An Implementation of Omniledger

Pablo Lorenceau

School of Computer and Communication Sciences

Decentralized and Distributed Systems Lab (DEDIS)

Master Semester Project

June 2018

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Linus Gasser
EPFL / DEDIS

Contents

1 Introduction

Distributed legder technology is gaining traction in many applications, first and foremost in decentralized payment systems such as e.g. Bitcoin. This adaption imposes challenges on the scalability of such systems, both in total transaction throughput, as well as in the number of participating processing nodes.

This report describes the implementation of OmniLedger [10], a scalable blockchain with a flexible user interface. We will first describe the necessary background in 2 and discuss how OmniLedger improves on the current state of blockchain technology.

We will also discuss some of the protocols and data structures employed by our implementation.

Section 3 will discuss implementation details such as how transactions are processed, how clients interact with the service and how access control is implemented.

Finally, section 4 will outline future work and conclude.

2 Background

The issue of scaling blockchains has been nicely summarized in [7]. To give the reader some motivation and context, we will now discuss some of the points from that paper which are relevant to the system at hand. The authors of [7] differntiate between several planes, such as e.g. the *Network Plane*, the *Storage Plane* and the *Consensus Plane*. OmniLedger is mainly concerened with optimizing the Consensus Plane. Key issues of the Consensus Plane the authors of [7] identify include:

- **The consensus protocol**: Bitcoin's Nakamoto consensus [12] allows (theoretically) for f byzantine nodes out of $2f + 1$ nodes, but a protocol with stronger assumptions, such as *Practical Byzantine Fault Tolerance* (PBFT) [6] could be more efficient.

- **Improving proof-of-work protocols**: In the Bitcoin protocol, there is a tradeoff between consensus speed (and thus bandwidth) and the number of forks.

- **Sharding**: The authors suggest that splitting up the consesus protocol between individual shards could increase throughput.

This report describes the current implementation of *OmniLedger* [10]. Omniledger is designed as a scalable, permissionless, sharded ledger. The motivation for the project is to create a decentralized ledger with the ability to "scale-out", i.e. increase throughput linearly with the number of nodes. To do so, Omniledger captures the ideas from [7] listed above. We will now go over each of them, and discussing how OmniLedger makes use of it to improve latency, throughput and scalability.

2.1 The Consensus Protocol

OmniLedger uses ByzCoin [11] as a consesnus protocol. In ByzCoin, the nodes decide collectively on one single chain, no forks occur. ByzCoin is a byzantine fault tolerant consesus protocol, whcih guarantees safety and liveness for up to f byzantine nodes out of $3f + 2$ total nodes. When a block is to be added to the blockchain, a consensus round is run in order to vote on whether to accept or reject the block. Due to this mechanism, no forks can occur in the blockchain. Therefore, no work which is put in the validation of blocks is ever lost, as would be the case if a fork occured and one of the branches had to be pruned.

ByzCoin is based on PBFT [6], but reduces the communication complexity from $O(n^2)$ to $O(\log n)$ by using a tree based collective signing protocol [15]. The node at the root of this tree is the one which initializes the collective signing protocol. We will refer to this particular node as the *leader* in

future occurences.

Figure 1: ByzCoin's collective signing protocol: The nodes are organized in a tree structure and collectively sign each block (copied from [15]).

2.2 The Proof-of-Work Protocol

In order to establish Sybil attack resistant identities for ByzCoin's consensus group, an identity blockchain, separate from the data holding blockchain is employed: When a miner finds a new block of this chain, she receives a *consensus group share*, which gives her voting power on data holding blocks until the next w blocks of the identity blockchain have been mined. This decouples the Proof-of-Work from the verification of transactions, allowing for a lower transaction processing latency.

2.3 Sharding

In order to take load off of individual nodes as well as reduce consesnus group size (which in turn reduces consensus latency), OmniLedger employs sharding. This means, that each validoator is only repsonsible for a subset of the data, namely its shard.

Letting nodes choose the shard they are responsible for might allow an attacker to focus on one single shard and control it. Therefore, nodes must be assigned to shards in a secure, random way. In order to assign validators to shards, OmniLedger uses RandHound [14]. RandHound is a scalable public randomness generation protocol. It provides publicly-verifiable randomness which can neither be biased nor predicted by any involved party except with negligible probability, as long as there are at most f malicious nodes out of $3f + 1$ total nodes.

