MadNet

Dr. Christopher Gorman Hunter Prendergast Anthony Dean
Tom Bollich Adam Helfgott

October 29, 2020

TL;DR

Mad Network is verifiable identity of assets, orgs or people in any market or group. The system today allows commercial enterprises to automate trusted transactions with each other based on certified identity of its primitives of assets, orgs and people. MadNetwork's verifiable data structure is similar to a decentralized version of Google's Trillian.

Contents

1 The Problem **4**

2 The Story **5**

3 Audience **7**

4 Paper Organization **7**

5 Canonical Object Encoding **8**

6 Transactions **8**
6.1 Overview 8
6.2 The Transaction Object 9
6.3 The TxIn Object 11
6.4 The DataStore Object 12
6.5 The ValueStore Object 13
6.6 The AtomicSwap Object 13
6.7 Account Abstraction 15

7 Consensus **16**
7.1 Diagrams 16
7.2 Definitions 16
7.3 Cryptography 17
7.4 Storage and Data Persistence 17

7.5 Protocol Overview . . . 18
7.6 The TxRoot . . . 20
7.7 The StateRoot . . . 20
7.8 The HeaderRoot . . . 20
7.9 The RClaims Object . . . 21
7.10 The RCert Object . . . 21
7.11 The BClaims Object . . . 21
7.12 The PClaims Object . . . 22
7.13 The Proposal Object . . . 22
7.14 The PreVote Object . . . 23
7.15 The PreVoteNil Object . . . 23
7.16 The PreCommit Object . . . 24
7.17 The PreCommitNil Object . . . 24
7.18 The NextRound Object . . . 25
7.19 The NextHeight Object . . . 25
7.20 The BlockHeader Object . . . 26
7.21 Validators . . . 26
7.22 SnapShots . . . 27
7.23 Withdrawing From Validation . . . 27
7.24 Leader Election . . . 27
7.25 Virtual Votes . . . 27
7.26 Protocol States . . . 28
7.26.1 PendingProposal . . . 29
7.26.2 ProposalStep . . . 34
7.26.3 PendingPreVote . . . 36
7.26.4 PreVoteNilStep . . . 37
7.26.5 PreVoteStep . . . 38
7.26.6 PendingPreCommit . . . 39
7.26.7 PreCommitNilStep . . . 39
7.26.8 PreCommitStep . . . 40
7.26.9 PendingNext . . . 41
7.26.10 NextRoundStep . . . 41
7.26.11 NextHeightStep . . . 42
7.26.12 RoundJump . . . 43
7.26.13 HeightJump . . . 43
7.26.14 FormNextHeight . . . 43
7.27 Mining Rewards . . . 44
7.28 Slashing . . . 44
7.29 Consensus Proofs . . . 44
7.29.1 Safety . . . 44
7.29.2 Rules . . . 45
7.29.3 Proof of Safety . . . 45

8 Networking **50**
8.1 Overview of Networking Stack . . . 50
8.2 Brontide . . . 50
8.3 Yamux . . . 50
8.4 gRPC . . . 50
8.5 Connection Handshaking . . . 51
8.6 Summary of Higher Order Protocols . . . 51
8.7 BootNode Protocol Summary . . . 51
8.8 Discovery Protocol Summary . . . 51
8.9 P2P Protocol Summary . . . 52

9 Data Structures **52**
9.1 Compact Sparse Merkle Tries . . . 52
9.2 State Trie . . . 53
9.3 BlockHash Trie . . . 53

10 Economics **53**

11 Layer Three **55**
11.1 Scriptless Scripts and State Channels . . . 55
11.2 AdLedger PKI . . . 57

A Transaction Object Specification **61**

B Consensus Object Specification **65**

C Technical Cryptography **68**
C.1 Comparision with Ethereum Distributed Key Generation Paper . . . 68
C.2 Specific Implementation Details . . . 69
C.3 Mathematical Background and Cryptographic Definitions . . . 72
C.4 Distributed Key Generation Protocol . . . 73
C.4.1 Participant Setup . . . 73
C.4.2 Verifiable Secret Sharing . . . 73
C.4.3 Malicious shares . . . 74
C.5 Shared Secret Encryption . . . 75
C.6 Group Signatures . . . 75
C.6.1 Constructing the Master Public Key . . . 75
C.6.2 Constructing Group Signatures . . . 78
C.6.3 Malicious Group Public Key Shares . . . 80
C.7 Hash-to-curve Functions . . . 85
C.7.1 Inverses, Square Roots, and Legendre Symbols in $\mathbb{F}_p$ . . . 86
C.7.2 Inverses, Square Roots, and Legendre Symbols in $\mathbb{F}_{p^2}$ . . . 86
C.7.3 Hashing to Base . . . 87
C.7.4 Base to $\mathbb{G}_1$ . . . 89
C.7.5 Base to $\mathbb{G}_2$ . . . 90

1 The Problem

MadNetwork was conceptualized as a mechanism to address many of the long-standing issues surrounding the modern advertising industry. The number of problems that plague this space are no secret. For instance, it has been estimated that in 2017 nearly 40% of all programmatic impressions were fraudulent [1]. In 2018 that same fraud accounted for more than $19 Billion USD worldwide [1]. These numbers seem unimaginably large already, but fraud is actually increasing at an exponential rate [1]. The problem will get much worse before it gets better.

Although many solutions have been proposed to address this fraud, they have proven unable to make a meaningful impact as of now. A prime example of this lack of innovation rests in ads.txt. This solution was introduced in 2017 [22] and in spite of widespread adoption, the general trend of fraud has done anything but relent.

The apparent void of solutions to this ever-growing problem has caused many companies to turn to any technology that can promise a potential solution. Given the nature of advertising fraud, blockchain seems a rational solution. Billions of dollars are securely transferred every day using these technologies in a transparent and auditable fashion [28]. The reality is, although many solutions work at the pilot level, the success of these solutions is dependent upon the completely unrealistic settings in which these pilots are executed. The core problem any AdTech blockchain solution must face is massive scale. This scale is what lacks in the pilot environment.

Although many blockchain projects have pitched themselves as the savior of AdTech, no solutions have, as of now, addressed the problem in a scalable manner. The repeated theme of many AdTech approaches to blockchain necessitates injecting large volumes of data into a blockchain system with the intent of bringing transparency to the supply chain. These projects seemed to have forgotten the hard-learned lessons surrounding blockchain scalability, or the inherent lack thereof [5]. Another often cited mechanism of revolutionizing the AdTech space through blockchain involves moving the real time bidding process into smart contract systems. The promise of such systems is to remove the middlemen of the supply chain while making the open bid process more transparent. Although noble in intent, these companies failed to realize that distributed consensus takes time. Blockchains are unable to produce blocks fast enough to meet the sub-second demands of AdTech, not to mention the millions of requests per second [20]. The good news is AdTech can benefit from blockchain by building systems that address the heart of the problem.

At the core of advertising, fraud is a very old problem. A problem that has been partially addressed many times in many different ways. That problem is data authenticity and integrity. The most recent proposal of ads.cert [9] reflects this reality. In order to prevent fraud, the information being exchanged in the real time bidding supply chains of modern AdTech must be protected against manipulation and this information must have provable origination. The IAB has unfortunately failed to see the inherent flaw in the system they have proposed. Similar to the fact that AdTech blockchain companies are still largely struggling with the modern realities of the very technology they depend upon, the specification fails to adequately cover the ever growing subset of the advertising market that is not directly attributable to a web domain. Any solution that hopes to be more than a band-aid for the current problems of AdTech must operate in a manner that will be viable for mobile applications, over-the-top streaming services, and even mediums we have not as of yet

contemplated. Further, these solutions are needed today. By writing ads.cert into openRTB 3.X and not providing backward compatibility, adoption will be significantly delayed.

2 The Story

MadNetwork has progressed through many iterations to bring the project to where we are today. In each of those iterations, we learned what could work and arguably, more importantly, what would never work. In an early pilot of the technology we leverage the Ethereum blockchain as a means to rapidly iterate design patterns. Although the Ethereum network is an outstanding piece of technology that will forever change the world, we kept finding the sharp edges while using the technology in the enterprise setting. The costs of storage, the inherent limitations on throughput, the complications around cleaning up stale state, and the complexity of integrating enterprise partners into the technology all caused complications. These efforts led us to research other systems that addressed the concerns we were experiencing. What we found was that these problems are by no means isolated. In fact, what we found was that the problems were fairly universal.

This early pilot project was a primitive form of our end goal. That goal was to build a better public key infrastructure (PKI). Given an application specific PKI, all members of the advertising supply chain can be assigned strongly unforgeable identities. By not linking the resolution of these keys to a presupposition that all members of the AdTech supply chain must have a registered internet domain, we open the possibility of using this system to more technologies. These identities may be used for many purposes, but chief among them is proofs of integrity and authenticity for messages communicated through third parties. In this way, we may prevent many forms of fraud and gain insight into the supply chains themselves. These systems also allow the negotiation and/or distribution of encryption keys between parties. In the context of advertising this means we can build broadcast encryption technologies that allow private data to be streamed through openRTB such that only the intended party/parties may decrypt it. This capability is sorely needed due to the recent industry transformations being forced by GDPR, CCPA, and other privacy-centric legislation. Additional capabilities are also possible due to the inclusion of pairing-based cryptography. These include Hierarchical Identity Based Cryptography for the use of securely sharing identities between partners such that external groups are not needed for matching operations. In summary, this system also allows network participants to communicate private data in the oRTB environment, as well as communicate identity of the user in the bid request to opted-in recipients.

Although x.509 stands as the de facto standard for enabling enterprise encryption and server authentication, this system is built on antiquated technologies that have been the root of many problems in the past two decades. Further, x.509 is fundamentally ill-equipped to perform all but the most basic of what was needed to fully address the problems of the AdTech industry. Thus, a new system is needed to combat the concerns of AdTech. This is once again why the IAB has begun work on ads.cert. Asymmetric cryptography would appear to be the best solution we have to protecting digital commerce. This should come as no surprise. What should be surprising is that a high tech industry such as AdTech has not managed to implement a solution already.

In order to build such a PKI using the Ethereum blockchain, smart contracts were leveraged that allowed a root of trust to associate cryptographic identities with real world businesses. This decentralized design meant that the cryptographic identity of any registered business could be referenced in a fully transparent system, without the need to contact our systems in the process. The other benefit to such a design was that unlike x.509 where revocation and transparency have been a pain point for years, we inherited these properties by default from day one. This system also afforded the ability for any registered entity to register new cryptographic keys, for many different applications, with no interaction from our systems outside initial registration. This system even allowed those safe keys to be revoked by simply removing them from the blockchain.

What we were building was analogous to modern technologies intended to shore up x.509 against rampant fraud that occurred in the last decade. Specifically, technologies such as Certificate Transparency. Unfortunately, simply using the core concepts of Certificate Transparency still did not address the issue of revocation. Considering the vast fraud that already plagues the AdTech industry, any PKI solution that did not accommodate efficient revocation and scale to millions of requests per second was a liability, not a solution. This inability to scale to millions of requests is the very failure of OCSP, another x.509 technology that was intended to address revocation. Fortunately, there does exist a mechanism to address the problem of revocation. This solution will be covered during the formal discussion of the PKI we have built.

Although the Ethereum system did work for the purposes of a pilot project, bringing thousands of new enterprise systems into the fold would eventually prevent any solution built on this technology from succeeding due to the scaling constraints of the underlying system. What we needed was a system that was built for the specific purpose of creating a better PKI. What we needed was a cryptographically verifiable map that could expire stale records automatically, was capable of sharding so that it could grow smoothly with system utilization, was inexpensive to write data into, and had mechanisms through which tokens could be abstracted for enterprise partners. What we needed was MadNetwork.

Unfortunately, such a system did not exist at that time. The lack of such a system seems astonishing given the trends in enterprise adoption of blockchain technology. Many enterprise projects do not require the full complexity of a smart contract enabled blockchain. Further, many enterprise projects require that data not be forever immutable, such as that mechanism that is afforded by Bitcoin's OpReturn method of writing data.

In order to accommodate the above challenges, while also acknowledging that creating strong consensus algorithms is a nontrivial task, MadNetwork has been built as an Ethereum sidechain utilizing a Proof of Stake consensus algorithm. By anchoring MadNetwork into an Ethereum smart contract system, those same properties that otherwise make Ethereum a difficult technology in the enterprise setting may be leveraged for the benefit of creating a more secure system. These benefits include the ability to codify complex governance and fail-safe mechanisms that would otherwise be incredibly difficult in the stand-alone Proof of Stake setting.

