of executing this logic can be included in the base fee charged for the transaction.

Asset semantics. As we explain in Section 2.1, the ledger state directly encodes digital assets with real-world value. Transaction execution must ensure that assets such as Libra coins are not duplicated, lost, or transferred without authorization. The Libra protocol uses the Move virtual machine (Section 3.4) to implement transactions and custom assets with these properties safely.

3.2 Transaction Structure

A transaction is a signed message containing the following data:

- **Sender address:** The account address of the transaction sender. The VM reads the sequence number, authentication key, and balance from the `LibraAccount.T` resource stored under this address.
- **Sender public key:** The public key that corresponds to the private key used to sign the transaction. The hash of this public key must match the authentication key stored under the sender's `LibraAccount.T` resource.
- **Program:** A Move bytecode transaction script to execute, an optional list of inputs to the script, and an optional list of Move bytecode modules to publish.
- **Gas price:** The number of Libra coins that the sender is willing to pay per unit of gas in order to execute this transaction.
- **Maximum gas amount:** The maximum number of gas units that the transaction is allowed to consume before halting.
- **Sequence number:** An unsigned integer that must be equal to the sequence number from the sender's `LibraAccount.T` resource. After this transaction executes, the sequence number is incremented by one. Since only one transaction can be committed for a given sequence number, transactions cannot be replayed.

3.3 Executing Transactions

Executing a transaction proceeds through a sequence of six steps inside the VM. Execution is separate from the update of the ledger state. First, a transaction is executed as part of an attempt to reach agreement on its sequencing. Since the execution is hermetic, this can be done without causing external side effects. Subsequently, if agreement is reached, its output is written to the ledger history. Executing a transaction performs the following six steps:

(1) Check signature. The signature on the transaction must match the sender's public key and the transaction data. This step is a function only of the transaction itself — it does not require reading any data from the sender's account.

(2) Run prologue. The prologue authenticates the transaction sender, ensures that the sender has sufficient Libra coin to pay for the maximum number of gas units specified in the transaction, and checks that the transaction is not a replay of a previous transaction. All of these checks are implemented in Move via the `prologue` procedure of the `LibraAccount` module. Gas metering is disabled during the execution of the prologue. Specifically, the prologue does the following:

- Checks that the hash of the sender's public key is equal to the authentication key stored under the sender's account. Without this check, the VM would erroneously accept a transaction with

a cryptographically valid signature even though there is no correspondence to the key that is associated with the account.
- Checks that `gas_price * max_gas_amount <= sender_account_balance`. Without this check, the VM would execute transactions that may fail in the epilogue because they would be unable to pay for gas.
- Ensure that the transaction sequence number is equal to the sequence number stored under the user's account. Without this check, an adversary could replay old transactions (e.g., Bob could replay a transaction from Alice that paid him ten Libra coins).

(3) Verify transaction script and modules. Once the transaction prologue has completed successfully, the VM performs well-formedness checks on the transaction script and modules using the Move bytecode verifier. Before actually running or publishing any Move code, the bytecode verifier checks crucial properties like type-safety, reference-safety (i.e., no dangling references), and *resource-safety* (i.e., resources are not duplicated, reused, or inadvertently destroyed).

(4) Publish modules. Each module in the program field of the transaction is published under the transaction sender's account. Duplicate module names are prohibited — for example, if the transaction attempts to publish a module named `M` to an account that already contains a module named `M`, the step will fail.

(5) Run transaction script. The VM binds the transaction arguments to the formal parameters of the transaction script and executes it. If this script execution completes successfully, the write operations performed by the script and the events emitted by the script are committed to the global state. If the script execution fails (e.g., due to having run out of gas or a runtime execution failure), no changes from the script are committed to the global state.

(6) Run epilogue. Finally, the VM runs the transaction epilogue to charge the user for the gas used and increment the sender's account sequence number. Like the prologue, the transaction epilogue is a procedure of the Move `LibraAccount` module and runs with gas metering disabled. The epilogue is always run if execution advances beyond step (2), including when steps (3), (4), or (5) fail. The prologue and the epilogue work together to ensure that all transactions accepted in the ledger history are charged for gas. Transactions that do not proceed beyond step (2) are not appended to the ledger history. The fact that these transactions were considered for execution is never recorded. If a transaction advances past step (2), the prologue has ensured that the account has enough Libra coins to pay for the maximum number of gas units allowed for the transaction. Even if the transaction runs out of gas, the epilogue is able to charge it for this maximum amount.

3.4 The Move Programming Language

As we have seen, a transaction is an authenticated wrapper around a Move bytecode program. Move [10] is a new programming language created during the design of the Libra protocol. Move has three important roles in the system:

1. To enable flexible transactions via *transaction scripts*.
2. To allow user-defined code and datatypes, including "smart contracts" via *modules*.
3. To support configuration and extensibility of the Libra protocol (see Section 9).

The key feature of Move is the ability to define custom *resource types*, which have semantics inspired by linear logic [21]. Resource types are used to encode programmable assets that behave like ordinary program values: resources can be stored in data structures, passed as arguments to procedures, and so on. However, the Move type system provides special safety guarantees for resources. A resource

can never be copied, only *moved*. In addition, a resource type can only be created or destroyed by the module that declares the type. These guarantees are enforced statically by the Move VM. This allows us to represent Libra coins as a resource type in the Move language (in contrast to Ether and Bitcoin, which have a special status in their respective languages).

We have published a separate, more detailed report on the design of Move [10]. In the remainder of this section, we give a brief overview of the key Move concepts for transaction execution: developing Move transaction scripts and modules and executing Move bytecode with the virtual machine.

Writing Move programs. There are three different representations of a Move program: source code, intermediate representation (IR), and bytecode. We are currently in the process of designing the Move source language, which will be an ergonomic language designed to make it easy to write and verify safe code. In the meantime, programmers can develop modules and transaction scripts in Move IR. Move IR is high level enough to write human-readable code, yet low level enough to directly translate to Move bytecode. Both the future Move source language and the Move IR are compiled into Move bytecode, which is the format used by the Libra protocol.

We use a verifiable bytecode as the executable representation of Move for two reasons:

- The safety guarantees described above must apply to *all* Move programs. Enforcing these guarantees in the compiler is not enough. An adversary could always choose to bypass the compiler by writing malicious code directly using the bytecode (running a compiler as part of transaction execution would make execution slower and more complex and would require validators trusting the correctness of the full compiler code base). Thus, the protocol avoids trusting the compiler by enforcing all of Move's safety guarantees via bytecode verification: type-safety, reference-safety, and resource-safety.
- Move's stack-based bytecode has fewer instructions than a higher-level source language would. In addition, each instruction has simple semantics that can be expressed via an even smaller number of atomic steps. This reduces the specification footprint of the Libra protocol and makes it easier to spot implementation mistakes.

Transaction scripts. A transaction script is the `main` procedure of the Libra protocol. Transaction scripts enable flexible transactions because a script is an arbitrary Move bytecode program. A script can invoke multiple procedures of modules published in the ledger state, use conditional logic, and perform local computation. This means that scripts can perform expressive one-off actions, such as paying a specific set of recipients.

We expect that most transaction scripts will perform a single procedure call that wraps generic functionality. For example, the `LibraAccount.pay_from_sender(recipient_address, amount)` procedure encapsulates the logic for performing a peer-to-peer payment. A transaction script that takes `recipient_address` and `amount` as arguments and invokes this procedure is the same as an Ether transfer transaction in Ethereum [16].