The protocol is designed as a client/server model, where a client requests randomness from a set of servers (nodes). The protocol works roughly as follows:

- The client allocates the servers to disjoint groups. The members of a given group will have to set up a shared secret. Splitting the servers up into groups instead of having one global group reduces the communication complexity.
- Each server chooses a secret random value. The servers of a group then create a shared secret based on all the random values of the group.
- Each server then sends his encrypted share of the secret to the client, who then chooses a subset of the servers and submits this selection to the nodes for collective signing [15].
- Subsequently, the servers send their unencrypted shares of the secret to the client who combines them to obtain the final random output.

This is only a quick overview over RandHound, for more details, consider [14].

OmniLedger supports cross shard transactions, i.e. transactions which touch data in more than one shard. Since one shard might allow his part of a transaction while the other does not, such transactions must be processed atomically. Otherwise the shards could end up in an inconsistent state. In a payment system, this means that funds could be locked forever. Atomicity is acheived by *Atomix* [10], an atomic cross-shard commit protocol. Atomix is a client driven protocol, where the client first locks all the resources the transaction touches. If a shard does not allow the transaction the client will not be able to obtain the corresponding lock. If the client could obtain all locks, he can subsequently unlock the resources and commit the transaction. Otherwise he unlocks to abort it. Since a shard is collectively honest and does not fail (this is acheived by Randhound, see above), this protocol is sufficient to provide the desired functionality.

2.4 Skipchain: The Underlying Data Structure

While not specified in [10], the implementation of OmniLedger uses the *skipchain* [13]: A skipchain is a blockchain which contains cryptographically signed forward links: In addition to the backwards pointing hash of the previous block, as known from e.g. Bitcoin [12], each block also contains pointers to future blocks.

These links enable a client to securely traverse time in forward as well as in backward direction. Since forward links may point to blocks several

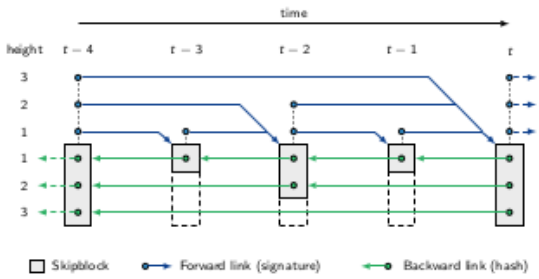

Figure 2: The skipchain: Both backward and forward links can span multiple hops, which allows a client to efficiently traverse the chain (copied from [13]).

positions ahead, blocks can be skipped when traversing the skipchain. This allows for more efficient traversal, which is important for clients which have not caught up with the skipchain's state in a long time. Since forward links point to future blocks, they can not be added at block creation time - the block they should point to does not yet exist. Therefore, they are added only once their target block exists and then cryptographically signed by the set of nodes that created the original block. This requires the corresponding nodes to stay live until all future blocks for which they have to sign forward links have been created.

Now we have seen how OmniLedger aims to improve current blockchain implementations. The paper however focuses mainly on the protocols needed for these improvements and does not provide further detail on how clients interact with the system or how transactions are applied to the system's state. Therefore, the next section will put its focus on these aspects of the implementaion.

3 Implementation

We will now discuss the current state of the OmniLedger implementation. The system is developed in the *Go* language [4].

In section 2, several building blocks which improve the scalability of blockchains have been described. Some of these have already been implemented in the *Cothority framework* [2], notably ByzCoin in the package *ByzCoinX* and the skipchain. The skipchain implementation already uses ByzCoinX, therefore our OmniLedger implementaion is built on top of it.

Currently, not all of the features described in section 2 are implemented, but they are planned to be in the future. For now, the only feature from section 2 present in the implementation is the consensus protocol: ByzCoinX. Furthermore, it should be noted that view changes in case the leader fails

are currently not implemented in the skipchain we use.

For the moment, in contrast to the system described in [10], the implementation operates as a permissioned blockchain. This means that the processing nodes of a skipchain must be known when it is initiated.

Having discussed the motivation and relevant background in 2, this section will now focus on how clients interact with the system and how transactions are processed.

First, a brief overview over the implementation's working principles will be given. We will then discuss the service's components and how transactions are processed in more detail. Finally, we will describe the messages the clients can use to interact with OmniLedger.