MadNetwork has been designed to facilitate building a fully-auditable PKI that allows for efficient revocation as well as compact proofs of certificate non-revocation. This system has also been built to allow proofs of the state of the sidechain to be verified within the Ethereum virtual machine. Our hope is that, outside of the industry partners we are already

building solutions for, other members of the Ethereum ecosystem may also leverage this technology to support their own projects. If this were to occur, it would allow what is the effective equivalent of smart contract state cold storage to be leveraged within the Ethereum blockchain. This would ultimately help address some of the very scaling issues we experienced while using Ethereum in the past.

3 Audience

This paper is intended for consumption by individuals that have a non-introductory understanding of blockchain systems. Although the best effort has been made to allow the contents of this paper to be communicated in a comprehensible manner, a full description of all fundamental concepts has been omitted for brevity. Thus, the reader should understand the general operation of blockchain systems. Specifically, the reader should be familiar with UTXO-based systems such as Bitcoin and account-based systems such as Ethereum. A complete understanding of cryptography is not necessary for the main body of this document, but an understanding of digital signatures is required. Knowledge of cryptographic hash functions is assumed. Further, the reader should be aware of threshold cryptosystem capable signatures such as BLS [6]. The academic treatment of these operations has been left for the technical appendices.

4 Paper Organization

This paper has been organized as a bottom up approach to the formation of our system. We have elected to explain the operation in this way because the system has been engineered to be largely ignorant of the operations at layers above and below a given primitive. Thus, our system allows application logic to change while preserving the consensus mechanisms. The system also allows the consensus mechanism to be changed while preserving the application logic. Due to this design, the system is most easily digested by building it in layers of complexity.

The first sections pertain to the encoding of messages within the system. The next section addresses the operation of transaction processing. The section following gives insight into the synchronization and how history may be safely pruned from the system. Following these sections the consensus algorithm itself is covered in full detail. The networking stack is then addressed to describe the communication between nodes, based on the consensus algorithm. Finally, the economics of the system are covered.

The treatment of these topics is not performed in a technical manner in the body of the paper proper, aside from the proof of the consensus algorithm. This choice was made because the consensus proof is far less technical than the cryptography of the system, and may be understood by the intended audience of this paper. The formal treatment of cryptographic operation has been performed in the technical addendum of this paper; see Appendix C.

5 Canonical Object Encoding

As with any blockchain system, a canonical encoding of objects and messages is required to allow validation of cryptographic signatures across systems. Rather than focus time on solving the problem of building a canonical encoding, we elected to select an existing robust encoding mechanism. This allowed us to focus more time on difficult problems that are more relevant to our direct intent. This also allowed the system to be replicated in other programming languages more readily by selecting a cross language encoding mechanism.

In order to achieve these goals, Cap'n Proto has been selected as the canonical serialization format. For those readers who may be unfamiliar with Cap'n Proto, this protocol was built by the same engineer that designed Protobufs for Google. The most important difference between Protobufs and Cap'n Proto, for the MadNetwork application, is the fact that all Cap'n Proto objects have a canonical encoding that is preserved even under additive changes of the object definitions [25]. This protocol has been ported to many languages and has been used in production systems by enterprise companies such as CloudFlare [26]. This protocol is only used for the serialization of objects and our system does not integrate any of the RPC mechanisms associated with the Cap'n Proto specification.

The selection of this encoding scheme is important to the following sections. It influenced the manner in which objects have been constructed to reflect the operation of the encoding scheme. This may be seen in the composition of objects to allow complex cryptographic proofs to be formed in an elegant manner.

6 Transactions

6.1 Overview

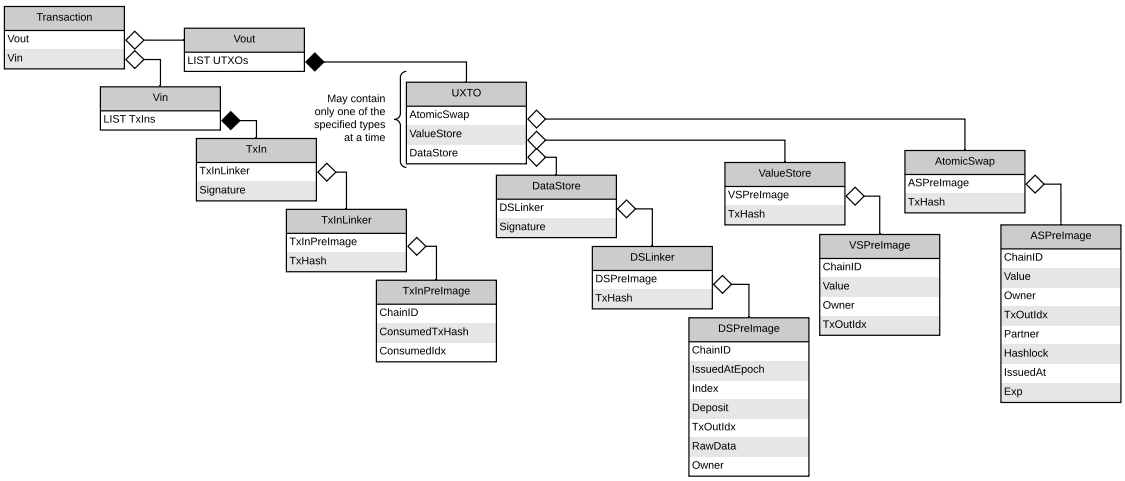

Figure 1: Transaction Object Primitives

MadNetwork operates on a UTXO model that has many similarities to the Bitcoin blockchain. The divergence from the Bitcoin system is based on an attempt to simplify the rules

surrounding transaction validation. Rather than integrate a scripting language, MadNetwork uses object specific validation logic. What this allows is a more efficient verification algorithm of pending transactions as well as more granular proofs with respect to blockchain state.

In the MadNetwork system, version numbers are not required at the transaction level. In the event that additional functionality is desired, new objects may be created to accommodate such changes and the validation of such objects may be treated as largely independent from the validation logic of prior assumptions. Thus updates to the protocol are purely additive modifications. This choice was made based on the desire to build a system that is resilient to the fact that it will require many years of continuous uptime.

6.2 The Transaction Object

A transaction is composed of two top level fields. These top level fields are vectors of objects. The first vector, Vin, contains a vector of TxIn objects. The second vector, Vout, contains a vector of UTXO objects. A UTXO object must contain only one element from the mutually exclusive set of allowed UTXO types. These vectors are limited to 256 elements each and a transaction is limited to a total of 3 megabytes max. TxIn objects and UTXOs will be more fully covered in the following sections.

The operation of concern at the Transaction Object level is the mechanism by which a transaction hash may be formed. As may be seen above, every element of Vin and Vout contains a PreImage type as the last element in the object hierarchy. The PreImage objects define the protected parameters of a transaction. In order to ensure that the process of transaction signing is secure, each signed object must be immutably bound to the collection of input and output objects. This binding is done through the process of PreImage hashing. The hashing of PreImage objects allows the formation of the TxHash value. The full details of this operation will follow shortly.

Once a TxHash has been formed it may be injected into the appropriate layer of all Vin and Vout objects. For all signed objects, the TxHash is injected into a Linker object. The canonical encoding of these Linker objects are what is signed for proof of possession of an identified public key as the owner of a UTXO. For all unsigned objects, the TxHash is injected into the top layer of the object hierarchy. This ultimately ensures protection against separation of the signature from the intended action of the signer while also protecting against the known transaction malleability attacks found in early versions of the Bitcoin Protocol. In order to provide more clarity around transaction hashing, the operation, in detail, follows. This example will utilize the TxIn object seen in the diagram above and speak about UTXO objects in a generalized manner.

Before an explanation of the mechanics of TxHashing may be explained two operations must be defined. First, the concept of a UTXOID shall be explained. Then the pseudoUTXOID shall be defined.

A UTXO may be identified by its UTXOID. A UTXOID is created by hashing the concatenation of the TxHash that creates the UTXO and the Vout index at which the UTXO was created. MadNetwork enforces the assumption that there may never be two UTXO objects that exist at the same time that are referenced by the same UTXOID. This requirement is observed through the use of the StateTrie, which is covered more thoroughly

in other sections of this paper. More formally:

$$\begin{aligned} \text{idxBytes} &:= \text{uint32ToBigEndianBytes(idx)} \\ \text{UTXOID} &:= \text{Keccak256(txHash}||\text{idxBytes)} \end{aligned}$$

The pseudoUTXOID, for a given UTXO, may be formed as follows:

$$\begin{aligned} \text{idxBytes} &:= \text{uint32ToBigEndianBytes(idx)} \\ \text{preImageHash} &:= \text{Keccak256(PreImage)} \\ \text{pseudoUTXOID} &:= \text{Keccak256(preImageHash}||\text{idxBytes)} \end{aligned}$$

The operational explanation of TxHash formation may now begin. Each TxIn object is composed of three layers. The first layer is the TxIn object itself. The first layer carries two fields. The first field is the TXInLinker and the second field is the signature of the canonical encoding of the TXInLinker. The second layer is the TXInLinker. The TXInLinker provides an intermediate layer such that the transaction hash may be injected into the object. The second layer contains two fields. The first field is the TXInPreImage and the second field is the TxHash. The third layer is the TXInPreImage. This object codifies the UTXO being consumed through inclusion of the ConsumedTxHash field and the ConsumedTxIdx field. This hierarchy is also constructed on the UTXO objects found in Vout.

This topology allows for each PreImage layer of the transaction inputs and outputs to be hashed concurrently. Each of these hashes is then inserted into a Compact Sparse Merkle Trie at a location defined by the type of object. The resulting CSMT root hash is the TxHash. For TxIn objects, this location is the UTXOID of the consumed UTXO and the value is the hash of the TxIn PreImage. For TxOut objects, the location is the pseudoUTXOID and the value is the hash of the object PreImage. Because a TxHash is not available to form the proper UTXOID of the object, a pseudoUTXOID is used instead.

Once the TxHash is computed, each object from Vin and Vout has the TxHash injected and all objects that must be signed are signed after this injection. This provides the means to immutably link a signature to a TxHash.

Unlike in Bitcoin, the sum of a Transactions input values must be equal to the sum of a Transactions output values. At this time, transaction fees are paid through the cleanup of stale state. Thus no direct mechanism to pay transaction fees is provided except through explicitly naming a given validator or sending a UTXO to the validator set using the validators Group Key. The Group Key is a negotiated BLS key under a threshold cryptosystem. This threshold cryptosystem will be explained more fully in the consensus algorithm description. Due to this threshold cryptosystem, the group of validators may coordinate to distribute mining fees sent to the Group Key. The authors of this paper are aware of the potential for an attack against the storage of the system due to this design choice. This attack is similar in nature to the DoS attacks on Ethereum that occurred due to malicious transactions eating computation time. In future versions of the protocol this attack may be trivially mitigated through several mechanisms. In the current implementation these attacks may be mitigated as well. This protection is as follows.

If it is observed that an attack is ongoing, the validators may refuse to mine any transaction that does not transfer value to the validator set. This is possible through the use of threshold cryptography that will be covered in the consensus section of this paper. The option that may be included in future versions is to allow a UTXO to be formed that grants the miner the ability to consume the reward directly or inject a key so the UTXO may be consumed later. This has not been implemented at this time in order to simplify the process of transaction processing. Currently, the system mandates that all consumed objects must exist prior to a transaction. The system also requires that all transactions in a block do not reference any object that any other transaction has already consumed. Lastly, all objects in all transactions of a block must exist prior to the block being mined. These assumptions protect the atomicity of state transitions, but this atomicity may be preserved with more complex handling of transaction fees as an edge case.

6.3 The TxIn Object

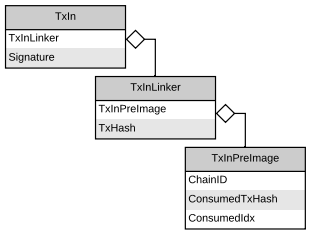

Figure 2: TxIn Object

A TxIn object allows a UTXO to be consumed by specifying the TxHash of the Transaction in which the UTXO was formed and the index at which the UTXO was created in the Vout vector. All TxIn objects must be signed in order to prove possession of the private key generated during object creation. MadNetwork allows several pending transactions to reference the same UTXO for consumption at a time through the utilization of an LRU-based reference counting system. A TxIn may not reference a UTXO created in the same transaction or block. As previously covered, this may be changed in future versions to allow the edge case of mining fees. All TxIn objects must point to either a known UTXO or a valid unspent deposit from Ethereum.