Modules. A module is a code unit published in the ledger state. A module declares both struct types and procedures. A struct value contains data fields that may hold primitive values, such as integers or other struct values.

Each struct must be tagged as either resource or *unrestricted* (i.e., non-resource). Unrestricted structs are not subject to the restrictions on copying and destruction described above. However, unrestricted structs cannot contain resource structs (either directly or transitively) and cannot be published under an account in the ledger state.

At a high level, the module/struct/procedure relationship is similar to the class/object/method relationship in object-oriented programming. However, there are important differences. A module may

declare multiple struct types (or zero struct types). No data field can be accessed outside of its declaring module (i.e., no `public` fields). The procedures of a module are static procedures rather than instance methods; there is no concept of `this` or `self` in a procedure. Move modules are similar to a limited version of ML-style modules [22] without higher-order functions.

Move modules are related to, but not the same as, the concept of "smart contracts" in Ethereum and other blockchain platforms. An Ethereum smart contract contains both code and data published in the ledger state. In Libra, modules contain code values, and resources contain data values. In object-oriented terms, an Ethereum smart contract is like a singleton object published under a single account address. A module is a recipe for creating resources, but it can create an arbitrary number of resources that can be published under different account addresses.

The Move virtual machine. The Move virtual machine implements a verifier and an interpreter for the Move bytecode. The bytecode targets a stack-based virtual machine with a procedure-local operand stack and registers. Unstructured control-flow is encoded via gotos and labels.

A developer writes a transaction script or module in Move IR that is then compiled into the Move bytecode. Compilation converts structured control-flow constructs (e.g., conditionals, loops) into unstructured control-flow and converts complex expressions into a small number of bytecode instructions that manipulate an operand stack. The Move VM executes a transaction by verifying then running this bytecode.

The Move VM supports a small number of types and values: booleans, unsigned 64-bit integers, 256-bit addresses, fixed-size byte arrays, structs (including resources), and references. Struct fields cannot be reference types, which prevents the storage of references in the ledger state.

The Move VM does not have a heap — local data is allocated on the stack and freed when the allocating procedure returns. All persistent data must be stored in the ledger state.

4 Authenticated Data Structures and Storage

After executing a transaction, a validator translates the changes to the logical data model into a new version of an authenticated data structure [7, 8, 9] used to represent the database. The short authenticator of this data structure is a binding commitment to a ledger history, which includes the newly executed transaction.

Like transaction execution, the generation of this data structure is deterministic. The consensus protocol uses this authenticator to agree on an ordering of transactions and their resulting execution (we discuss consensus in detail in Section 5). As part of committing a block of transactions, validators collectively sign the short authenticator to the new version of the resulting database.

Using this collective signature, clients can trust that a database version represents the full, valid, and irreversible state of the database's ledger history. Clients can query any validator (or a third-party replica of the database) to read a specific database value and verify the result using the authenticator and a short proof. Therefore, clients do not need to trust the party that executes the query for the correctness of the resulting read.

Data structures in the Libra protocol are based on Merkle trees and inspired by those of other blockchains; however, in several cases, we have made slightly different decisions which we highlight below. First, we briefly discuss the Merkle tree approach to creating an authenticator. We then describe the authenticated data structure, starting from the root of the data structure, then we discuss the substructures within it. Figure 3 depicts the data structure and provides a visual guide to this section.

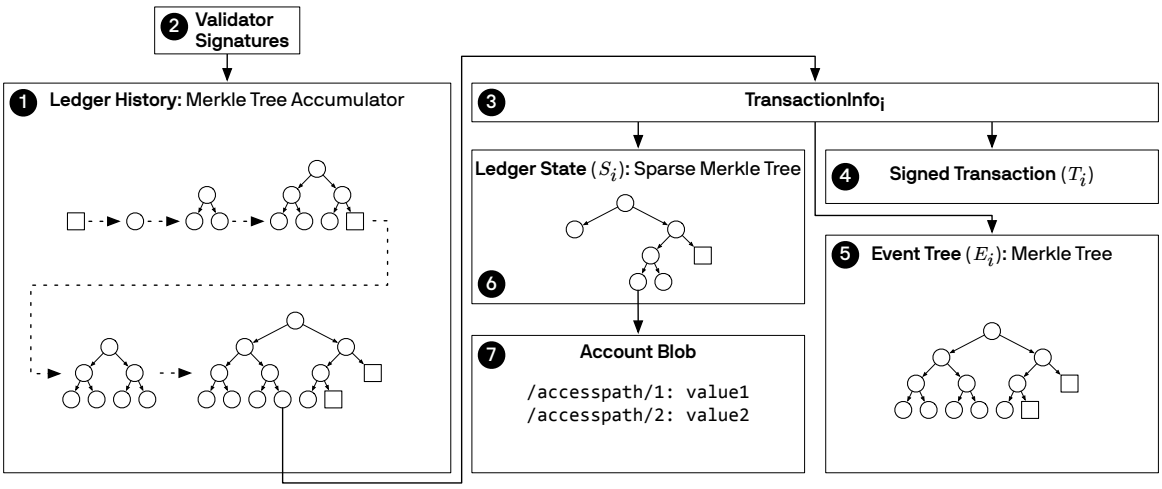

Figure 3: (❶) The root hash of the ledger history structure is the authenticator to the full state of the system that is (❷) signed by a quorum of validators. As transactions are added to the database, the authenticator (Section 4.2) committing to the ledger history grows (depicted with dashed arrows). Each leaf of the ledger history commits to a (❸) TransactionInfo structure. This structure commits to the (❹) signed transaction (T_i), (❺) the list of events generated during that transaction (E_i, Section 4.5), and the (❻) state after the execution of that transaction (S_i, Section 4.3). The state is a sparse Merkle tree with an (❼) account blob at each leaf.

4.1 Background on Authenticated Data Structures

An authenticated data structure allows a verifier V to hold a short authenticator a, which forms a binding commitment to a larger data structure D. An untrusted prover P, which holds D, computes $f(D) \to r$ and returns both r — the result of the computation of some function f on D — as well as π — a proof of the correct computation of the result — to the verifier. V can run $\mathsf{Verify}(a, f, r, \pi)$, which returns true if and only if $f(D) = r$. In the context of the Libra Blockchain, provers are generally validators and verifiers are clients executing read queries. However, clients — even those with only a partial copy of the database — can also serve as a prover and perform authenticated read queries for other clients.

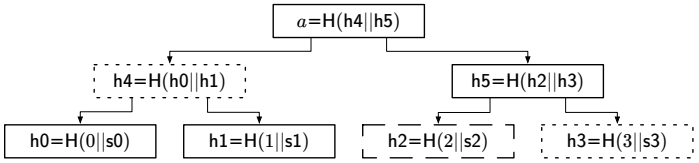

Figure 4: A Merkle tree storing $D = \{0 : \mathsf{s0}, \ldots\}$. If f is a function that gets the third item (shown with a dashed line) then $r = \mathsf{s2}$ and $\pi = [\mathsf{h3}, \mathsf{h4}]$ (these nodes are shown with a dotted line). $\mathsf{Verify}(a, f, r, \pi)$ verifies that $a \stackrel{?}{=} \mathsf{H}(\mathsf{h4} \| \mathsf{H}(\mathsf{H}(2 \| r) \| \mathsf{h3}))$.