From now on, the name *OmniLedger* will refer to the implementation described in this section. If we refer to the paper [10], we will do this explicitly.

3.1 Overview

OmniLedger is implemented as a service accepting requests from clients. It maintains several independent skipchains, each storing its own state. The nodes of each skipchain are organized in a static tree, the leader at the root. We will refer to the nodes who are not the leader as *validators*.

The leader accepts clients' requests, queues them and periodically issues blocks, containing the transactions which have been queued since the last block. The validators must then find consensus on whether or not they accept the proposed block via ByzCoinX. If the block is accepted, its transactions will be applied.

A client's transaction is only accepted if the client can prove that he has the right to perform this transaction. The access right control is implemented via *Darcs*, which is short for distributed access rights control. These rights are part of the service's state itself and can also be evolved by the corresponding transactions.

Before we start discussing how OmniLedger processes transactions, we need first to discuss darcs a little bit, in order to understand how access control is implemented.

A darc is a structure which contains rules, a mapping from actions to expressions. An action can be performed if and only if the corresponding expression is satisfied. An expression is a formula over identities which contains conjunctions and disjunctions. An identity can either be an Ed25519 public key, an X509EC public key or a darc itself. Having a darc as an identity allows to delegate access rights to groups of entities rather than just share a private key. An expression is satisfied if a set of signatures is

provided, which satifies the formula.
The darc package provides the following example in its documentation: Assume an expression of the form $(a \wedge b) \vee (c \wedge d)$. Then, the expression is satisfied if and only if either both signatures of the identities a and b are provided, or both signatures of c and d are provided. Given the corresponding access rights, a darc can be evolved, meaning that rules (i.e. action-expression mappings) can be added, modified, or removed. Evolving a darc will create a copy of it which contains a reference to all the previous versions of it. This allows a client to later correctly verify the evolution of the darc, by checking the signatures provided at each evolution step.

OmniLedger stores all of its state in a Merkle-tree based data structre. This allows the service to provide cryptographic proofs about the presence or absence of a given item in its state. Any client holding the Merkle-root of the tree can then verify these proofs.

To allow clients to obtain the Merkle-root in a secure fashion, it is stored in the block's header, which will also contain the hashes of all transactions and the resulting state modifications applied with that block.

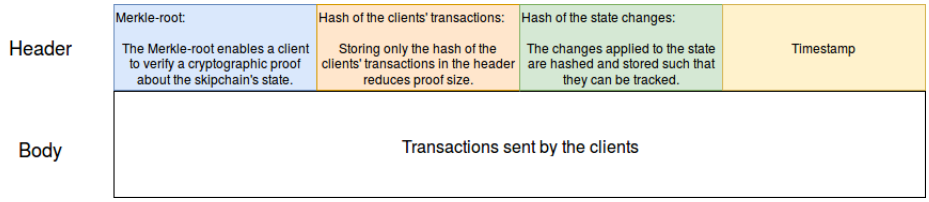

Figure 3: The structure of a skipblock. Only the data in the header is needed to validate the state of a skipchain.

Figure 3 shows the structure of OmniLedger's skipblock. We will discuss the content of the depicted fields in the follwoing. Note that the figure does not display any information about forward links or other data belonging to the skipchain implementation, since OmniLedger does not handle these.

Now that an overview over the system has been established, we will discuss the individual components needed to store state and process transaction in more detail.

3.2 Components

The OmniLedger service's components are grouped together in the type `Service`.

```
type Service struct {
    // We need to embed the ServiceProcessor,
    // so that incoming messages are correctly
    // handled.
    *onet.ServiceProcessor
    // collections cannot be stored,
    // so they will be re-created whenever
    // the service reloads.
    collectionDB map[string]*collectionDB

    // wokersMu protects access to queueWorkers
    workersMu sync.Mutex
    // queueWorkers is a map that points to
    // channels that handle queueing and
    // starting of new blocks.
    queueWorkers map[string]chan ClientTransaction

    // CloseQueues is closed when the queues
    // should stop - this is mostly for
    // testing and there should be a better
    // way to clean up services for testing...
    CloseQueues chan bool
    // contracts map kinds to kind specific
    // verification functions
    contracts map[string]OmniLedgerContract
    // propagate the new transactions
    propagateTransactions messaging.PropagationFunc

    storage *storage

    createSkipChainMut sync.Mutex
}
```

Listing 1: The structure of the OmniLedger service.