In addition to consumption of MadNetwork UTXO objects, the TxIn object may also cause the consumption of a Deposit from the Ethereum blockchain. Deposits into the sidechain may be triggered by any token holder on the Ethereum blockchain. These deposits become virtual UTXOs in the sidechain after a ten block wait. This process is controlled by a deposit smart contract on the Ethereum blockchain. Internal to this deposit contract there exists a monotonically increasing counter. For each deposit, this counter is incremented. When a deposit is made, a hash is passed into the contract. This hash is the hash of the public key that may be used to claim the deposit on the sidechain. The deposit itself will cause the associated tokens to be burned on the Ethereum blockchain. This action causes an event to be emitted that is observed by the miners of the sidechain. Once this deposit

has matured for at least 10 blocks, it will become available for use in the sidechain. The deposit may be consumed by referencing the deposit in a TxIn object. This reference may be formed by setting the consumedTxIdx field of the TxIn object to the max value of uint32, setting the txHash to the big endian uint256 equivalent of the counter at the time of the deposit, and signing with the same private key as was used to create the public key hash specified in the deposit contract invocation. In order to prevent a deposit from being double spent, a permanent record of having consumed the deposit is tracked in the State Trie of the sidechain. This tracking of state is permanent and allows the validators to not perform any acknowledgment operations of a deposit having been spent against the deposit smart contract.

6.4 The DataStore Object

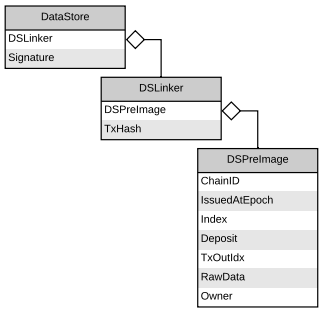

Figure 3: Data Store Object

A DataStore object is a type of UTXO. This is the fundamental mechanism by which data may be stored in MadNetwork. Rather than storing data in an account-based relationship, we selected the use of a UTXO-based storage system to allow for a more traditional atomic commitment mechanism as is readily available in database technologies in general. This capability allows a single transaction to consume many DataStore objects and rewrite these objects in an atomic manner. A DataStore is bound to a single epoch, where an epoch is a bounded division of blocks in MadNetwork. These divisions are deterministic and based on simple modulo operations. The full specification of an epoch will be handled later in this paper. Any transaction containing a DataStore must be included in the named epoch of generation. This constraint is manifest through the rent-based data storage mechanisms employed by MadNet. In addition to the actual data being stored, the deposit, the chainID, the output index, the epoch of issuance, and the owner, each DataStore contains an index. The index allows a datastore to be referenced as a named element in the virtual namespace defined by the hash of the public key of the DataStore signer. This index is built through a reference system in the database behind the application, and this reference system enforces uniqueness of an index in a given namespace. This reference system allows $O(1)$ access to an element in a namespace. There also exists an index that allows all objects in a namespace to be traversed in $O(n)$ and $O(\log n)$ search may be implemented in future versions of this system without impacting consensus.

6.5 The ValueStore Object

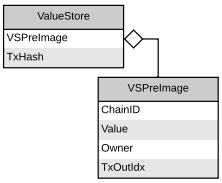

Figure 4: Value Store Object

A ValueStore allows for the conveyance of MadNetwork tokens between accounts in the MadNetwork and provides a mechanism for generating a change output from a transaction. The named owner is the only party that may consume a ValueStore object. All value stores are also indexed according to the owner by default. This allows iteration of UTXO objects for the purpose of simplifying wallet software. These objects are indexed in either smallest value to largest value and may be iterated in forward and reverse. This indexing allows efficient solutions of the knapsack problem using greedy algorithms. These greedy algorithms are not included in the logic of the node at this time. The inclusion of this index is intended to facilitate the minimization of UTXO dust, as it is named in the Bitcoin context.

6.6 The AtomicSwap Object

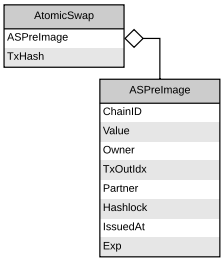

Figure 5: Atomic Swap Object

Withdrawals from the sidechain occur through atomic swaps only. This decision was made to dramatically reduce the complexity of the protocol. Since the primary utility of the underlying system is to write data into the sidechain, and tokens are destroyed when such an event occurs, this simplification reflects the intent of the system and allows the amount of state that must be tracked to be dramatically reduced. In this way the sidechain acts as a universal sink in the flow of tokens.

The manner in which atomic swaps are facilitated is through the native AtomicSwap object. This object acts as a time-locked hash contract where time is measured in epoch boundaries. The safest manner in which to negotiate an atomic swap is for the sidechain party to be the initial actor. This pattern is also convenient because it allows an offer contract to exist in the Ethereum EVM that orders may be matched against, and MadNet was intentionally not built for complex smart contract operations of this nature.

At the time of creation, any value stored in the AtomicSwap is locked. During the time between issuedAt and exp, the partner may claim the value stored in the AtomicSwap by revealing the preImage of the hashlock. The only party that may claim an AtomicSwap before exp is the partner and only with a proof of knowledge of the hashlock-preimage. If the AtomicSwap is not consumed before exp, any value stored in the AtomicSwap will revert to owner. Thus, only the partner or owner may consume this UTXO type.

The operational flow is as follows. For the purpose of this explanation Mike will hold MadNetwork Tokens in the MadNetwork itself and Erin will hold Ethereum in the EVM. The protocol may begin once Mike and Erin are aware of each other's desire to exchange, an exchange rate has been set, Mike has the hash of Erin's public key that she will use in MadNetwork, and Erin has Mike's Account he will use in Ethereum.

First, Mike will form a Transaction that transfers the required value into an AtomicSwap object on the MadNetwork and sets the partner value to the hash of Erin's public key. Mike will set the exp at least three Epochs into the future, and Mike will hash a random value to act as the preimage of the hashlock. The random value Mike selects must be a 32 byte object. This constraint is enforced by the MadNetwork AtomicSwap object as well as by the Ethereum AtomicSwapContract. This requirement is to prevent maliciously large values from being selected.

Mike may then inform Erin of the transaction hash, or Erin may watch for a transaction with her public key hash. Once Erin observes the transaction, she may form an Ethereum transaction by calling the AtomicSwapContract in the EVM. This contract should have the same hashlock as Mike's transaction and the expiration of this offer should be at least one Epoch shorter than the exp used in Mike's transaction. Note that if Mike chose an expiration that is less than three epochs into the future, Erin should abort the protocol and not form the offer in the AtomicSwapContract. A failure to observe this requirement may result in the loss of assets due to a race condition. Specifically, Mike could wait until immediately prior to expiration of the AtomicSwapContract object to reveal the hashlock. If both the AtomicSwap object and the AtomicSwapContract expire at the same time, there are no guarantees Erin would be able to submit a transaction before the expiration of the AtomicSwap object. This would allow Mike to recover Erin's funds without Erin recovering anything from Mike.

Pending that both Mike and Erin have constructed the appropriate objects with all constraints met, the finalization may now commence. First, Mike must claim his value in the EVM by sending a transaction from the specified address that he previously claimed. In this transaction Mike must reveal the preimage of his hashlock in order to claim the funds. Once Erin observes this transaction she will know the hashlock preimage. Given the hashlock preimage Erin may claim her value in the MadNetwork by forming a transaction that spends the AtomicSwap object.

In the event that Mike creates the AtomicSwap object and Erin does not reciprocate or forms the AtomicSwapContract with incorrect parameters, Mike may recover his funds after the AtomicSwap object has expired. Similarly, if Mike forms the AtomicSwap object with incorrect parameters or never claims the funds from the AtomicSwapContract, Erin may recover her funds after the expiration of the AtomicSwapContract.

6.7 Account Abstraction

Transaction signing in MadNetwork may take several forms and the signature itself may be generated under two allowed Elliptic Curves. The first curve that may be utilized is Secp256k1. Every Secp256k1 signature must be an Ethereum-compliant ECDSA signature. The second curve is the BN256 Elliptic Curve as specified in the Ethereum Yellow Paper. All BN256 signatures must sign transactions using BLS with the defined hash to curve operations as seen in the technical addendum. These BLS signatures require the public key of the signer to be concatenated with the value being hashed. This does allow safe aggregation of the signatures, but this ability is not leveraged at this time. BLS signatures must also prepend the public key to the signature itself for validation purposes. In the event that a multisignature account is desired, BLS signatures may be used with off chain coordination of key negotiation. Lower setup complexity of BLS multisignatures may be included in the future with ease.

In order to build the system of account abstraction, the owner field of an object indicates what curve is being used by setting the first byte of the owner field. Each object type defined above also corresponds to a single signature verification algorithm per curve that may be used to sign each object. This strategy allows for four signature verification operations at this time and may be extended to many more as necessary for additional functionality.

The first form is a public key hash signature verification algorithm. This type of signature verification requires that the signature be valid per the specified curve and data, and that the hash of the public key that generated the signature matches the owner field from byte one to end using zero index counting. This algorithm is the required algorithm for all value store objects at this time, and may use either BLS or ECDSA type signatures. This operation is only allowed for the ValueStore objects at this time.

The second form is a hashed timelock signature verification algorithm. This signature verification algorithm is only allowed to be used with ECDSA at this time and this signature type is only allowed to be used with the AtomicSwap object at this time. In this signature verification algorithm, the hashlock preimage must prepend the signature.

The third form is a transparent signature verification algorithm, as is required by a datastore object. This signature verification algorithm requires that the owner field is itself a signature and that the signature that consumes the object with this signature verification type be a different signature using the same public key as the original signature. This algorithm allows both ECDSA and BLS type signatures at this time. In the event the signatures are BLS signatures, the public key must be prepended to the signature. The ECDSA implementation used is the Ethereum compliant recoverable ECDSA, so the public key is not required to be prepended. This signature verification mechanism is the only type allowed for DataStore objects at this time and this mechanism may only be used with DataStore objects. The reason for requiring this mechanism is to allow the proof of possession of the signing key at the time the object is constructed. This is required due to the namespaced indexing that DataStore objects enforce. For instance, Bob may not write into the namespace of Alice without knowledge of the private key of Alice.

7 Consensus

7.1 Diagrams

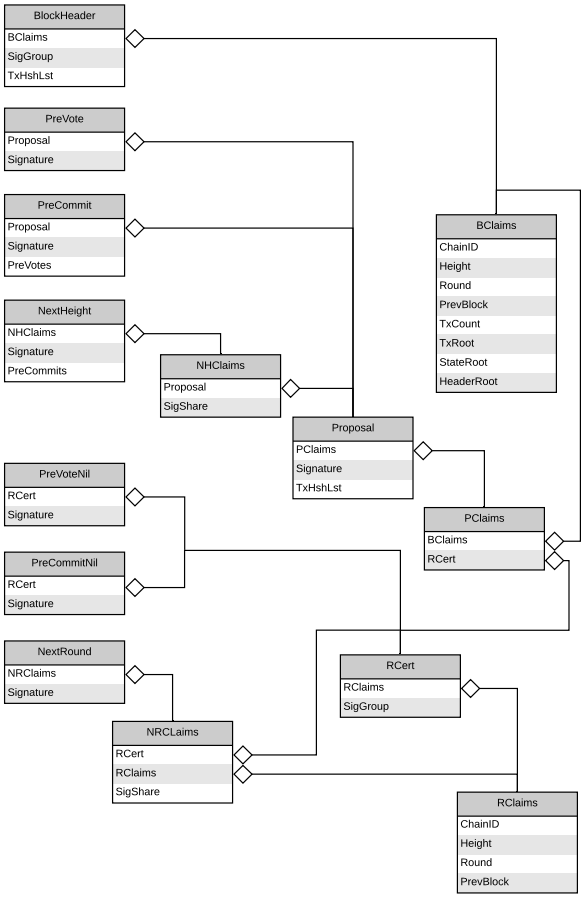

Figure 6: Consensus Object Primitives

7.2 Definitions

Validator A validator is a node that has placed a stake in the Ethereum staking smart contract, successfully negotiated a group key, and is actively participating in block creation on MadNetwork. Validators coordinate the generation of new blocks through a Byzantine Fault Tolerant consensus algorithm.

Height The number of blocks that are in the blockchain starting from index one.

BlockHash A cryptographic hash of the canonical encoding of the BClaims object.

ChainID A unique uint32 value used to identify a particular instantiation of the MadNetwork blockchain software.

Round — A round is defined by an iteration of the consensus protocol in which a single validator may be considered the leader. For each change of leader that occurs without a new block being produced, the round is incremented. The round starts at index one and is monotonically increasing. Upon a new block being produced, the round is reset to one.

SigShare — This is a signature of an object under a BLS threshold cryptosystem.

SigGroup — This is the aggregated signature formed by a threshold number of SigShares.

Signature — Signatures are validator signatures under the secp256k1 curve using the Ethereum compliant recoverable ECDSA algorithm.

7.3 Cryptography

We use Elliptic Curve Cryptography within our system. Specifically, we use the curves Secp256k1 and BN256-Eth. Secp256k1 is the same elliptic curve used by Bitcoin and Ethereum for their public key cryptography. BN256-Eth is the pairing-friendly elliptic curve from the Ethereum blockchain which is convenient for constructing group signatures. All validators are required to have an Ethereum account and so they have a Secp256k1 public key. During the distributed key generation process, a master public key (a public key for the entire group) is constructed and split amongst the participants. Each member is able to compute a partial signature, and these partial signatures are able to be combined into a group signature. A more in-depth discussion about cryptography is relegated to Appendix C.