Merkle trees [11] are a common form of authenticated data structure, used to store maps between integers and string values. In a Merkle tree of size 2^k, the structure D maps every integer key $i \in [0, 2^k)$ to a string value s_i. The authenticator is formed from the root of a full binary tree created from the strings, labeling leaves as $\mathsf{H}(i \| s_i)$ and internal nodes as $\mathsf{H}(\text{left} \| \text{right})$, where H is a cryptographic hash function (which we will refer to as a hash).[6] The function f, which the prover

[6] Secure Merkle trees must use different hash functions to hash the leaves and internal nodes to avoid confusion between the two types of nodes. While we have omitted this detail in the example for simplicity, the Libra protocol uses a unique hash function to distinguish between different hash function types to avoid attacks based on type confusion [23].

wishes to authenticate, is an inclusion proof that a key-value pair (k, v) is within the map D.

P authenticates lookups for an item i in D by returning a proof π that consists of the labels of the sibling of each of the ancestors of node i. Figure 4 shows a lookup for item three in a Merkle tree of size four. Item three's node is shown with a dotted line, and the nodes included in π are shown with a dashed line.

4.2 Ledger History

Most blockchains, starting with Bitcoin [24], maintain a linked list of each block of transactions agreed on by the consensus protocol with a block containing the hash of a single ancestor. This structure leads to inefficiencies for clients. For example, a client that trusts some block B and wants to verify information in an ancestor block B' needs to fetch and process all intermediate ancestors.

The Libra protocol uses a single Merkle tree to provide an authenticated data structure for the ledger history. Figure 3 illustrates the full data structure that underpins the Libra database, including the ledger history that contains $\mathsf{TransactionInfo}_i$ records, which, in turn, contain information about database states, events, and accounts. The ledger history is represented as a Merkle tree, mapping a sequential database version number i to a $\mathsf{TransactionInfo}_i$ structure. Each $\mathsf{TransactionInfo}_i$ structure contains a signed transaction (T_i), the authenticator for the state after the execution of T_i (S_i, discussed in Section 4.3), and the authenticator for the events generated by T_i (E_i, discussed in Section 4.5). Like other Merkle trees, the ledger history supports $O(\log n)$-sized proofs — where n is the total number of transactions processed — to authenticate the lookup of a specific $\mathsf{TransactionInfo}_i$. When a client wishes to query the state of version i or lookup an event generated in version i, it performs an authenticated lookup of $\mathsf{TransactionInfo}_i$ along with an authenticated lookup using the contained state or event list authenticator.

Specifically, the ledger history uses the *Merkle tree accumulator* [25, 26] approach to form Merkle trees, which also provides efficient append operations. This operation is useful, as a ledger history can be incrementally computed by appending new transactions to an old ledger history. Merkle tree accumulators can also generate an efficient proof that, given an authenticator a committing to the ledger info up to transaction i, an authenticator a' committing to the ledger info up to $j > i$ has an identical history up to transaction i — in other words, proving that a is a prefix of a'. These proofs allow efficient verification that one ledger history commitment is a continuation of another.

Pruning Storage. The root hash of the ledger history Merkle accumulator is an authenticator for the full state of the system. It commits to the state at every existing version of the system, every transaction ever sent, and every event ever generated. While the state storage described in Section 4.3 allows efficient storage of multiple versions of the ledger state, validators may wish to reduce the space consumed by past versions. Validators are free to prune old states as they are not necessary for the processing of new transactions. Merkle tree accumulators support the pruning of historical data, requiring only $O(\log n)$ storage to append new records [25].

Any replica of the system is free to retain the entire state and can trace the authenticity of their data to the root hash signed by the consensus protocol. Replicas are more amenable to scaling than validators. Replicas do not need to be secured as they do not participate in consensus, and multiple replicas can be created to handle read queries in parallel.

4.3 Ledger State

A ledger state S_i represents the state of all accounts at version i as a map of key-value pairs. Keys are based on the 256-bit account addresses, and their corresponding value is the authenticator of the account (discussed in Section 4.4).[7] Using a representation similar to Figure 4, the map is represented as a Merkle tree of size 2^{256}.

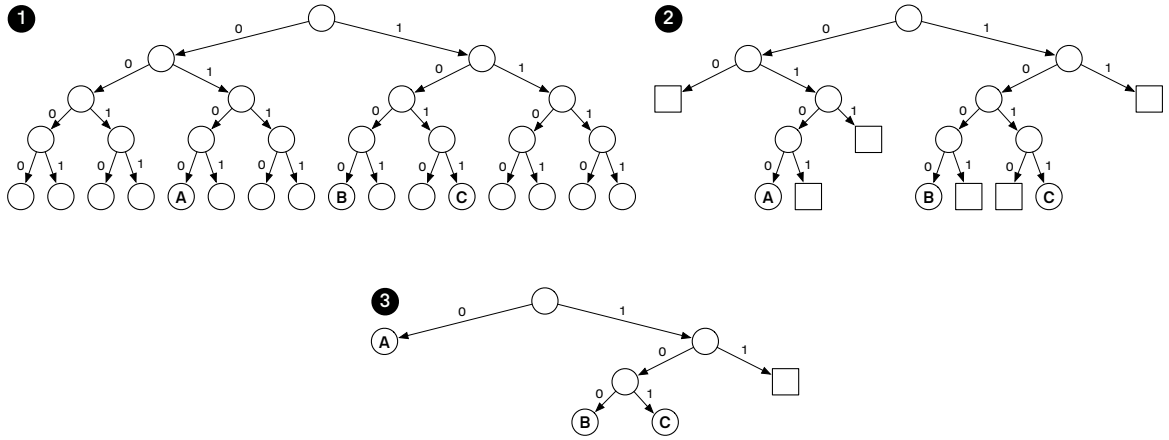

Figure 5: Three versions of a sparse Merkle tree storing $D = \{0100_2 : \mathsf{A}, 1000_2 : \mathsf{B}, 1011_2 : \mathsf{C}\}$.

While a tree of size 2^{256} is an intractable representation, optimizations can be applied to form a sparse Merkle tree. Figure 5 shows two optimizations that are applied to transform a naive implementation (❶) into an efficient one. First, subtrees that consist entirely of empty nodes are replaced with a placeholder value (❷) as used in the certificate transparency system [27]. This optimization creates a representation of a tractable size without substantially changing the proof generation of the Merkle tree. However, leaves are always stored at the bottom level of the tree, meaning that the structure requires 256 hashes to be computed on every leaf modification. A second optimization replaces subtrees consisting of exactly one leaf with a single node (❸). In expectation, the depth of any given item is $O(\log n)$, where n is the number of items in the tree. This optimization reduces the number of hashes to be computed when performing operations on the map.

Efficient implementations, such as Libra Core, can optimize their disk layout and batch layers of nodes to avoid seeks during lookup.[8]

When the sparse Merkle tree is updated after the execution of a transaction, the new tree reuses unchanged portions of the previous version, forming a *persistent data structure* [28, 29]. If a transaction modifies m accounts out of n accounts in the system, on average $O(m \cdot \log n)$ new branches and leaves are created in the new ledger state tree that differ from the previous version. This approach allows validators to store multiple versions of the ledger state efficiently. This feature also allows the efficient recomputation of the ledger state authenticator after processing a transaction.

We considered AVL-based trees, which provide more optimal worst-case proof length than sparse Merkle trees [30]. However, the sparse Merkle tree approach allows implementers to make optimizations, such as sharding storage across servers and parallelizing root hash computation.