In short, communication is handeled by the embedded `onet.ServiceProcessor`, state is stored in the `collectionDB`, `queueWorkers` are channels used to communicate with worker functions which queue requests and create blocks, each one associated to a given skipchain. `CloseQueues` is used to terminate the worker functions, and `workersMu` is used to protect `queueWorkers` from concurrent accesses. The variable `contracts` associates to each kind of state

an `OmniLedgerContract`, which is essentially a function which takes state, a so called `Instruction` and a slice of `coin`s and returns all the changes it has applied to the state, as well as a slice of remaining `coins`. The `coin`s can be used to limit the amount of instructions a client can execute.

We will now discuss the most important components and some of their interfaces in more detail. For further information, we would like to refer you to the corresponding repository [5].

3.2.1 State Storage: Collections

At the heart of the service lies the type `Collection`. A collection is a Merkle tree based data structure, which allowes to store key-value associations in the form of `byte` slices and can issue cryptographic profs of the presence or absence of a key-value pair. It allows to store data on an untrusted server while still having proofs about it's exact state. In the implementation, we use the wrapper `collectionDB`, which allows us to use a *BoltDB* [1] to store the state. The type `collectionDB` exposes `Add`, `Set` and `Delete` methods, which allow to add new key-value pairs, set the value for a given key or remove a key-value pair from the collection, respectively.

3.2.2 Processing Transactions with Contracts

The OmniLedger service uses the `collectionDB` to store state associated with so called `OmniLedgerContract`s. These contracts define how clients can interact with the service. They are responsible for applying the deciding whether the transactions in a lock are valid or not and thus whether the block should be accepted or not.

Clients send transactions to the service, (see type `ClientTransaction`) which contain insdivitual instrucions (see the type `Instruction`). An instruction contains one of three actions: `Spawn`, `Invoke` and `Delete`. These actions allow to initiate a contract with a specified initial state or modify or remove the state associated with a contract. The spawn and invoke instructions can also contain arguments format which the contract can access. Additionally -among others-, an instruction contains an `ObjectID` which works as a key for the collection in order to identify the state to be modified or deleted, or as a definition of a new intial state's key. Additionally, the `ObjectID` contains a `DarcID`, which in turn points to the darc which is responsible for the access rights to this object. Furthermore, an `Instruction` contains a slice of darc signatures, the use of which will be discussed below.

Each value stored in the collection contains a `ContractID`. This ID is a key for the service's `contracts` map, which contains `OmniLedgerContract`s as values. These contracts have the type `func(cdb collection.Collection, tx Instruction, c []Coin) ([]StateChange, []Coin, error)`. Contracts specify how the Instructions are to be interpreted by the service. Since the

interpretation of `Instructions` depends on the state of the collection, a contract takes a reference to it as an input parameter. Additionally, `coins` can be required to limit the number of instructions a client can execute. The `ContractID` and the state of the contract is stored in the collection, accessible via the key `ObjectID` contained in the interpreted instruction. When a block is proposed to the validators, they iterate through each transaction in it and apply the corresponding contract to each instruction.

In order to ensure that only clients with the corresponding access rights interact with a given object, the contracts can check the darc signatures in the instruction against the darc pointed to by the `ObjectID`. If the contract does not output an error, the instruction is considered valid. A transaction is valid if and only if all its instructions are valid. This prevents the service from partially applying transactions, which might otherwise have effects which the client did not expect.

For valid transactions, the contract outputs a slice of state changes (see the type `StateChange`). A state change contains a `StateAction`, one of `Create`, `Update` or `Remove`. These state changes can only be created from `Spawn`, `Invoke` and `Delete` instructions, respectively. The state change also contains a `ContractID`, an `ObjectID` and a value. For the `Create`/`Update` action, the value with the key specified in the `ObjectID` is inserted/updated with the value and the `ContractID`. For the `Delete` instruction, this data is ignored, since a `Delete` instruction simply removes the corresponding key-value pair.

The advantage of translating instructions to state changes, is that it provides a more flexible interface for the clients which is easier to extend. One could e.g. also imagine instructions which are not translated to single state changes, but to a sequence.