7.4 Storage and Data Persistence

In order to ensure all information is stored in a system that is efficient, crash resilient, and allows transactional updates, BadgerDB was selected as the backing storage mechanism for the system. The consensus mechanism may be built on other key value stores in a similar manner, but the choice of Golang as our programming language made BadgerDB an optimal choice for an embedded key value store. BadgerDB also supports PubSub systems, allows namespacing of objects through key prefixes, ordered/key only iteration, and many other useful primitives.

In the context of the consensus algorithm, the crash resilience and PubSub mechanisms have been leveraged to simplify the problems surrounding accidental violations of validator protocol. These accidental violations are most likely to occur in other systems when a crash occurs as a value is being written to a database and also being transmitted to peers. Such crash conditions could allow an otherwise honest party to act in a contradictory manner through nothing more than a crash problem coupled to a race condition. In order to mitigate such problems, we leverage the PubSub system of BadgerDB to send messages out to peers only after they have been verified as having been flushed to the file system. This decoupling also allows the simplification of the consensus algorithm logic by not requiring complex locking and chaining of operations.

Future versions of the node software will include the ability to coordinate data persistence in a master slave failover architecture for validator nodes. This will mitigate the previously

described race condition under the extended assumption that multiple validators are being run in a failover configuration and both validators use the same signing key. This race condition caused the first slashing event on the Cosmos Hub, so it is worth addressing. Fortunately the same PubSub system may be leveraged to facilitate this action painlessly. This may be implemented by forcing the master to write each message to the database at a location that causes the PubSub system to send a message to the slave. The system may then wait for an ack response from the slave that specifies the key of the value that has been persisted to disk. Another thread in the master process may then write this same value to a location that is monitored by the PubSub system for transmission as gossip to other peers. Through this simple modification a single master slave configuration may be built that ensures atomic coordination between the master and slave while sacrificing a small latency in network operation.

7.5 Protocol Overview

The MadNetwork consensus protocol is based on a modified form of Tendermint. In our implementation we utilize the PaceMaker concept of HotStuff in order to create stronger bounds on the mean time to consensus. This is implemented as the RCert system described below. Unlike Tendermint, we do not allow transitions between rounds to occur as the default response to a timeout. In order to exit a round, the system requires at least threshold messages have been observed in each intermediate state. A validator may also exit a round or height, with constraints, if a proof of consensus is observed for a higher round/block. The observation of these threshold messages allows the formation of a group signature under a threshold signature scheme. This leaves the potential for halting problems to arise under double proposing in a round, but we have addressed the halting problems through a novel concept we have named virtual voting. The flow of this protocol is also optimized through an optimistic fast path that allows termination of the protocol in slightly more than a single timeout under general consensus. The system does, however, force slower operation under split votes in order for more state information to be observed before forward progress is made.

This system is operated as a Proof of Stake sidechain to the Ethereum Blockchain. This network is governed and secured through the use of smart contracts that exist in the Ethereum Blockchain. This choice of operations allows the system progress to be governed by a system that is beyond the control of the validators in the Proof of Stake protocol itself. Thus, many complexities around malicious majority attacks may be mitigated either completely or at least in part.

The system operates on a hierarchical division of time. The measurement of this time is relative to two systems. The first system is the Ethereum blockchain. The system protects some actions from occurring until a threshold number of blocks have occurred in the Ethereum blockchain to prevent attacks that attempt to violate required delays of the system through pre-arranged message negotiation in secret. The full details of these attacks are not covered because they have been mitigated, but the basis for these attacks is apparent in the section describing validator stake withdrawal operations.

The second system of time measurement is local blocks within the Mad Network itself. Large groups of these blocks form Epochs. An epoch is defined for two purposes. First, it

designates a boundary point for recording a snapshot of blockchain state into an Ethereum smart contract. Second, it provides a logical transition point for the exit and entrance of new validators.

In order to build the protocol in an auditable manner, a synchronous core algorithm was designed such that asynchronous messages may be queued for processing. The primary control loop for the consensus algorithm is a three state system. The system begins in the unsynchronized state. Once synchronization is complete, the algorithm alternates between updating local state based on messages observed and collecting messages for processing and storage.

All network interaction is decoupled from the consensus algorithm proper through the use of BadgerDB's native PubSub mechanisms. Thus, a message may never be placed on the wire without first being recorded in the database. In the following explanation network communication should be assumed to occur following writes to persistent storage.

Within our architecture each validator maintains a set of linked lists. These lists are stored in BadgerDB using transactional commitments. This architectural decision was made in order to provide strong protection against an inadvertent double signing by honest validators.

The validator lists are maintained such that every known validator has a single list constructed that will remain in the database for two epochs past when a validator is no longer active. The local node maintains an identical linked list for its own state as well. The collection of linked lists is instantiated prior to a validator being active or on first detection of a change of validator set. Each linked list stores a RoundState object that contains a field for each vote type that is possible in the consensus algorithm. As new rounds occur, a new RoundState object is built and the previous object is pushed back on the list. In this way the evidence log and the state storage system may be combined in an elegant manner. The operation of these queues is as follows.

For each new message received, the message RCert height is compared with the currently synchronized maximum block height of the local node. If the block height is greater than one less than the currently synchronized block height, the message is dropped. If the message passes the height check, the entire message is checked for consistency of formatting and cryptographic signatures. If any error occurs at this time, the message is dropped. After this initial validation, the message is dispatched to the consensus algorithms core handler logic.

The first step of additional validation is performed by loading the remote validator's RoundState. If the element for the message type received is newer than the received message, the message is dropped. If the message is equal in height and round to the element, a consistency check is performed. If this check fails, it is an indication of a double vote for a given height/round and thus the value is stored in a way that allows an accusation to be formed. In this way duplicate messages are dropped, and contradictory messages of equal height are also caught. In addition to the validation of contradictory messages internal to a message type, the message is then checked for consistency with the other message types. These checks include ensuring that a validator does not PreVote and PreVoteNil in the same round.

Pending the message passes all verification, the message is stored. In addition to updating the RoundState of the peer that signed the message, the local nodes RCert may be

updated in the next round of local state updates. Specifically, if the RCert of the message is greater than the locally known RCert and the locally known RCert may be validated as not conflicting with the peer submitted RCert, an update will occur. This operation is what drives round/height jumping in the system and prevents deadlocks that may otherwise occur. If the peer submitted RCert is of a height greater than the local RCert plus one, the message is stored in the state space of the peer, but the RCert is not stored in the state space of the local node. In addition to not storing the RCert in the local RCert object, a signal error is raised causing the local node to change state into synchronization mode. If the RCert height is only one block height larger than the locally known RCert, the local node checks the PrevBlock field of the RCert to see if the associated proposal is known. If this proposal is known, and the local validator agrees that this proposal is valid, the validator may proceed to the block height associated with the observed RCert.

7.6 The TxRoot

The TxRoot field is a member field of the BClaims object. This field is the root hash of a Compact Sparse Merkle Trie. The trie is built by first constructing the TxHash values for all transactions that will be included in the Proposal that contains the given BClaims. The objects are inserted at the location key that is equal to the TxHash value and the value stored at this location is the hash of the TxHash itself. This mechanism was selected to build the trie such that it formed an unordered set for simple proofs of inclusion. This design choice does mandate that all transactions that use this mechanism must be strongly independent from each other with respect to modified state. The purpose of this field is to bind the set of transaction hashes that are associated with a blockheader into a compact form of encoding that allows for efficient proofs of inclusion and exclusion.

7.7 The StateRoot

The StateRoot field is a member of the BClaims object. This field is equal to the root hash of the StateTrie after the transactions associated with the Proposal that contains the BClaims has been processed. The purpose of this object in the BClaims is to allow for snapshot synchronization of all UTXOs from a known good value. This allows clients to download only the data that is associated with a snapshot as the start point of synchronization. Once a trie that represents a snapshot is fully synchronized, a client may begin synchronizing block headers and transactions. A full explanation of this process will be addressed later.

7.8 The HeaderRoot

The HeaderRoot field is an object of the BClaims object type. This field is the root hash of a Compact Sparse Merkle Trie that acts as an append only log. The keys in this trie are the big endian uint256 value of a block height. The values of this trie are the associated block hash as determined by the key of insertion. Although this choice of insertion does destroy some efficiency of the underlying Sparse Merkle Trie, this choice was made with the intent of optimizing Merkle Multi Proofs across ranges of the trie. The purpose of this field is to allow a peer to request a block header for a specified block number along with a proof of inclusion.

The basis for this action is to allow a client to request a block header from a known good snapshot root hash with strong unforgeability of the underlying data. This design choice, with respect to ordered insertion at the key equal to the block height mandates that the first block of the network start at index one.

7.9 The RClaims Object

RClaims is an abbreviation for the name Round Certificate Claims. The RClaims object contains four fields that uniquely define the current state of the consensus algorithm. These fields are height, round, chainID, and the blockhash of the previous block. An RClaims object is never transmitted outside of an RCert object.

7.10 The RCert Object

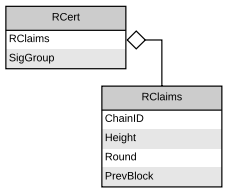

Figure 7: RCert Object

RCert is an abbreviation of the name Round Certificate. An RCert object wraps an RClaims object and contains a signature of the canonical encoding of the RClaims object. The purpose of the Round Certificate object is to allow compact proof of consensus for a given height and round.

In order for a Round Certificate signature to be treated as valid, the signature must be a valid group signature under the GroupKey of the corresponding validator set for the height specified in the Round Certificates RClaims object. In round one of a given height this is the SigGroup of the BClaims object from the block header. This is the same requirement of a BlockHeader signature. Thus, the round one RCert is a proof of the previous blocks hash. In any higher round of a given height the SigGroup must be of the RClaims object of the RCert.

The existence of a validly signed RCert implies that at least threshold validators have provided signatures. This either proves a new block height, in round one, or it proves a new round has begun in some height. This signature is a compact proof of consensus for the contents of the RClaims or the block.

7.11 The BClaims Object

BClaims is an abbreviation for Block Claims. The BClaims object encodes the proposed modifications to chain state in a compact format. The BClaims object is embedded, either directly or through composition, in the BlockHeader, Proposal, PreVote, PreCommit, and

NextHeight messages. The BClaims object does not itself carry the transactions that have been proposed for inclusion in the next block.

7.12 The PClaims Object

PClaims is an abbreviation for Proposal Claims. The PClaims object couples a Round Certificate to a set of Block Claims. The purpose of this coupling is to prove to any participants that observe such a message that the validators have progressed to the specified height and round with the specified previous Block Hash. In this way, a set of malicious validators may not vote in advance of a round that has yet not begun according to the rules of consensus. The PClaims object is never directly transmitted over the wire. The PClaims object is always an embedded subobject of an actual message type.

7.13 The Proposal Object

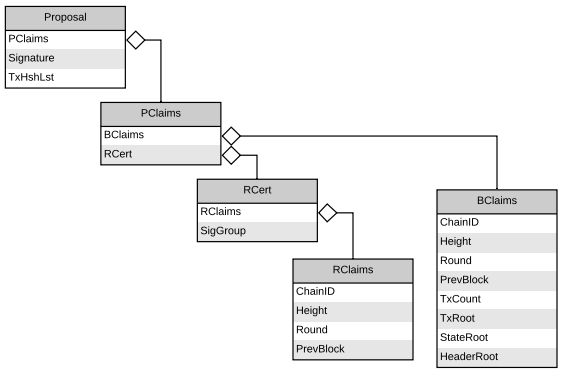

Figure 8: Proposal Object

The Proposal object allows a validator to propose a set of changes be applied to the blockchain as a transactional commitment. Only one proposal may be validly formed in a round. This fact is an implied constraint based on two requirements of the system. First, there may only be one validator who is the elected proposer for a given round. Second, a valid proposer will only propose one value in any round.

The proposal object contains a list of all transaction hashes that should be included in the calculation of the resulting block parameters as described by the embedded BClaims object of a given proposal. The acquisition of the actual transaction data is handled by an asynchronous background operation that queries the other validators for any unknown transaction hash. If a validator is unable to obtain the associated data for an unknown transaction, the validator will never vote for such a Proposal.