[7] The hash of the address is used as the key. Even though addresses are hashes of the public key, an adversary could create a new account at an address that is a near collision with a real address. While no transactions can be sent from this account because the private key corresponding to its address is unknown, the existence of this near-collision in the tree increases the proof length for the nearby address. Hashing the address before using it as a key reduces the impact of this type of attack.

[8] This optimization gives similar on-disk performance to the 16-ary Patricia Merkle tree approach used in Ethereum but with shorter proofs.

4.4 Accounts

At the logical level, an account is a collection of resources and modules stored under the account address. At the physical level, an account is treated as an ordered map of *access paths* to byte array values. An access path is a delimited string similar to a path in a file system.

In the first iteration of the protocol, we serialize an account as a list of access paths and values sorted by access path. The authenticator of an account is the hash of this serialized representation. Note that this representation requires recomputing the authenticator over the full account after any modification to the account. The cost of this operation is $O(n)$, where n is the length of the byte representation of the full account. Furthermore, reads from clients require the full account information to authenticate any specific value within it.

Our strategy for storing data within accounts differs from other systems such as Ethereum.[9] Our approach allows the Move language to provide better programming abstractions by representing accounts as ordered maps of access paths to values. This representation allows Move to efficiently manage the conservation of resources through the VM. Move encourages each user to hold resources in their own account. In the initial release of Libra, we optimize for small accounts. In future releases, we may consider supporting more efficient structures to represent larger accounts, such as Merkle AVL trees [30].

Account Eviction and Recaching. We anticipate that as the system is used, eventually storage growth associated with accounts may become a problem. Just as gas encourages responsible use of computation resources (see Section 3.1), we expect that a similar rent-based mechanism may be needed for storage. We are assessing a wide range of approaches for a rent-based mechanism that best suits the ecosystem. We discuss one option that can be applied to any policy that determines an expiration time after which data can be evicted.

After each transaction execution affecting an account, the VM computes the logical expiration time at which the storage allocated for an account can be freed. The VM is free to apply any deterministic approach to determine the expiration time. One example of such a policy would be to charge a fee denominated in Libra coins that is based on the amount of time the account is stored and its size. The account's authenticator is represented as $\mathsf{H}(\mathsf{H}(\mathsf{AccountBlob})\|\mathsf{ExpirationTime})$, which encodes the expiration time for the account. After expiration, the VM denies access to the account, raising an error. Since the AccountBlob is inaccessible, a validator is free to prune this data after ExpirationTime. However, a validator allows a transaction to reactivate the account by recaching its contents after the expiration time. The transaction must contain the preimage of the AccountBlob and have an associated transaction fee that covers the cost of further storage. The authenticity of the recached contents is ensured through recomputing and checking the hash value associated with the account, which is not deleted. This method to implement rent is an improvement on existing account eviction schemes, which require a third party to send a transaction to delete the storage of the account.

4.5 Events

E_i is the list of events emitted during the execution of T_i. Each event is stored as a leaf in a Merkle tree that forms an authenticator for E_i. An event is serialized as a tuple of the form (A, p, c), which represents the access path of the event structure that emitted the event (A), the data payload (p), and the counter value (c). These tuples are indexed by the order j in which the events were emitted

[9] In contrast, Ethereum [16] stores data as a sparse memory map using a Merkle tree that represents an unordered map of 256-bit keys to 256-bit values. This approach allows Ethereum to handle large accounts efficiently. This design matches the standard practice within that ecosystem, which has accounts that manage assets on behalf of many users.

during the execution of T_i. By convention, we denote an event $j \rightarrow (A, p, c)$ being included in E_i as $(j, (A, p, c)) \in E_i$. The authenticator for E_i is included in $\mathsf{TransactionInfo}_i$. Thus, a validator can construct an inclusion proof that within the i^{th} transaction the j^{th} event was (A, p, c).

The counter c, included in each event, plays a special role in allowing clients to retrieve a list of events for a given access path A. Clients can also be assured that the list is complete. An event structure representing events with access path A within the Move language maintains a counter C for the total number of events it has emitted. This counter is stored within the ledger state and incremented after each transaction execution emits an event on access path A.

A client can obtain an authenticator for a recent ledger state, query the counter associated with the event structure for access path A, and retrieve the total number of events C. Then the client can query an untrusted service for the list of events on access path A. The query response consists of a set of tuples (i, j, A, p, c) — one corresponding to each event — where i is the transaction sequence number emitting the event and j is the order of the event emission within this transaction. Associated proofs of inclusion for $(j, (A, p, c)) \in E_i$ can be provided for each event. Since the client knows the total number of events emitted by access path A, they can ensure that the untrusted service has provided this number of distinct events and also order them by their sequence number $0 \leq c < C$. This convinces the client that the list of events returned for A is complete.

This scheme allows the client to hold an authenticated subscription to events on access path A. The client can periodically poll the total counter C for event access path A to determine if the subscription is up-to-date. For example, a client can use this to maintain a subscription to incoming payment transactions on an address it is watching. Untrusted third-party services can provide feeds for events indexed by access path, and clients can efficiently verify the results they return.

5 Byzantine Fault Tolerant Consensus

The consensus protocol allows a set of validators to create the logical appearance of a single database. The consensus protocol replicates submitted transactions among the validators, executes potential transactions against the current database, and then agrees on a binding commitment to the ordering of transactions and resulting execution. As a result, all validators can maintain an identical database for a given version number following the state machine replication paradigm [31]. The Libra Blockchain uses a variant of the HotStuff [13] consensus protocol called LibraBFT. It provides safety and liveness in the partial synchrony model in the tradition of DLS [32] and PBFT [33] as well as the newer Casper [34] and Tendermint [35]. This section outlines the key decisions in LibraBFT. A more detailed analysis, including proofs of safety and liveness, is covered in the full report on LibraBFT [12].

Agreement on the database state must be reached between validators, even if there are Byzantine faults [36]. The Byzantine failures model allows some validators to arbitrarily deviate from the protocol without constraint, except for being computationally bounded (and thus not able to break cryptographic assumptions). Byzantine faults are worst-case errors where validators collude and behave maliciously to try to sabotage system behavior. A consensus protocol that tolerates Byzantine faults caused by malicious or hacked validators can also mitigate arbitrary hardware and software failures.

LibraBFT assumes that a set of $3f + 1$ votes is distributed among a set of validators that may be honest, or Byzantine. LibraBFT remains safe, preventing attacks such as double spends and forks when at most f votes are controlled by Byzantine validators. LibraBFT remains live — committing transactions from clients — as long as there exists a global stabilization time (GST), after which all messages between honest validators are delivered to other honest validators within a maximal network delay δ (this is the *partial synchrony* model introduced in [32]). In addition to traditional guarantees,

LibraBFT maintains safety when validators crash and restart — even if all validators restart at the same time.

Overview of LibraBFT. Validators receive transactions from clients and share them with each other through a shared mempool protocol. The LibraBFT protocol then proceeds in a sequence of *rounds*. In each round, a validator takes the role of *leader* and proposes a block of transactions to extend a certified sequence of blocks (see quorum certificates below) that contain the full previous transaction history. A validator receives the proposed block and checks their voting rules to determine if it should vote for certifying this block. These simple rules ensure the safety of LibraBFT — and their implementation can be cleanly separated and audited.[10] If the validator intends to vote for this block, it executes the block's transactions speculatively and without external effect. This results in the computation of an authenticator for the database that results from the execution of the block. The validator then sends a signed vote for the block and the database authenticator to the leader. The leader gathers these votes to form a quorum certificate (QC), which provides evidence of $\geq 2f+1$ votes for this block and broadcasts the QC to all validators.