We have now seen how clients sent transactions to OmniLedger and how the validity of blocks is ensured. The corresponding state changes are then only applied when the block has been accepted.

Having seen how a transaction is processed from the client to the collection, we will now discuss how the queueing of transactions at the leader is implemented.

3.3 Worker Functions

In order to reduce the amount of blocks on which the system has to vote, and thus improve throughput (in terms of transactions per second), transactions are queued at the leader. This is implemented by having one Goroutine per skipchain which listens on a channel for new incoming requests and creates a block containing all queued requests periodically. The frequency of blocks can be specified by the client when the skipchain is created. The channels to communicate with these worker functions are accessible through the

queueWorkers-map via the corresponding skipchain's SkipChainID.

We have now seen how blocks are created by the leader and how contracts are used to validate transactions and thus vote on proposed blocks.

We will now discuss what messages the client can use to interact with the service and how they are handled by OmniLedger.

3.4 Messages

The OmniLedger service accepts the following messages from clients:

3.4.1 Create the Genesis Block of a New Skipchain

The message CreateGenesisBlock creates a new skipchain at the service which will be maintained by the nodes specified in the Roster variable. A genesis darc is stored in the genesis block, and in order to write to this skipchain subsequently, a client must provide the signature of that darc. Thus, the entire configuration of this new skipchain can be found in it's genesis block.

```
// CreateGenesisBlock asks the service
// to set up a new skipchain.
type CreateGenesisBlock struct {
    // Version of the protocol
    Version Version
    // Roster defines which nodes
    // participate in the skipchain.
    Roster onet.Roster
    // GenesisDarc defines who is
    // allowed to write to this skipchain.
    GenesisDarc darc.Darc
    // BlockInterval in int64.
    BlockInterval time.Duration
}
```

Listing 2: The structure of the CreateGenesisBlock message.

The reply a client receives to this request (if no error occurred at the service while handling the request) is of the type `CreateGenesisBlockResponse`:

```
// CreateGenesisBlockResponse holds the
// genesis-block of the new skipchain.
type CreateGenesisBlockResponse struct {
    // Version of the protocol
    Version Version
    // Skipblock of the created skipchain
    // or empty if there was an error.
    Skipblock *skipchain.SkipBlock
}
```

Listing 3: The structure of the `CreateGenesisBlockResponse` message.

The `CreateGenesisBlock` request is handled by the service's method `CreateGenesisBlock`. This method makes sure that the `GenesisDarc` has the correct format, creates the genesis `Transaction` and uses it to create a new block which is then proposed to the other nodes to sign them via ByzCoinX protocol. Additionally, an asynchronous worker function corresponding to this new skipchain is created, which will queue requests and create blocks as specified in the `BlockInterval` field of the request. The worker function is implemented as a Goroutine which listens to requests on a channel stored in a `map` and has the corresponding `SkipchainID` as key.

3.4.2 Add a New Transaction

A client can request the service to execute a transaction by sending a message of the type `AddTxRequest`:

```
// AddTxRequest requests to apply
// a new transaction to the ledger.
type AddTxRequest struct {
    // Version of the protocol
    Version Version
    // SkipchainID is the hash of the first
       skipblock
    SkipchainID skipchain.SkipBlockID
    // Transaction to be applied to the kv-store
    Transaction ClientTransaction
}
```

Listing 4: The structure of the `AddTxRequest` message.

This message contains the transaction to be executed. The service handles the instructions with the `AddTransaction` method, which checks that

the corresponding skipchain exists and then writes the `Transaction` to the corresponding channel. The worker listening on the channel then takes care of creating the corresponding block and propose it to the validators, which will then vote on it.

Since the block is not necessarily issued directly after the request has been processed by the leader, the reply to this type of request is just an *ack*, containing the version of the running service.

3.4.3 Get a Proof About the Collection's State

As described in subsection 3.1, clients can obtain proofs about the collection's state. This is done by sending a message of the type `GetProof` to the service. The service will handle this request by creating a proof of the presence or absence of the value associated with `Key`.

```
type GetProof struct {
    // Version of the protocol
    Version Version
    // Key is the key we want to look up
    Key []byte
    // ID is any block that is known to
    // us in the skipchain, can be the genesis
    // block or any later block. The proof
    // returned will be starting at this block.
    ID skipchain.SkipBlockID
}
```

Listing 5: The structure of a `GetProof` message.