7.14 The PreVote Object

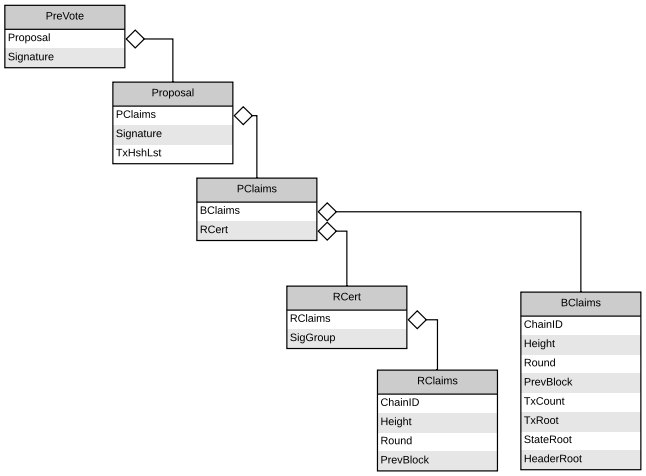

Figure 9: PreVote Object

The PreVote object is sent by a validator to indicate an agreement with a proposal as the next valid state transition. Upon receipt of a PreVote, the nested Proposal may be extracted and applied to the state of the appropriate validator if the Proposal has not otherwise been observed already. A valid validator will not PreVote more than one Proposal in any round. A valid validator will not PreVote and PreVoteNil in the same round.

7.15 The PreVoteNil Object

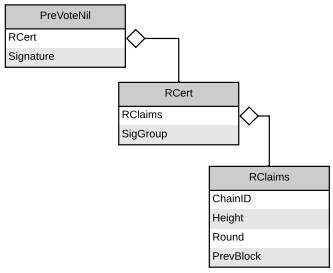

Figure 10: PreVoteNil Object

A PreVoteNil object is sent by a validator to indicate that no valid proposal has been observed before the ProposalTimeout or that the validator is otherwise locked on a competing value. A valid validator will not PreVote and PreVoteNil in the same round. The validator does not specify a proposal in a PreVoteNil; rather, the embedded Round Certificate acts as an identifier for a particular round and height. This indicator may be treated as a blanket statement for all Proposals seen in a given round.

7.16 The PreCommit Object

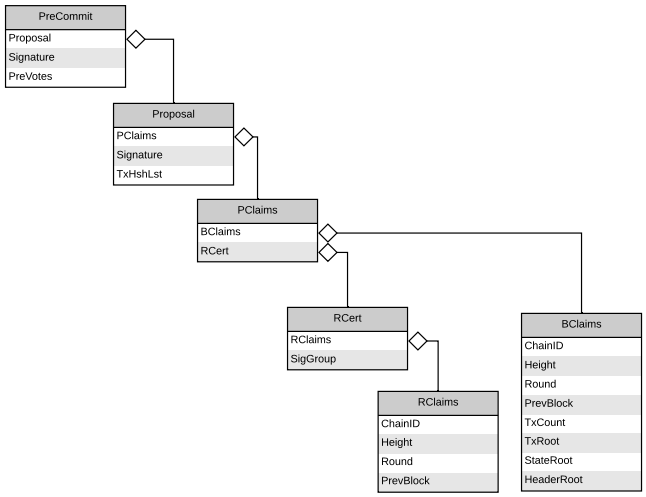

Figure 11: PreCommit Object

A PreCommit object is sent by a validator to indicate that the validator believes there is sufficient evidence to indicate that the associated proposal will be accepted as the next state transition. A validator may not form a valid PreCommit without having first seen at least threshold total validators have PreVoted for the specified Proposal in the same round. This requirement is enforced by appending the associated signatures from the required number of PreVotes to the PreCommit. A valid validator may not PreCommit more than one Proposal in a round. A valid validator will not PreCommit for a Proposal that it has also cast a PreVoteNil for in the same round. A validator will not PreCommit for a value that it has not cast a PreVote for in the same round.

7.17 The PreCommitNil Object

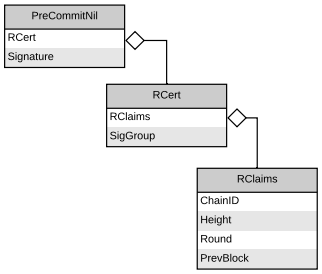

Figure 12: PreCommitNil Object

The PreCommitNil object is sent by a validator to indicate that it has not observed enough PreVotes to ensure a valid state transition may be created in the current round. A valid validator will not PreCommit and PreCommitNil in the same round.

7.18 The NextRound Object

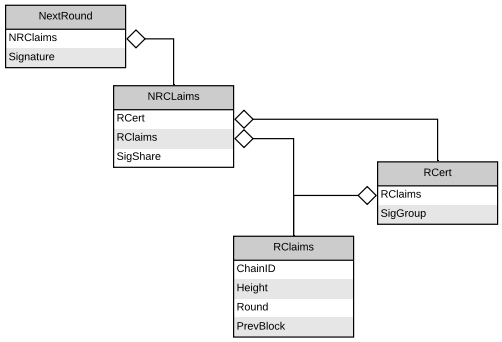

Figure 13: NextRound Object

The NextRound object is issued at the last phase of a round by any validator that does not have knowledge of consensus for the next block. This object contains two signatures. The first signature is a secp256k1 signature of the NRClaims object. The second signature is a BLS signature under the group share of the validator threshold cryptosystem. This second signature signs the RClaims object of the NRClaims object. This RClaims object contains the same parameters as the RCert for the current round, except the Round field has been incremented. After observing a threshold number of NextRound messages, a new RCert may be formed by aggregating the BLS signatures. This provides evidence of a consensus to enter a new round.

7.19 The NextHeight Object

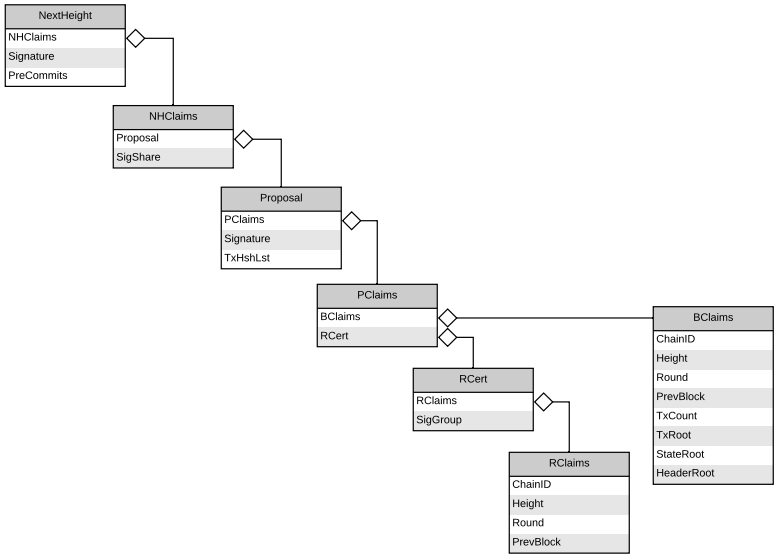

Figure 14: NextHeight Object

The NextHeight object is sent by a validator that has proof of threshold consensus in both the PreVote and PreCommit phases of operation. The object carries proof of this knowledge by appending the PreCommit signatures into the message. The NextHeight message carries two signatures. The first signature is a signature under secp256k1 ECDSA. The second signature is a signature under the threshold BLS key of the validator group. The secp256k1 key signs the canonical hash of the NHClaims objects. The BLS signature is a signature of the BClaims object. In this way, once a threshold number of NextHeight messages are formed, two objects may be created. The first object is the BlockHeader. The second object is the round one RCert for the first round of the next block height. These objects are formed by creating the SigGroup object through aggregation of the SigShare objects from threshold NextHeight objects.

7.20 The BlockHeader Object

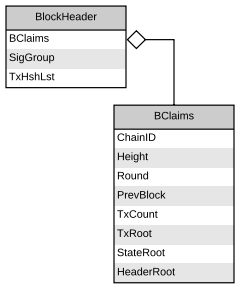

Figure 15: BlockHeader Object

A BlockHeader object may be formed following the successful completion of a round. This object may be formed by aggregating the BLS signatures of threshold valid NextHeight messages. The creation of a BlockHeader is an indication of consensus among the validators and the BClaims described state transition.

7.21 Validators

A validator is an entity that performs two actions. First, a validator must register as an entity that wishes to become a validator using a smart contract in the Ethereum Blockchain. This contract requires a deposit of stake at the time of this registration. The act of registering queues the validator up to become a new validator in the next epoch transition if a slot is available. The maximum number of slots the system may accommodate is 256, but the actual system will likely be constrained to far less than this limit in order to minimize key negotiation costs associated with the formation of the threshold BLS signatures needed for signing BlockHeaders and RCert objects.

Once a slot is available for the newly registered entity the second requirement comes into force. This requirement is that the entity is running a validating node in the network. This requirement is observed during the negotiation of the group key for the validators.

Full details may be seen in the technical addendum for how this key is negotiated and how unresponsive validators are removed.

The job of the validators is to mine new blocks into the chain and enforce the rules of consensus.

7.22 SnapShots

A snapshot is performed by writing a block header into the Ethereum blockchain. The block header is verified as validly signed under the group key of the validators and all fields are stored. This value may be changed within one epoch if an invalid state transition may be proven. This is not possible unless greater than two thirds of the validators collude to violate the rules of consensus. In the event that this does occur, the guilty validators may be identified through the Ethereum compliant secp256k1 ECDSA signatures used to sign every consensus message type. If no accusations are formed and proven within the epoch boundary, the snapshot becomes canonical.

7.23 Withdrawing From Validation

A validator may request to be withdrawn from a role as a validator at any time. The validator will not be available for actual withdrawal of funds until three epochs in the future, but the validator may not sign any additional consensus messages at the termination of the epoch in which the request was made. This down time guards against malicious validators performing an attack and then withdrawing before an accusation may be formed.

7.24 Leader Election

At this time the leader election algorithm is based on a simple round robin algorithm. This round robin algorithm does not track state across block heights. Specifically the algorithm operates on the sum of the blockheight and round modulo the number of validators. A single validator is selected from a common array of validators that every validator has available to them through the coordination of state provided by the Ethereum blockchain.

7.25 Virtual Votes

The concept of a virtual vote is integral to the security of the MadNetwork consensus algorithm. We define a virtual vote as a vote that has been cast based on the indirect observation of cryptographically-secure evidence. Although the message passing protocol of the consensus algorithm may seem overly verbose on first inspection, there is good reason.

Due to the fact that all message objects that allow the protocol to advance in height are objects that directly embed the Proposal they reference, and this Proposal must be cryptographically-signed, any party who observes a PreVote, PreCommit, or NextHeight message may extract the underlying Proposal and apply this Proposal against the state of the validator that formed the Proposal. The result of this operation may be one of three possible outcomes.

In the first case, the Proposal may be otherwise already known, and no action outside the normal tracking of messages is necessary.

In the second case, the local node may not have otherwise observed any valid Proposal from the validator that cast the Proposal. If this is the case, the message may be applied to the validator that formed the Proposal and the PreVote, PreCommit, or NextHeight message may be attributed to the validator that formed the actual message.

In the third case, the Proposal may be in conflict with an already known proposal from the associated validator for the same height and round. The handling of this case forms the basis for one of the most important security assumptions of the system: no conflicting PreVotes, PreCommits, or Proposals may be stored in the state space of a single validator in any round such that these conflicting votes may be counted toward consensus.

If the Proposal is in conflict, the Proposal will be discarded. In addition to discarding the Proposal, for the message types of PreVote and PreCommit, the validator that signed the PreVote or PreCommit will have a virtual PreVoteNil or PreCommitNil tracked for the purposes of consensus. This virtual Nil vote is a placeholder to indicate that the validator that signed the PreVote or PreCommit is not capable of voting in agreement with the local node. If the message that contained the Proposal is a NextHeight message, the NextHeight is followed if the local node may validate the Proposal and has not otherwise already performed a NextHeight operation for a conflicting value in the same height. If the local node has previously performed a NextHeight operation that is in conflict, this node must follow the NextHeight it has already locked. This should never be possible with less than two thirds malicious validators, but this situation may be mitigated by performing accusations against the malicious parties in the Ethereum smart contract system. Since all consensus messages are signed in Ethereum compliant ECDSA, the proof of a double proposal or a proof of signature for an out-of-turn/invalid proposal is guaranteed to be possible. Thus, the system will halt for a time but may re-enter consensus in the next block by forcing the assumed value of an empty block for the height at which the accusation was formed.

These operations are necessary because the consensus protocol requires a threshold number of votes before progress may be made through the steps of the protocol. Simply discarding these messages with no additional action would cause the protocol to halt. Tracking these messages would add many additional layers of complexity to the protocol. Thus we have selected to take the course of simplification.

In summary, the tracking of virtual votes performs three functions. First, this logic prevents the system from ever allowing two conflicting Proposal messages to ever be stored in the database of a single node. Second, because this operation is performed for all PreVote and PreCommit message types, this prevents any conflicting votes from ever influencing the counting of local votes that contribute to the addition of blocks. Third, because a virtual Nil vote may be assigned in the presence of a conflicting vote, the algorithm may be prevented from halting due to missing votes.