A block is committed when a contiguous 3-chain commit rule is met. A block at round k is committed if it has a QC and is confirmed by two more blocks and QCs at rounds $k+1$ and $k+2$. The commit rule eventually allows honest validators to commit a block. LibraBFT guarantees that all honest validators will eventually commit the block (and proceeding sequence of blocks linked from it). Once a sequence of blocks has committed, the state that results from executing their transactions can be persisted and forms a replicated database.

Advantages of the HotStuff paradigm. We evaluated several BFT-based protocols against the dimensions of performance, reliability, security, ease of robust implementation, and operational overhead for validators. Our goal was to choose a protocol that would initially support at least 100 validators and would be able to evolve over time to support 500–1,000 validators. We had three reasons for selecting the HotStuff protocol as the basis for LibraBFT: (1) simplicity and modularity of the safety argument; (2) ability to easily integrate consensus with execution; and (3) promising performance in early experiments.

The HotStuff protocol decomposes into modules for safety (voting and commit rules) and liveness (pacemaker). This decoupling provides the ability to develop and experiment independently and on different modules in parallel. Due to the simple voting and commit rules, protocol safety is easy to implement and verify. It is straightforward to integrate execution as a part of consensus to avoid forking issues that arise from non-deterministic execution in a leader-based protocol. Finally, our early prototypes confirmed high throughput and low transaction latency as independently measured in HotStuff [13]. We did not consider proof-of-work based protocols due to their poor performance and high energy (and environmental) costs [24].

HotStuff extensions and modifications. In LibraBFT, to better support the goals of the Libra ecosystem, we extend and adapt the core HotStuff protocol and implementation in several ways. Importantly, we reformulate the safety conditions and provide extended proofs of safety, liveness, and optimistic responsiveness. We also implement a number of additional features. First, we make the protocol more resistant to non-determinism bugs by having validators collectively sign the resulting state of a block rather than just the sequence of transactions. This also allows clients to use QCs to authenticate reads from the database. Second, we design a pacemaker that emits explicit timeouts, and validators rely on a quorum of those to move to the next round — without requiring synchronized clocks. Third, we design an unpredictable leader election mechanism in which the leader of a round

[10] Validators only vote for increasing rounds of proposals and only vote if the proposed block links to a previous block with an equal or higher preferred round number. The full details and proofs of safety are presented in the separate report on LibraBFT.

is determined by the proposer of the latest committed block using a verifiable random function [37]. This mechanism limits the window of time in which an adversary can launch an effective denial-of-service attack against a leader. Fourth, we use aggregate signatures [38] that preserve the identity validators who sign QCs. This allows us to provide incentives to validators that contribute to QCs (discussed in Section 9.3). It also does not require a complex threshold key setup [39].

Validator management. The Libra protocol manages the set of validators using a Move module. This creates a clean separation between the consensus system and the cryptoeconomic system that defines a trusted set of validators. We discuss the Libra Blockchain's implementation of this contract in Section 9.2. Abstracting the validator management in this way is an example of the flexibility granted by defining core blockchain primitives in Move.

Each change to the group of validators defines a new epoch. If a transaction causes the validator management module to alter the validator set, that transaction will be the last transaction committed by the current epoch — any subsequent transactions in that block or future blocks from that epoch will be ignored. Once the transaction has been committed, the new set of validators can start the next epoch of the consensus protocol.

Within an epoch, clients do not need to synchronize every QC. Since a committed QC contains a binding commitment to all previous states, a client only needs to synchronize to the latest available QC in its current epoch. If this QC is the last in its epoch, the client can see the new set of validators, update its epoch, and again synchronize to the latest QC. If a validator chooses to prune history as described in Section 4.2, it needs to retain at least enough data to provide proof of the validator set change to clients.

The validator management contract must provide a validator set that satisfies the security properties required by the consensus protocol. No more than f voting power can be controlled by Byzantine validators. The voting power must remain honest both during the epoch as well as for a period time after the epoch in order to allow clients to synchronize to the new configuration. A client that is offline for longer than this period needs to resynchronize using some external source of truth to acquire a checkpoint that they trust (e.g., from the source it uses to receive updated software). Furthermore, the validator set must not rotate so frequently that the rotation disrupts the performance of the system or causes clients to download QCs for an excessive number of epochs. We plan to research the optimal epoch length but anticipate it to be less than a day.

6 Networking

The Libra protocol, like other decentralized systems, needs a networking substrate to enable communication between its members. Both the consensus and shared mempool protocols between validators require communication over the internet, as described in Section 5 and Section 7, respectively.

The network layer is designed to be general-purpose and draws inspiration from the *libp2p* [40] project. It currently provides two primary interfaces: (1) Remote Procedure Calls (RPC) and (2) DirectSend, which implements fire-and-forget-style message delivery to a single receiver.

The inter-validator network is implemented as a peer-to-peer system using *Multiaddr* [41] scheme for peer addressing, TCP for reliable transport, *Noise* [42] for authentication and full end-to-end encryption, *Yamux* [43] for multiplexing substreams over a single connection, and push-style gossip for peer discovery. Each new substream is assigned a *protocol* supported by both the sender and the receiver. Each RPC and DirectSend type corresponds to one such *protocol*.

The networking system uses the same validator set management smart contract as the consensus system as a source of truth for the current validator set. This contract holds the network public key and consensus public key of each validator. A validator detects changes in the validator set by watching for changes in this smart contract. To join the inter-validator network, a validator must authenticate using a network public key in the most recent validator set defined by the contract. Bootstrapping a validator requires a list of seed peers, which first authenticate the joining validator as an eligible member of the inter-validator network and then share their state with the new peer.

Each validator in the Libra protocol maintains a full membership view of the system and connects directly to any validator it needs to communicate with. A validator that cannot be connected to directly is assumed to fall within the quota of Byzantine faults tolerated by the system. Validator health information, determined using periodic liveness probes, is not shared between validators; instead, each validator directly monitors its peers for liveness. We expect this approach to scale up to a few hundred validators before requiring partial membership views, sophisticated failure detectors, or communication relays.

7 Libra Core Implementation

To validate the Libra protocol, we have built an open-source prototype implementation, *Libra Core* [6]. The implementation is written in Rust. We chose Rust due to its focus on enabling safe coding practices, support for systems programming, and high performance. We have split the internal components of the system as gRPC [44] services. Modularity allows for better security; for instance, the consensus safety component can be run in a separate process or even on a different machine.

The security of the Libra Blockchain rests on the correct implementation of validators, Move programs, and the Move VM. Addressing these issues in Libra Core is a work in progress. It involves isolating the parts of the code that contribute to a validator signing a block of transactions during consensus and applying measures to increase assurance in the correctness of these components (e.g., extensive testing, formal specification, and formal verification). Developing high assurance also involves ensuring the security of the dependencies of the code (e.g., the code review process, source control, open-source library dependencies, build systems, and release management).

7.1 Write Request Lifecycle

Figure 6 shows the lifecycle of transactions within Libra Core that support write operations to the decentralized database. This section takes an in-depth look at how a transaction flows through the internal components of the validators.

A request begins when a client wishes to submit a transaction to the database. We currently assume clients have an out-of-band mechanism to find the addresses of validators to submit transactions to — the final version of this mechanism is yet to be designed.