To obtain a verifiable proof, the client sends the ID of the last skipblock of the corresponding skipchain he knows. The service then replies by sending a structure containing an inclusion proof from the collection in its current state, the most recent block of the skipchain as well as the cryptographically singed forward links from the client's last known block to the latest skipblock. With this information, the client can now retrieve the Merkle-root of the collection in its current state from the last skipblock and verify the inclusion proof of the collection. Since the forward links from the last known block to the latest block are signed, the client can rest assured that the Merkle-root he retrieves from this block is genuine.

This section has described how OmniLedger's implementation processes clients' transactions and how clients can interact with the system.

The next section will offer a conclusion about the work done so far and briefly talk about future work.

4 Conclusion and Future Work

This report has presented the implementation of OmniLedger, as well as the necessary background on how this system fits into current blockchain technology and how it seeks to improve the state of said technology.

It has been shown how OmniLedger implements contracts to control how state is modified, and what language (i.e. transactions) is used to interact with it. Additionally we have seen how darcs can be used to have access rights which can be delegated, and where a specific combination of signatures can be required.

We have, however not seen any applications running on top of OmniLedger, in part because this would be beyond the scope of this report, but also since such applications -at the time of writing- have not yet been implemented or ported to OmniLedger.

However, several applications are planned to be implemented on top of OmniLedger. One example are *Onchain-secrets* [9], which would enable a blockchain to securely store secrets which are only accessible to clients which can provide the corresponding credentials. This could be implemented by having nodes holding the secret in an encrypted form and sending the corresponding key to the clients provided the correct credentials, log the access in the skipchain and then re-encrypt the secret with a new key.

Another application, which is currently implemented in the Cothority Framework, but is planned to be ported to Omniledger is *Proof of Personhood* [8]. Proof of Personhood (PoP) anonymously links a cryptographic token to a physical person, such that no person can have more than one token. This can be achieved by organizing parties with physical attendees who each deposit a public key, mutually verify that each attendee is an actual person, and then collectively sign the set of key. The results and proceedings of such parties could be stored on OmniLedger.

We hope that OmniLedger can be used as a flexible platform to implement a wide variety of services.

A Installation

The OmniLedger implementation is written in Go [4]. Therefore, an installation of Go should be present on the host where OmniLedger is to be installed. Note that the version of Go should not be older than 1.9. To install Go, follow the instructions in [3]. Furthermore, `git` should be installed on the host.

Once the Go installation is up and running, create the directory `$GOPATH/src/github.com/dedis` and `cd` into it.

Now type `go get github.com/dedis/student_18_omniledger`. This will clone the repository into your current directory. In order to get the de-

pendencies needed to build the project, run `go get ./...` in the projects root directory. This will automatically download all the dependencies declared in the source code.

The OmniLedger service itself does not provide any `main` method, but tests can be run from the repository's root directory with `make test`.

References

[1] bbolt. `https://github.com/coreos/bbolt`.

[2] The cothority framework. `https://github.com/dedis/cothority`.

[3] Go installation guide. `https://golang.org/doc/install`.

[4] The go programming language. `https://golang.org`.

[5] Omniledger. `https://github.com/dedis/student_18_omniledger`.

[6] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[7] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[8] Bryan Ford. Pseudonym parties: An offline foundation for online accountability. *Unpublished manuscript*, 2007.

[9] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Sandra Deepthy Siby, Nicolas Gailly, Philipp Jovanovic, Linus Gasser, and Bryan Ford. Hidden in plain sight: Storing and managing secrets on a public ledger.

[10] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger. *IACR Cryptology ePrint Archive*, 2017:406, 2017.

[11] Lefteris Kokoris-Kogias, Philipp Svetolik Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. *Proceedings Of The 25Th Usenix Security Symposium*, pages 18. 279–296, 2016.

[12] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[13] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. Chainiac: Proactive software-update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1271–1287, 2017.

[14] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 444–460. Ieee, 2017.

[15] Ewa Syta, Iulia Tamas, Dylan Visher, David Wolinsky, Philipp Svetolik Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Alexander Ford. Keeping authorities honest or bust with decentralized witness cosigning. *2016 Ieee Symposium On Security And Privacy (Sp)*, pages 20. 526–545, 2016.