7.26 Protocol States

The consensus algorithm has been divided into discrete states. The current state of the system is determined by loading all known messages sent by the local node as well as secondary information about the block height, round number, and authorized validators. This

operation is performed once for each iteration of the consensus algorithm and this data is loaded directly from the database under a protected view. An explanation of the states and the conditions that determine a local nodes current state will be covered shortly.

The general flow of the algorithm is to attempt consensus for a fixed number of times at each block height. In each block height there may be many attempts and these attempts are named rounds. If the maximum number of rounds is reached without consensus, the system defaults to mining an empty block to ensure forward progress. The flow of each round is as follows.

Each round starts with a Proposal phase where one validator forms a proposal for the next block. This message is transmitted to all peers and each peer gossips the message to all other peers. This is a flood event that is designed to prevent split view attacks. This gossip halts at any node that has already gossiped the same message. This phase ends at the termination of a timeout. At the termination of the Proposal phase each validator will PreVote or PreVoteNil. This is the PreVote phase that may optimistically end before a timeout only if a threshold consensus is reached. Otherwise, the system will force a timeout wait before entering the next phase. In all cases no forward progress can be made without at least threshold votes having been observed. The next phase is the PreCommit phase in which a validator may either PreCommit or PreCommitNil. This phase is also constrained by optimistic threshold progress the same as the PreVote phase. Lastly is the commitment phase in which round consensus is determined. In this phase a validator may vote for a new round through a NextRound message or a new block through a NextHeight message. The full details are covered shortly.

The sections below will not explicitly discuss the process of gossip since it may be treated as independent of the core operation of the consensus algorithm for this portion of the analysis.

The approach taken to defining the consensus algorithm is far more verbose than many approaches normally taken. The justification for this verbosity is that although many papers describe what is claimed to be a convergent system, this is often not the case. Rather than leave a possible transition as not fully defined, we have chosen to fully expand the branch logic in the following discussion. We were able to fully cover the truth table of possible optionality by limiting the number of branch points. The basis for the proof of validity of this algorithm follows this section.

The main loop of the consensus algorithm is represented in pseudocode in Alg. 1. For simplicity, the algorithm waits for the timeouts to occur. In practice, if we reach consensus before the timeout, the algorithm proceeds to the next stage.

7.26.1 PendingProposal

Possible Next States:

- RoundJump
- HeightJump
- FormNextHeight
- ProposalStep
- PendingProposal
- PendingPreVote

Algorithm 1 Main loop of MadNet Consensus algorithm.

```
1: function MainBlockLoop()
2:     for r = 1; r ≤ DeadBlockRound; r++ do
3:         if r = DeadBlockRound then
4:             DeadBlockRoundProcedure()
5:             break
6:         end if
7:         nhBool = RegularRoundProcedure()
8:         if nhBool then
9:             break                          ▷ Proceed to the next block height
10:        end if
11:    end for
12:    return
13: end function
```

Algorithm 2 DeadBlockRound procedure

```
1: function DeadBlockRoundProcedure()
2:     PreVote(EmptyBlock)
3:     Wait until EmptyBlockPreVotes ≥ Threshold
4:     PreCommit(EmptyBlock)
5:     Wait until EmptyBlockPreCommits ≥ Threshold
6:     NextHeight(EmptyBlock)
7:     return
8: end function
```

Algorithm 3 Regular Round procedure

```
1: function RegularRoundProcedure()
2:     doPendingProposalStep()
3:     Wait for ProposalTimeout
4:     CurProp, LocalPreVote = doPendingPreVoteStep()
5:     Wait for PreVoteTimeout
6:     doPendingPreCommitStep(CurProp, LocalPreVote)
7:     Wait for PreCommitTimeout
8:     nhBool = doPendingNextStep(CurProp)
9:     return nhBool
10: end function
```

Algorithm 4 Proposal procedure

```
1: function doPendingProposalStep()
2:     if IsProposer() then
3:         if !LockedValueCurrent() ∧ !ValidValueCurrent() then
4:             NewProposal = MakeNewProposal()
5:             Propose(NewProposal)
6:             ValidValue = NewProposal
7:         else if !LockedValueCurrent() ∧ ValidValueCurrent() then
8:             Propose(ValidValue)
9:         else
10:            Propose(LockedValue)
11:        end if
12:    end if
13:    return
14: end function
```

Algorithm 5 PreVote procedure

```
1: function doPendingPreVoteStep()
2:     CurProp = GetCurrentProposal()
3:     LocalPreVote = Nil
4:     if CurProp ≠ Nil then
5:         if !LockedValueCurrent() ∧ !ValidValueCurrent() then
6:             if CurProp.IsValid() then
7:                 PreVote(CurProp)
8:                 ValidValue = CurProp
9:                 LocalPreVote = CurProp
10:            else
11:                PreVote(Nil)
12:            end if
13:        else if !LockedValueCurrent() ∧ ValidValueCurrent() then
14:            if CurProp = ValidValue then
15:                PreVote(ValidValue)
16:                LocalPreVote = CurProp
17:            else
18:                PreVote(Nil)
19:            end if
20:        else
21:            if CurProp = LockedValue then
22:                PreVote(LockedValue)
23:                LocalPreVote = CurProp
24:            else
25:                PreVote(Nil)
26:            end if
27:        end if
28:    else
29:        PreVote(Nil)                                  ▷ No current proposal
30:    end if
31:    return CurProp, LocalPreVote
32: end function
```

Algorithm 6 PreCommit procedure

```
1: function doPendingPreCommitStep(CurProp, LocalPreVote)
2:     NumPreVotes = GetCurrentPreVotes(CurProp)
3:     if NumPreVotes ≥ Threshold then
4:         ValidValue = CurProp
5:         if CurProp = LocalPreVote then
6:             PreCommit(CurProp)
7:             LockedValue = CurProp
8:             return
9:         else
10:            ValidValue = CurProp
11:        end if
12:    end if
13:    PreCommit(Nil)
14:    return
15: end function
```

Algorithm 7 NextStep procedure

```
1: function doPendingNextStep(CurProp)
2:     nhBool = False
3:     NumPreCommits = GetCurrentPreCommits(CurProp)
4:     if NumPreCommits ≥ Threshold then
5:         NextHeight(CurProp)
6:         nhBool = True
7:     else
8:         NextRound()
9:     end if
10:    return nhBool
11: end function
```

A valid node that has entered this state has not cast a Proposal, PreVote, PreVote-Nil, PreCommit, PreCommitNil, or NextRound vote for the current round. The Proposal-Timeout, the PreVoteTimeout, and the PreCommitTimeout have not expired for this round. The node has not seen a valid NextHeight message for the current height. The node has not seen a message for a higher round in the same height. The node has not seen a valid message for a higher BlockHeight.

If the local node is the proposer for the current round and neither ValidValue or Locked-Value are from the same height as the current round, the validator will form a new proposal. The validator will write this Proposal to the database and store this Proposal as ValidValue in the database as well.

If the local node is the proposer for the current round and ValidValue is from the current height, but LockedValue is not from the current height, the validator will propose the value defined by ValidValue and will set ValidValue equal to the constructed Proposal. The validator may then return.

If the local node is the proposer for the current round and LockedValue is from the current height, but ValidValue is not from the current height, the validator will propose the value defined by LockedValue and will set ValidValue equal to the constructed Proposal. The validator may then return.

All logic below this point is guarded by the condition that ValidValue and LockedValue are of the same blockheight as the current round.

If the local node is the proposer for the current round and the round number of ValidValue is greater than the round number for LockedValue, then ValidValue will be proposed and the validator will set ValidValue equal to the constructed Proposal. The validator may then return.

If the local node is the proposer for the current round and the round number for Locked-Value is greater than the round number for ValidValue, then the value defined by Locked-Value will be proposed and the validator will set ValidValue equal to the constructed Proposal. The validator may then return.

If the local node is the proposer for the current round and both LockedValue and Valid-Value are from the same round number, then the value defined by LockedValue will be proposed and the validator will set ValidValue equal to the constructed Proposal. The validator may then return.

If the local node is not the proposer for the current round, the node may immediately return without performing any work.

In order to give the reader insight as to what the structure of the previous statements look like in the actual implementation, the following code is provided from the source code of the MadNetwork repository; see Listing 1. The intent of including this example code is to allow the reader to understand how the conditional structures have been implemented. We have gone to great lengths to structure the code of the consensus algorithm as fully covering truth tables where possible to increase readability.

7.26.2 ProposalStep

Possible Next States:

- RoundJump

Listing 1: Pending Proposal Step

```
func (ce *Engine) doPendingProposalStep(
        txn *badger.Txn, rs *roundStates) error {
    // if local node is the proposer for this round
    // make a proposal
    if rs.localIsProposer() {
        // if not locked or valid form new proposal
        if !rs.LockedValueCurrent() && !rs.ValidValueCurrent() {
            // 00 case
            return ce.castNewProposalValue(txn, rs)
        }
        // if not locked but valid known, propose valid value
        if !rs.LockedValueCurrent() && rs.ValidValueCurrent() {
            // 01 case
            return ce.castProposalFromValue(txn, rs, rs.ValidValue())
        }
        // if locked, propose locked
        // 10
        // 11 case
        return ce.castProposalFromValue(txn, rs, rs.LockedValue())
    }
    // local node is not proposer,
    // do nothing until proposal timeout
    return nil
}
```

- HeightJump
- FormNextHeight
- ProposalStep
- PendingPreVote

A valid node that has entered this state is a designated proposer for the current round who has cast a Proposal but has not cast a PreVote, PreVoteNil, PreCommit, PreCommitNil, NextRound, or NextHeight vote for the current round. The ProposalTimeout, the PreVote-Timeout, and the PreCommitTimeout have not expired for this round. Further, the node has not seen a valid NextHeight message for the current height.

The node may immediately return without performing any work.

This state is a logically empty state that is intended to guard against a double proposal.

7.26.3 PendingPreVote

Possible Next States:

- RoundJump
- HeightJump
- FormNextHeight
- PreVoteStep
- PreVoteNilStep

A valid node that has entered this state has not cast a PreVote, PreVoteNil, PreCommit, PreCommitNil, NextRound, or NextHeight vote for the current round, and the Proposal-Timeout has expired. The PreVoteTimeout and the PreCommitTimeout have not expired for this round. Further, the node has not seen a valid NextHeight message for the current height.

If the local node is not a validator for the current round, the node may return without performing any additional work. The following logic is guarded by this conditional.

If a validator has received a Proposal that it has been able to verify as valid, then that proposal shall be called the PendingProposal for the purpose of this section. If the validator has not received a Proposal that it was able to verify as valid, then the PendingProposal shall be considered equal to the value Nil for the purpose of this section.

If the PendingProsposal is Nil, the validator will store a PreVoteNil in the local database and return.

If the PendingProsposal is not equal to Nil, the PendingProposal must be checked for validity with respect to the state transition. All logic below this point is guarded by the condition that the PendingProposal is not Nil.

If the PendingProposal may be verified as invalid, the validator may write a PreVoteNil to the database and return.

If the PendingProposal may be verified as valid, then the LockedValue and ValidValue must be checked for equivalence and recency. All logic below this point is guarded by the conditions that the PendingProposal is not Nil and has been verified as valid with respect to the state transition of the system.

If neither the ValidValue or LockedValue is from the current height, the validator may write a PreVote for the PendingProposal to the database and return.

If ValidValue is from the current height, but LockedValue is not from the current height, and the PendingProposal and the ValidValue are for the same BClaims object, then the validator will write a PreVote for the PendingProposal to the database and return.

If ValidValue is from the current height, but LockedValue is not from the current height, and the PendingProposal and the ValidValue are not for the same BClaims object, then the validator will write a PreVoteNil to the database for the current round and return.

If LockedValue is from the current height, but ValidValue is not from the current height, and the PendingProposal and the LockedValue are for the same BClaims object, then the validator will write a PreVote for the PendingProposal to the database and return.

If LockedValue is from the current height, but ValidValue is not from the current height, and the Pending proposal and the LockedValue are not for the same BClaims object, then the validator will write a PreVoteNil to the database for the current round and return.

If both the ValidValue and the LockedValue are from the current height, then the round numbers will be compared. All logic below this point is guarded by the condition that the ValidValue and the LockedValue are both for the same height and that height is equal to the height of the current round.

If the round number of the ValidValue is greater than the round number of the LockedValue, and the PendingProposal and the ValidValue are for the same BClaims object, then the validator will write a PreVote for the PendingProposal to the database and return.

If the round number of the ValidValue is greater than the round number of the LockedValue, and the Pending proposal and the ValidValue are not for the same BClaims object, then the validator will write a PreVoteNil to the database for the current round and return.