Admission control. Upon receiving a transaction, a validator's *admission control* component performs initial syntactic checks (❶) to discard malformed transactions that will never be executed. Doing these checks as early as possible avoids spending resources on transactions that spam the system. Admission control may access the VM (❷), which uses the storage component to perform checks, such as ensuring the account has a sufficient balance to pay the gas for the transaction. The admission control component is designed so that a validator can run multiple instances of the component. This design allows the validator to scale the processing of incoming transactions and mitigate denial-of-service attacks.

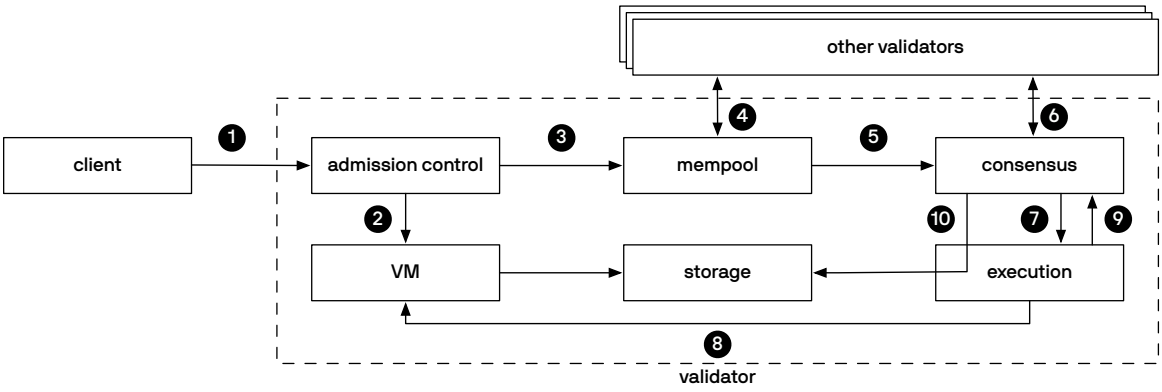

Figure 6: The flow of a write transaction through the internal components of Libra Core.

Mempool. Transactions that pass the checks of the admission control component are sent to the validator's *mempool*, which holds transactions waiting to be executed in an in-memory buffer (❸). The mempool may hold multiple transactions sent from the same address. Because the mempool processes multiple transactions at a time, it can perform checks such as validating that a sequence of operations on the same address can all pay for gas that the admission control system cannot. Using the shared-mempool protocol (❹), a validator shares the transactions in its mempool with other validators and places transactions received from other validators in its own mempool.

Consensus. Validators create blocks by selecting a sequence of transactions from their mempool. When a validator acts as the leader of the consensus protocol, it forms a block of transactions from its mempool (❺). It sends this block of transactions as a proposal to other validators (❻). The *consensus* component is responsible for coordinating agreement among all validators on the sequence of these blocks of transactions and their resulting execution using the LibraBFT protocol (Section 5).

Transaction execution. As part of reaching agreement, the block of transactions is passed to the *executor* component (❼), which manages the execution of transactions (Section 3) in the VM (❽). This execution happens speculatively before the transaction has been agreed on. This early execution is safe since transactions are deterministic and have no external effects. After executing the transactions in the block, the execution component builds a ledger history (Section 4.2) with these transactions appended. The ledger history authenticator is returned to the consensus component (❾).

The leader then attempts to reach consensus on this authenticator by forming a chain of quorum certificates (Section 5), each of which is signed by a set of validators with at least $2f + 1$ votes.

Committing a block. Once the consensus algorithm reaches agreement, any honest validator can be sure that all other honest validators will eventually commit a consistent ledger history. The validator reads the result of the block execution from the cache in the execution component and updates its local database storage (❿).

7.2 Read Request Lifecycle

Clients may also submit read requests to query validators for the content of an account in the decentralized database. Read requests do not mutate state and can be processed locally without going through consensus. Reads are submitted to the validator's admission control component. The admission control component performs preliminary checks, reads the data from storage, and the result is sent back to the client. The response comes with a quorum certificate (described in Section 5) that

contains a root hash (described in Section 4). The QC allows the client to authenticate the response to the query. The client uses the same technique as the VM to interpret the raw bytes of an account using the logical data model (Section 2). For example, to read the balance of the account at address a, the client decodes the `LibraCoin.T` resource embedded inside the account's raw bytes.

8 Performance

The mission of the Libra protocol is to support a global financial infrastructure. Performance is an integral part of meeting that need. We discuss three components of blockchain performance:

1. **Throughput:** The number of transactions that the blockchain can process per second.
2. **Latency:** The time between a client submitting a transaction to the blockchain and another party seeing that the transaction was committed.
3. **Capacity:** The ability of the blockchain to store a large number of accounts.

While the Libra protocol is still at a prototype stage and we do not have concrete performance metrics yet to report, we anticipate the initial launch of Libra protocol to support 1,000 payment transactions per second with a 10-second finality time between a transaction being submitted and committed. Over time, we expect to be able to increase the system's throughput to meet the needs of the network. We anticipate that many payment transactions will occur off-chain, for example, within a custodial wallet or by using payment channels [45]. Therefore, we believe that supporting 1,000 transactions per second on the blockchain will meet the initial needs of the ecosystem. The Libra protocol is designed to achieve these goals in several ways:

Protocol design. Many elements of the Libra protocol are chosen partly based on performance. For example, the LibraBFT algorithm achieves consensus in three rounds of network communication and does not require any real-time delay to propose or vote on blocks. This allows the commit latency to be limited only by the network latency between validators.

We also select elements of the protocol with parallelization and sharding in mind. The sparse Merkle tree approach to computing authenticators allows sharding the database across multiple machines (which increases capacity) or processing updates in parallel (which increases throughput). Initial transaction validation, which includes computationally expensive signature verification, can also be parallelized.

Validator selection. Like most services, the performance of the Libra Blockchain depends on the performance of the underlying validators that operate it. There is a tradeoff between decentralization and performance. Requiring extremely well-resourced validators limits the number of entities that could perform that role. However, the presence of extremely under-resourced validators would limit the performance of the whole system.

We favor a balance of these approaches by targeting nodes that can run on commodity hardware that many entities can purchase. However, we do assume that nodes run on server-class hardware and within well-connected data centers. We use an approximate analysis to show that the system is likely able to meet the demand of 1,000 transactions per second.

- **Bandwidth**: If we assume that each transaction requires 5 KB of traffic — including the cost of receiving the transaction via the mempool, rebroadcasting it, receiving blocks from the leader, and replicating to clients — then validators require a 40 Mbps internet connection to support 1,000 transactions per second. Access to such bandwidth is widely available.

- **CPU**: Signature verification is a significant computational cost associated with a payment transaction. We have designed the protocol to allow parallel verification of transaction signatures. Modern signature schemes support over 1,000 verifications per second over a commodity CPU.
- **Disk**: Servers with 16 TB of SSD storage are available from major server vendors. Since the current state is the only piece of information the validator needs to use to process a transaction, we estimate that if accounts are approximately 4 KB (inclusive of all forms of overhead), then this allows validators to store 4 billion accounts. We anticipate that developments in disk storage, scaling validators to multiple shards, and economic incentives will allow the system to remain accessible to commodity systems.