If the round number of the LockedValue is greater than the round number of the ValidValue, and the PendingProposal and the LockedValue are for the same BClaims object, then the validator will write a PreVote for the PendingProposal to the database and return.

If the round number of the LockedValue is greater than the round number of the ValidValue, and the Pending proposal and the LockedValue are not for the same BClaims object, then the validator will write a PreVoteNil to the database for the current round and return.

If the round number of the LockedValue and the ValidValue are equal, and the PendingProposal and the LockedValue are for the same BClaims object, then the validator will write a PreVote for the PendingProposal to the database and return.

If the round number of the LockedValue and the ValidValue are equal, and the Pending proposal and the LockedValue are not for the same BClaims object, then the validator will write a PreVoteNil to the database for the current round and return.

7.26.4 PreVoteNilStep

- RoundJump
- HeightJump
- FormNextHeight
- PreVoteNilStep
- PendingPreCommit
- PreCommitNilStep

A valid node that has entered this state has cast a PreVoteNil for the current round. A node that has entered this state has not cast a PreVote, PreCommit, PreCommitNil,

NextRound, or NextHeight vote for the current round, and the ProposalTimeout has expired. The PreVoteTimeout and the PreCommitTimeout have not expired for this round. Further, the node has not seen a valid NextHeight message for the current height.

If the local node is not a validator for the current round, the node may return without performing any additional work. The following logic is guarded by this conditional.

Upon entering this state, the node will first count the number of PreVote and PreVoteNil messages observed for the current round.

If greater than threshold PreVotes has been seen for this round, the node will write a PreCommitNil for the current round to the database. The node will also write the Proposal that has greater than threshold PreVotes to the database as ValidValue. The node may then return.

If greater than threshold PreVoteNils has been seen for this round, the node will write a PreCommitNil for the current round to the database and return.

If neither the number of PreVotes or the number of PreVoteNil messages is greater than the threshold, the node may return without performing any additional work.

7.26.5 PreVoteStep

Possible Next States:

- RoundJump
- HeightJump
- FormNextHeight
- PreVoteStep
- PendingPreCommit
- PreCommitStep

A valid node that has entered this state has cast a PreVote for the current round. A node that has entered this state has not cast a PreVoteNil, PreCommit, PreCommitNil, NextRound, or NextHeight vote for the current round, and the ProposalTimeout has expired. The PreVoteTimeout and the PreCommitTimeout have not expired for this round. Further, the node has not seen a valid NextHeight message for the current height.

If the local node is not a validator for the current round, the node may return without performing any additional work. The following logic is guarded by this conditional.

Upon entering this state, the node will first count the number of PreVote and PreVoteNil messages observed for the current round.

If greater than threshold PreVotes has been seen for this round, the node will write a PreCommit for the current round to the database. The node will store the Proposal associated with the PreCommit to the database as the ValidValue. The node will store the Proposal associated with the PreCommit to the database as the LockedValue. The node may then return.

If greater than threshold PreVoteNils has been seen for this round, the node will write a PreCommitNil for the current round to the database and return.

If neither the number of PreVotes or the number of PreVoteNils is greater than the threshold, the node may return without performing any additional work.

7.26.6 PendingPreCommit

Possible Next States:

- RoundJump
- HeightJump
- FormNextHeight
- PreCommitNilStep
- PreCommitStep

A valid node that has entered this state has cast either a PreVote or a PreVoteNil for the current round. A node that has entered this state has not cast a PreCommit, PreCommitNil, NextRound, or NextHeight vote for the current round. Both the ProposalTimeout and the PreVoteTimeout have expired for this round. The PreCommitTimeout has not expired for this round. Further, the node has not seen a valid NextHeight message for the current height.

If the local node is not a validator for the current round, the node may return without performing any additional work. The following logic is guarded by this conditional.

Upon entering this state, the node will first count the number of PreVote and PreVoteNil messages observed for the current round.

If the local node has cast a PreVote in the current round and the number of PreVotes is greater than the threshold, the node will write a PreCommit for the current round to the database. The node will also store the Proposal associated with the PreCommit to the database as the ValidValue. The node will also store the Proposal associated with the PreCommit to the database as the LockedValue. The node may then return.

If the local node has cast a PreVoteNil in the current round and the number of PreVotes is greater than the threshold, the node will write a PreCommitNil for the current round to the database. The node will also store the Proposal associated with the PreVotes to the database as the ValidValue. The node may then return.

If the local node has cast a PreVote in the current round and the number of PreVotes observed in the current round is less than the threshold and the sum of the number of PreVotes and PreVoteNils in the current round is greater than the threshold, the node will write a PreCommitNil to the database. The node may then return.

If the local node has cast a PreVoteNil in the current round and the number of PreVotes observed in the current round is less than the threshold and the sum of the number of PreVotes and PreVoteNils in the current round is greater than the threshold, the node will write a PreCommitNil to the database and return.

If none of the above conditions hold, the node may return without performing any additional work.

7.26.7 PreCommitNilStep

Possible Next States:

- RoundJump
- HeightJump
- FormNextHeight
- PreCommitNilStep
- NextRound

- NextHeight

A valid node that has entered this state has cast a PreCommitNil for the current round. A node that has entered this state has not cast a PreCommit, NextRound, or NextHeight vote for the current round. The ProposalTimeout has expired for this round. The PreVoteTimeout may have expired for this round. The PreCommitTimeout has not expired for this round. Further, the node has not seen a valid NextHeight message for the current height.

If the local node is not a validator for the current round, the node may return without performing any additional work. The following logic is guarded by this conditional.

Upon entering this state, the node will first count the number of PreCommit and PreCommitNil messages observed for the current round.

If the number of PreCommits is greater than the threshold and the node is capable of validating the Proposal associated with the PreCommits as performing a valid state transition, then the local node may write a NextHeight message to the database for the current round and return.

If the number of PreCommitNils is greater than the threshold and the number of PreCommits is greater than the zero, and the node is capable of validating that the Proposal associated with the PreCommit results in a valid state transition, the node will store the Proposal associated with the PreCommit to the database as the ValidValue. The node will also write to the database a NextRound for the current round and return.

If the number of PreCommitNils is greater than the threshold and the number of PreCommits is equal to zero the node will write to the database a NextRound for the current round and return.

If none of the above conditions hold, the node may return without performing any additional work.

7.26.8 PreCommitStep

Possible Next States:

- RoundJump
- HeightJump
- FormNextHeight
- PreCommitStep
- NextRound
- NextHeight

A valid node that has entered this state has cast a PreVote and PreCommit for the current round. A node that has entered this state has not cast a PreCommitNil, NextRound, or NextHeight vote for the current round. The ProposalTimeout has expired for this round and the PreVoteTimeout may have expired for this round. The PreCommitTimeout has not expired for this round. Further, the node has not seen a valid NextHeight message for the current height.

If the local node is not a validator for the current round, the node may return without performing any additional work. The following logic is guarded by this conditional.

Upon entering this state, the node will first count the number of PreCommit and PreCommitNil messages observed for the current round.

If the number of PreCommits is greater than threshold, the node will write a NextHeight message to the database and return.

If the number of PreCommitNils is greater than threshold, the node will write a NextRound message to the database and return.

If neither the number of PreCommits or the number of PreCommitNils is greater than the threshold, the node will return without performing any additional work.

7.26.9 PendingNext

Possible Next States:

- RoundJump
- HeightJump
- FormNextHeight
- PendingNext
- NextRound
- NextHeight

A valid node that has entered this state has cast a PreCommit or a PreCommitNil for the current round. A node that has entered this state has not cast a NextRound or NextHeight vote for the current round. The ProposalTimeout, the PreVoteTimeout, and the PreCommitTimeout have expired for this round. Further, the node has not seen a valid NextHeight message for the current height.

If the local node is not a validator for the current round, the node may return without performing any additional work. The following logic is guarded by this conditional.

Upon entering this state, the node will first count the number of PreCommit and PreCommitNil messages observed for the current round.

If the local node has cast a PreCommit in this round, and the number of PreCommits is greater than threshold, the node will write a NextHeight message to the database and return.

If the local node has cast a PreCommitNil in this round, the number of PreCommits is greater than threshold, and the local node is capable of validating the Proposal associated with the PreCommits, the node will write a NextHeight message to the database and return.

If the local node has cast a PreCommitNil in this round, the number of PreCommits is greater than threshold, and the local node is not capable of validating the Proposal associated with the PreCommits, the node will return without doing any further work.

If the number of PreCommits is less than threshold, but the sum of the PreCommits and the PreCommitNils is greater than the threshold, the local node will write a NextRound message to the database and return.

7.26.10 NextRoundStep

Possible Next States:

- RoundJump
- HeightJump
- FormNextHeight
- PendingNext

- NextRound
- NextHeight
- PendingProposal

A valid node that has entered this state has cast a PreCommit or a PreCommitNil for the current round. A node that has entered this state has not cast a NextHeight vote for the current round. The ProposalTimeout has expired for this round. The PreVoteTimeout may have expired for this round. The PreCommitTimeout may have expired for this round. Further, the node has not seen a valid NextHeight message for the current height.

If the local node is not a validator for the current round, the node may return without performing any additional work. The following logic is guarded by this conditional.

Upon entering this state, the node will first count the number of PreCommit and PreCommitNil messages observed for the current round. Upon entering this state, the node will also count the number of NextRound messages observed for the current round.

If the local node has cast a PreCommit in this round, and the number of PreCommits is greater than threshold, the node will write a NextRound message to the database and return.

If the local node has cast a PreCommitNil in this round, the number of PreCommits is greater than threshold, and the local node is capable of validating the Proposal associated with the PreCommits the node will write a NextRound message to the database and return.

If the local node has cast a PreCommitNil in this round, the number of PreCommits is greater than threshold, and the local node is not capable of validating the Proposal associated with the PreCommits the node will return without doing any further work.

If the local node has not seen greater than threshold PreCommits, and has not seen greater than threshold NextRound messages, the node may return without doing any further work.

If the local node has not seen greater than threshold PreCommits, and has seen greater than threshold NextRound messages, the node will write to the database a new RoundCert object and return.

7.26.11 NextHeightStep

Possible Next States:

- HeightJump
- NextHeight
- PendingProposal

A valid node that has entered this state has cast a NextHeight message for the current height.

There are no other guarantees about the state of this node.

If the local node is not a validator for the current round, the node may return without performing any additional work. The following logic is guarded by this conditional.

Upon entering this state, the node will first count the number of NextHeight messages observed for the current height.

If the number of NextHeight messages is greater than threshold, the validator will form a new BlockHeader and write it to the database.

If the number of NextHeight messages is not greater than threshold, the validator may return without performing any additional work.

7.26.12 RoundJump

Possible Next States:

- HeightJump
- FormNextHeight
- PendingProposal

A valid node that has entered this state has observed a valid Round Certificate with a higher round number in the same height as the current round. Further, the node has not seen a valid NextHeight message for the current height.

The node will write the new RCert to the database in the location of the local nodes MostRecentRoundCert and return.

7.26.13 HeightJump

Possible Next States:

- RoundJump
- HeightJump
- FormNextHeight
- PendingProposal

A valid node that has entered this state has seen a validly signed BlockHeader or Round Certificate for a height that is greater than the height of the current round.

If the height of the BlockHeader or Round Certificate is equal to one greater than the current round of the validator, and the PrevBlock value of the Round Certificate or BlockHeader is equal to the the expected blockhash from the current ValidValue, LockedValue, or the most recent PreVote that the local node cast, store the BlockHeader and return.

If the height of the BlockHeader or Round Certificate is equal to one greater than the current round of the validator, and the PrevBlock value of the Round Certificate or BlockHeader is NOT equal to the the expected blockhash from the current ValidValue, LockedValue, or the most recent PreVote that the local node cast, perform no additional work and return.

If the height is greater than one more than the height of the current round, return without performing any additional work.

Note: This logic was chosen to allow the synchronization protocol to perform the work necessary to bring the node up to the correct height.

7.26.14 FormNextHeight

Possible Next States:

- RoundJump
- HeightJump
- FormNextHeight
- NextHeight

A valid node that has entered this state has seen a valid NextHeight message for the current height.

There are no other guarantees about the state of this node.
Form a NextHeight message. Write the message to the database and return.

7.27 Mining Rewards

Mining rewards take two forms at this time. First, a validator is rewarded for cleaning up stale DataStore objects from the chain. Second, a validator is paid a base reward in tokens for performing the function of being a validator. These rewards are distributed in the block snapshotting process and are minted into the Ethereum blockchain. These rewards are not available for transfer or use until after at least one additional snapshot has been written to the Ethereum blockchain.