Historical data may grow beyond the amount that can be handled by an individual server. Validators are free to discard historical data not needed to process new transactions (see Section 4.2); however, this data may be of interest to clients who wish to query events from past transactions. Since the validators sign a binding commitment to this data, clients are free to use any system to access data without having to trust the system that delivers it. We expect this type of read traffic to be easy to scale through parallelism.

9 Implementing Libra Ecosystem Policies with Move

The Libra Blockchain is a unique system that balances the stability of traditional financial networks with the openness offered by systems governed by cryptoeconomic means. As we discuss in Section 1, the Libra protocol is designed to support the Libra ecosystem in implementing novel economic [4] and governance [3] policies. The protocol specifies a flexible framework that is parametric in key system components such as the native currency, validator management, and transaction validation. In this section, we discuss how the Libra Blockchain uses the Move programming language to customize these components. Our discussion focuses on both the challenges of aligning network participant and validator incentives as well as the challenges of supporting the operations, governance, and evolution of the Libra ecosystem.

9.1 Libra Coin

Many cryptocurrencies are not backed by real-world assets. As a result, investment and speculation have been primary use cases. Investors often acquire these currencies under the assumption that they will substantially appreciate and can later be sold at a higher price. Fluctuations in the beliefs about the long-term value of these currencies have caused corresponding fluctuations in price, which sometimes yield massive swings in value.

To drive widespread adoption, Libra is designed to be a currency where any user will know that the value of a Libra today will be close to its value tomorrow and in the future. The reserve is the key mechanism for achieving value preservation. Through the reserve, each coin is fully backed with a set of stable and liquid assets. With the presence of a competitive group of liquidity providers that interface with the reserve, users can have confidence that any coin they hold can be sold for fiat currency at a narrow spread above or below the value of the underlying assets. This gives the coin intrinsic value from the start and helps protect against the speculative swings that are experienced by existing cryptocurrencies.

The reserve assets are a collection of low-volatility assets, including cash and government securities from stable and reputable central banks. As the value of Libra is effectively linked to a basket of fiat currencies, from the point of view of any specific currency, there will be fluctuations in the value of Libra. The makeup of the reserve is designed to mitigate the likelihood and severity of these

fluctuations, particularly in the negative direction (e.g., even in economic crises). To that end, the basket has been structured with capital preservation and liquidity in mind.

The reserve is managed by the Libra Association (see Section 9.2), which has published a detailed report on the reserve's operations [4]. Users do not directly interface with the reserve. Instead, to support higher efficiency, there are authorized resellers who are the only entities authorized by the association to transact large amounts of fiat and Libra in and out of the reserve. These authorized resellers integrate into exchanges and other institutions that buy and sell cryptocurrencies and provide these entities with liquidity for users who wish to convert from cash to Libra and back again.

To implement this scheme, the Libra coin contract allows the association to mint new coins when demand increases and destroy them when the demand contracts. The association does not set a monetary policy. It can only mint and burn coins in response to demand from authorized resellers. Users do not need to worry about the association introducing inflation into the system or debasing the currency: for new coins to be minted, there must be a commensurate fiat deposit in the reserve.

The ability to customize the Libra coin contract using Move allows the definition of this scheme without any modifications to the underlying protocol or the software that implements it. Additional functionality can be created, such as requiring multiple signatures to mint currency and creating limited-quantity keys to increase security.

9.2 Validator Management and Governance

The consensus algorithm relies on the validator-set management Move module to maintain the current set of validators and manage the allocation of votes among the validators. This contract is responsible for maintaining a validator set in which at most f votes out of $3f + 1$ total votes are controlled by Byzantine validators.

Initially, the Libra Blockchain only grants votes to *Founding Members*, entities that: (1) meet a set of predefined Founding Member eligibility criteria [46] and (2) commit a certain amount into the project. These rules help to ensure the security requirements of having a safe and live validator set. Using the Founding Member eligibility criteria ensures that the Founding Members are organizations with established reputations, making it unlikely that they would act maliciously, and suggesting that they will apply diligence in defending their validator against outside attacks.

While this method of assessing validator eligibility is an improvement on traditional permissioned blockchains, which usually form a set of closed business relationships, we aspire to make the Libra Blockchain fully permissionless. To do this, we plan to gradually transition to a *proof-of-stake* [47] system where validators are assigned voting rights proportional to the number of Libra coins they hold. This transitions the governance of the ecosystem to its users while preventing Sybil attacks [48] by requiring validators to hold a scarce resource and align their incentives with healthy system operations. This transition requires (1) the ecosystem to be sufficiently large to prevent a single bad actor from causing disruption; (2) the existence of a competitive and reliable market for delegation for users that do not wish to become validators; and (3) addressing technological and usability challenges in the staking of Libra coins.

Another unique aspect of governance in the Libra ecosystem is that the validators form a real-world entity, the not-for-profit Libra Association, which is managed by a council of validators. A member in the association council represents each validator. A member's voting weight in the council is the same as the validator's voting weight in the consensus protocol. The association performs tasks such as managing the reserve, assessing the validator eligibility criteria, and guiding the open-source development of the Libra protocol.

Move makes it possible to encode the rules for validator management and governance as a module. Move allows the staking of coins by wrapping them in a resource that prevents access to the underlying asset. The staked resources can be used to compute the voting rights of the validators. The contract can configure the interval at which changes take effect, to reduce churn of the validator set.

The Libra Association's operation is also aided by Move. Since the association is the operator of the reserve, it can create Move modules that delegate the authority to mint and burn coins to an operational arm that interacts with authorized resellers. This operational arm can also assess if potential Founding Members meet the eligibility criteria. Move allows flexible governance mechanisms such as allowing the council to assert its authority and take back its delegated authority through a vote.

The association has published a detailed document outlining its proposed structure [3]. All governance in the association stems from the council of validators — this council has the ultimate right to assert any authority provided to the association. Thus, the governance of the entire Libra ecosystem evolves as the validator set changes from the initial set of Founding Members to a set based on proof of stake.

9.3 Validator Security and Incentives

In the initial setting, using Founding Members as validators, we believe that the institutional reputation and financial incentives of each validator are sufficient to ensure that Byzantine validators control no more than f votes. In the future, however, an open system where representation is based on coin ownership will require a substantially different market design. We have started to understand the governance and equilibrium structure of a blockchain system based on stakeholdings and consumer confidence in wallets and other delegates. In the process, we have identified new market design trade-offs between the Libra approach and more established approaches, such as proof of work [49].

However, more research is needed to determine how best to maintain long-run competition in the ecosystem while ensuring the security and efficiency of the network. Furthermore, as stake-based governance introduces path dependence in influence, it is essential to explore mechanisms for protecting smaller stakeholders and service providers.

Move allows flexibility in the definition of the relevant incentive schemes such as gas pricing or staking. For example, *stake slashing*, a commonly discussed mechanism, could be implemented in Move by locking stake for a period of time and automatically punishing validators if they violate the rules of the LibraBFT algorithm in a way that affects safety.

Similarly, when a validator votes in the LibraBFT algorithm, those votes can be recorded in the database. This record allows a Move module to distribute incentives based on participation in the algorithm, thereby incentivizing validators to be live. Interest generated on the Libra Reserve and gas payments can also be used as sources of incentives. Both sources are managed by Move modules, which add flexibility in their allocation. While more research is required to design an approach that will support the evolution of the Libra ecosystem, the flexibility of Move ensures that the desired approach can be implemented with few, if any, changes to the Libra protocol.

10 What's Next for Libra?

We have presented a proposal for the Libra protocol, which allows a set of validators to provide a decentralized database for tracking programmable resources. We have discussed an open-source prototype — Libra Core — of the Libra protocol and shown how the introduction of the Move

programming language for smart contracts allows the protocol to implement the unique design of the Libra ecosystem.

Both the protocol and the implementation described in this paper are currently at the prototype stage. We hope to gather feedback from the newly formed Libra Association, as well as the broader community, to turn these ideas into an open financial infrastructure. We are currently running a testnet to allow the community to experiment with this system.

We are working toward an initial launch of the system, and to keep it within a manageable scope, we plan to make several simplifications in the first version. In the early days of the system, using an externally recognized set of Founding Members reduces the demands on the consensus incentive system and allows for a faster pace of updates. We anticipate using Move only for system-defined modules, as opposed to letting users define their own modules, which allows for the Libra ecosystem to launch before the Move language is fully formed. This also allows for breaking changes to be made without compromising the flexibility that comes from defining core system behavior using Move. However, we intend for future versions of the Libra protocol to provide open access to the Move language.

Finally, we are currently working within the framework of the Libra Association to launch the technical infrastructure behind this new ecosystem. We have published a roadmap [50] of work that we plan to contribute to support this launch. One of the top goals of the Libra Association is to migrate the Libra ecosystem to a permissionless system. We have documented the technical challenges [5] involved in making this migration. The association's open-source community [6] provides information about how to start using the Libra testnet, try out the Move language, and contribute to the Libra ecosystem.

Acknowledgments

We would like to thank the following people for helpful discussions and feedback on this paper: Adrien Auclert, Morgan Beller, Tarun Chitra, James Everingham, Maurice Herlihy, Ravi Jagadeesan, Archana Jayaswal, Scott Duke Kominers, John Mitchell, Rajesh Nishtala, Roberto Rigobon, Jared Saia, Christina Smedley, Catherine Tucker, Kevin Weil, David Wong, Howard Wu, Nina Yiamsamatha, and Kevin Zhang.

References

[1] C. Catalini and J. S. Gans, "Some simple economics of the blockchain," *WP No. 22952, National Bureau of Economic Research*, 2016.

[2] The Libra Association, "An Introduction to Libra," https://libra.org/en-us/whitepaper.

[3] The Libra Association, 2019.

[4] C. Catalini *et al.*, "The Libra reserve," 2019.

[5] S. Bano *et al.*, "Moving toward permissionless consensus," 2019.

[6] The Libra Association, "Libra developers," https://developers.libra.org/.

[7] P. T. Devanbu *et al.*, "Authentic third-party data publication," in *Data and Application Security, Development and Directions, IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security, Schoorl, The Netherlands, August 21-23, 2000*, 2000, pp. 101–112.

[8] M. Naor and K. Nissim, "Certificate revocation and certificate update," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 4, pp. 561–570, 2000.

[9] R. Tamassia, "Authenticated data structures," in *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, 2003, pp. 2–5.

[10] S. Blackshear *et al.*, "Move: A language with programmable resources," https://developers.libra.org/docs/move-paper.

[11] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, 1987, pp. 369–378.

[12] S. Bano *et al.*, "State machine replication in the Libra Blockchain," https://developers.libra.org/docs/state-machine-replication-paper.

[13] M. Yin *et al.*, "Hotstuff: BFT consensus in the lens of blockchain," 2018. [Online]. Available: https://arxiv.org/abs/1803.05069

[14] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency control and recovery in database systems." Addison-Wesley, 1987.

[15] D. P. Reed, "Naming and synchronization in a decentralized computer system," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[16] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger," http://gavwood.com/paper.pdf, 2016.

[17] G. Bertoni *et al.*, "Keccak," in *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, 2013, pp. 313–314.

[18] S. Josefsson and I. Liusvaara, "Edwards-curve digital signature algorithm (EdDSA)," *RFC*, vol. 8032, pp. 1–60, 2017.

[19] A. Pfitzmann and M. Köhntopp, "Anonymity, unobservability, and pseudonymity—a proposal for terminology," in *Designing privacy enhancing technologies*, 2001, pp. 1–9.

[20] B. C. Pierce, "Types and programming languages." MIT Press, 2002, ch. 19.

[21] J. Girard, "Light linear logic," *Inf. Comput.*, vol. 143, no. 2, pp. 175–204, 1998.

[22] R. Milner, M. Tofte, and R. Harper, "Definition of standard ML." MIT Press, 1990.

[23] "Leaf-node weakness in Bitcoin merkle tree design," https://bitslog.com/2018/06/09/leaf-node-weakness-in-bitcoin-merkle-tree-design/.

[24] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," https://bitcoin.org/bitcoin.pdf, 2008.

[25] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, 2009, pp. 317–334.

[26] L. Reyzin and S. Yakoubov, "Efficient asynchronous accumulators for distributed PKI," in *Security and Cryptography for Networks - 10th International Conference, SCN 2016, Amalfi, Italy, August 31 - September 2, 2016, Proceedings*, 2016, pp. 292–309.

[27] B. Laurie, A. Langley, and E. Käsper, "Certificate transparency," *RFC*, vol. 6962, pp. 1–27, 2013.

[28] J. R. Driscoll *et al.*, "Making data structures persistent," *J. Comput. Syst. Sci.*, vol. 38, no. 1, pp. 86–124, 1989.

[29] C. Okasaki, "Purely functional data structures." Cambridge University Press, 1999.

[30] L. Reyzin *et al.*, "Improving authenticated dynamic dictionaries, with applications to cryptocurrencies," in *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*, 2017, pp. 376–392.

[31] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, pp. 299–319, 1990.

[32] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[33] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999, pp. 173–186.

[34] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *CoRR*, vol. abs/1710.09437, 2017.

[35] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018.

[36] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS'82)*, vol. 4, no. 3, pp. 382–401, 1982.

[37] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *40th Annual Symposium on Foundations of Computer Science (FOCS'99)*. IEEE, 1999, pp. 120–130.

[38] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," in *Advances in Cryptology (ASIACRYPT 2001)*. Springer-Verlag, 2001, pp. 514–532.

[39] A. Kate and I. Goldberg, "Distributed key generation for the internet," in *IEEE International Conference on Distributed Computing Systems (ICDCS'09)*, 2009, pp. 119–128.

[40] "The `libp2p` project," https://libp2p.io/.

[41] "The `multiaddr` project," https://multiformats.io/multiaddr/.

[42] T. Perrin, "The `noise` project," https://noiseprotocol.org/noise.html, 2018.

[43] "The `yamux` project," https://github.com/hashicorp/yamux/blob/master/spec.md, 2016.

[44] "The `gRPC` project," https://grpc.io/.

[45] L. Gudgeon *et al.*, "SoK: Off the chain transactions," *IACR Cryptology ePrint Archive*, vol. 2019, p. 360, 2019.

[46] The Libra Association, "Becoming a Founding Member," 2019.

[47] I. Bentov, A. Gabizon, and A. Mizrahi, "Cryptocurrencies without proof of work," in *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, 2016, pp. 142–157.

[48] J. R. Douceur, "The sybil attack," in *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*, 2002, pp. 251–260.

[49] C. Catalini, R. Jagadeesan, and S. D. Kominers, "Market design for a blockchain-based financial system," *SSRN Working Paper No. 3396834*, 2019.

[50] The Libra Association, "The path forward," https://developers.libra.org/blog/2019/06/18/the-path-forward.