7.28 Slashing

In order to ensure the miners are honest, there is not only the possibility of mining rewards for following the protocol but also the threat of punishment for misbehavior. There will be two types of fines: major fine and minor fine. A major fine is any fine that can be cryptographically-verifiable malicious action. These include submitting incorrect shares or group public keys during the DKG process. While mining blocks, double-signing at a given block height is also malicious. A major fine should reduce a validator's stake to the point where he is unable to proceed with the DKG process. A minor fine may occur during the DKG process when a validator fails to submit the appropriate information. Because it is possible this was the result of technical failure and not malicious intent, we believe this should not be as large of a fine. Even so, validators are a critical part of the process and are expected to be resilient against technical failures.

7.29 Consensus Proofs

We now turn our attention to proving the safety of our consensus algorithm. This will follow from a number of lemmas.

7.29.1 Safety

The arguments of safety and fault tolerance for the system are natural extensions of the proofs for the ancestral origin of both the consensus mechanism and its parent. These algorithms are, respectively, DLS and Tendermint.

We build our model in the case of Partial Synchrony. Specifically, we assume that all messages that are sent must be eventually received. The sending of this message may be accomplished in a single attempt, or it may be accomplished through multiple retransmissions. In either case the message will be received and handled. Messages may arrive in any order and may arrive more than once. We aim for the following properties:

- All valid nodes decide on the same valid output value.
- All valid nodes eventually decide on some valid output value.
- A value is defined as valid if it satisfies a predefined predicate *isValid()*.

Our algorithm operates in the model of total group order equal to $3f+1$ with a threshold of $2f+1$ in order to ensure progress. Thus, we may tolerate up to f faults. Our system can tolerate strictly less than one-third Byzantine failures. We built our system to ensure safety even in the presence of greater than one-third total failures as long as we limit Byzantine failures to strictly less than one-third. This design principle does come with the necessary sacrifice of liveliness under the conditions that failures exceed the allowable threshold.

This section will be written entirely in the context of consensus and will not speak about this information in terms of transactions. We will speak about the system in terms of blocks and values, where a block represents an append-only operation onto a shared distributed log and a value will be a portion of the contents of the object appended to this log. For the purpose of this discussion, we may assume all other fields within this object are deterministically created based on application state and the proposed value itself.

7.29.2 Rules

If you PreCommit a Proposal in any round at any height, you will set LockedValue equal to the Proposal that you PreCommitVoted and you will not unset this value at the same block height in any round before signing the NextRound message that would lead to the DeadBlockRound.

After signing the NextRound message that would lead to the DeadBlockRound for a given block height, you must unset LockedValue and ValidValue.

After signing the NextRound message that would lead to the DeadBlockRound for a given block height, you must ignore any NextHeight message from any previous round.

7.29.3 Proof of Safety

The proof of safety relies on this observation from Lemma 1: two subsets with at least a threshold number of validators in each subset will share at least one honest validator.

Lemma 1
Let f be the number of malicious validators. Given any two subsets with at least $2f+1$ validators in a system containing $3f+1$ validators, those two subsets share at least $f+1$ validators. Thus, these two subsets share at least one honest validator.

Proof. Let A and B be two subsets with at least $2f+1$ validators chosen from $N = 3f+1$ validators. From De Morgans laws, we know

$$A \cap B = (A^c \cup B^c)^c.$$

Here, A^c denotes the complement of A (that is, the elements of the system not in A). In what follows, $|A|$ denotes the number of elements in A. Thus, we have

$$
\begin{aligned}
|A \cap B| &= |(A^c \cup B^c)^c| \\
&= N - |A^c \cup B^c| \\
&\geq N - (|A^c| + |B^c|) \\
&\geq N - 2f \\
&= f + 1.
\end{aligned}
$$

Their intersection shares at least $f+1$ validators and there are f malicious validators, so we see that A and B share at least one honest validator. □

This lemma ensures that honest participants will always agree when PreCommitting a proposal or submitting a NextHeight message in a given round, as the next two lemmas show.

Lemma 2
In any round, any 2 honest participants who PreCommit a proposal will PreCommit the same proposal.

Proof. Suppose two honest participants PreCommit proposals P_1 and P_2. To PreCommit a proposal, they both must have knowledge of at least $2f + 1$ PreVote messages from participants. There corresponds subsets S_1 and S_2, where S_1 contains the participants who submitted prevotes for P_1 and S_2 contains the participants who submitted PreVotes for P_2. By Lemma 1, S_1 and S_2 share at least one honest participant. An honest participant will only PreVote for one value in a round, so the proposals P_1 and P_2 must agree. □

Lemma 3
In any round, any 2 honest participants who submit a NextHeight message will submit a NextHeight message for the same proposal.

Proof. Mutatis mutandis, the proof is the same as that of Lemma 2. □

We are now able to show the multiple distinct NextHeight messages from honest participants are not able to occur before the DeadBlockRound. This is a stronger result than Lemma 3 and follows from the fact that a threshold number of LockedValues must occur before submitting a NextHeight message.

Lemma 4
In any round before the DeadBlockRound, any 2 honest participants who submit a NextHeight message will submit a NextHeight message for the same proposal.

Proof. Suppose we have not yet reached the DeadBlockRound. Let P_1 and P_2 be NextHeight messages from two honest participants with corresponding signing subsets S_1 and S_2. When P_1 is signed, all members of S_1 are supposed to set LockedValue to the proposal corresponding to P_1 because they PreCommitted P_1. Similarly, all members of S_2 are supposed to set LockedValue to the proposal corresponding to P_2 because they PreCommitted P_2. This

implies S_1 and S_2 are two subsets of participants of size at least $2f + 1$ with LockedValue set. Once an honest participant sets his LockedValue, it is never unset before the DeadBlockRound. By Lemma 1, S_1 and S_2 have at least one honest participant in common with his LockedValue set to one proposal. Thus, we see that the P_1 and P_2 NextHeight messages must be for the same proposal when they are submitted by honest participants. □

Lemma 5 is useful in bounding the number of honest validators who may not proceed to the next round.

Lemma 5
In order for $2f + 1$ validators to enter a round, at least $f + 1$ honest validators must have signed a NextRound message and at most f honest validators failed to sign a NextRound message.

Proof. We recall an honest validator may not enter a round without a valid RoundCertificate. A NextRound message contains two objects. The first object is a RoundCertificate for the current round. The second is a RoundShare object for the next round. A RoundCertificate contains both round number and block height as internal fields, and in order to form a valid Round Certificate, at least $2f + 1$ validators must have signed a RoundShare. Because there are at most f dishonest validators who signed for the RoundCertificate, at least $f + 1$ honest validators must have also signed the RoundCertificate. Thus, at most f honest validators did not sign the RoundCertificate. □

Once validators reach the DeadBlockRound, they will not acknowledge any NextHeight messages for any preceding round. This ensures that dishonest validators are not able to fork the chain by producing a block based on those NextHeight messages.

Lemma 6
If at least $2f + 1$ participants sign the RoundCertificate to the DeadBlockRound, then it is not possible for $2f + 1$ participants to form a valid NextHeight message for any round preceding the DeadBlockRound.

Proof. Given Lemma 5 and the rule that any honest validator who has signed a NextRound message for the DeadBlockRound will never sign or acknowledge any NextHeight message from a previous round, at least $f+1$ honest validators must have signed a NextRound message for the DeadBlockRound if there exists a RoundCertificate for the DeadBlockRound. If at least $f + 1$ honest validators are in the DeadBlockRound, at most f honest validators may remain in a round preceding the DeadBlockRound. If at most f honest validators remain in a preceding round, then the malicious validators are unable to use the signatures of f honest validators to form a set of $2f + 1$ valid NextHeight messages. Therefore, the malicious validators may not form a block from those NextHeight messages. □

With this assurance, we allow for participants to safely proceed to the DeadBlockRound even if they previously were locked onto a NextHeight message.

Lemma 7
It is safe for any participant who is locked on a NextHeight message to unlock and proceed to the DeadBlockRound upon receiving a RoundCertificate for the DeadBlockRound.

Proof. This follows from Lemma 6. □

The previous work allows us to show that we will converge to the EmptyBlock when a RoundCertificate for the DeadBlockRound exists. This ensures new blocks will be created even if no transactions are performed.

Lemma 8
The DeadBlockRound must converge to the EmptyBlock if a RoundCertificate for the DeadBlockRound exists.

Proof. Upon entering the DeadBlockRound, every valid process will immediately PreVote the EmptyBlock and ignore all other contradicting votes. These contradicting votes include any PreVote for a Proposal that is not the EmptyBlock, any NextRound message, any PreCommit that is not a PreCommit for the EmptyBlock, and any PreVoteNil or PreCommitNil message as well. As a result of Lemma 6, at least $f+1$ honest validators enter the DeadBlockRound. Therefore, the honest validators who enter the DeadBlockRound will only progress once at least f other validators also PreVote in the DeadBlockRound. Because we assume all messages are eventually received, the at most f honest validators who did not sign the NextRound Certificate for the DeadBlockRound will eventually receive the RoundCertificate for the DeadBlockRound and PreVote for the EmptyBlock in the DeadBlockRound. It follows that the round eventually converges to the EmptyBlock and no other possible block. □

The previous work also allows us to show we will not converge to more than one valid block at a given block height.

Lemma 9
It is not possible for our system to converge to more than one valid block for any given block height.

Proof. If the round enters the DeadBlockRound, then Lemma 8 shows that we will converge to the EmtpyBlock. Lemma 4 proves that it is not possible to have more than one proposal for which an associated NextHeight message has been validly formed before the DeadBlockRound. Lemma 7 allows for a safe transition into the DeadBlockRound. □

The next few lemmas assure the behavior of honest validators as it relates to voting.

Lemma 10
An honest validator who enters a round must eventually PreVote or PreVoteNil in that round.

Proof. All honest validators will either PreVote or PreVoteNil at the termination of the ProposalTimeout. Therefore, they must eventually prevote. □

Lemma 11
As long as $2f + 1$ honest validators enter a round, there will be at least $2f + 1$ PreCommits or PreCommitNils in that round.

Proof. By Lemma 10, all honest validators will eventually PreVote or PreVoteNil in a round. In the event that a PreVote is received for a competing Proposal from the perspective of a validator who has already PreVoted, that validator will count this PreVote as a PreVoteNil. From there, the honest validators will be able to either PreCommit or PreCommitNil, thus leading to $2f + 1$ PreCommits or PreCommitNils. □

Lemma 12
As long as $2f + 1$ honest validators enter a round, there will be at least $2f + 1$ NextRound or NextHeight messages in that round.

Proof. Mutatis mutandis, the proof is the same as that of Lemma 11. □

We now show that a round must terminate or a higher block is formed.

Lemma 13
If at any time a valid RoundCertificate exists for a round, that round must eventually terminate or a higher block must be formed.

Proof. We first focus on the case when no validator has signed a DeadBlockRound RoundCertificate. In this case, we may have that $f + 1$ honest validators have signed a NextHeight message and the round cannot terminate but a new block will be formed because the validators will eventually observe the previous NextHeight messages and will follow them. Otherwise, at least $f + 1$ honest validators will have entered the current round and it must eventually terminate; we will fall back to the previous case if any of these validators observe a NextHeight message.

In the case that a DeadBlockRound RoundCertificate exists, we have already proven termination by Lemma 8. □

Taken together, we are now able to show that our blockchain will make forward progress provided there are a limited number of faults.

Lemma 14
Our blockchain will always make forward progress so long as there are no more than f faults in the system.

Proof. If there are at most f faults, then all rounds must terminate or a new block will be formed by Lemma 13. □

8 Networking

8.1 Overview of Networking Stack

The networking stack consists of several layers that have been selected to allow for code generation and security. These layers consist of low level networking protocols upon which the Peer-to-Peer networking, Discovery, and BootNode protocol have been built.

8.2 Brontide

In order to provide authenticated encryption, we selected Brontide as the backbone of our networking stack. Brontide is the authenticated encryption protocol that is used by the Bitcoin Lightning Network and has been used securely in the wild for some time. The Brontide system was derived from the Noise Protocol Framework. Noise uses Diffie-Hellman key agreement to enable Authenticated Encryption with Associated Data (AEAD). The Noise Protocol Framework was created by Trevor Perrin, one of the original authors of the precursors to the Signal Protocol. The official protocol name Brontide uses is

`Noise_XK_secp256k1_ChaChaPoly_SHA256`.

Here, the `Noise_XK` handshake specifies the initiator knows the responders static public key, so it is never transmitted. The additional requirements include the elliptic curve secp256k1, the AEAD method based on the ChaCha20 symmetric cipher and Poly1305 authenticator, and the cryptographic hash function SHA256. We selected this type of protocol to ensure that a peer was protected from MitM attacks if the peer was connecting to a previously known host. Further, the protocol ensures data integrity without the need to depend on x.509 systems.

8.3 Yamux

Yamux is a robust multiplexing library built by HashiCorp. This lightweight TCP streaming multiplexor protocol was built to allow a TCP connection to be converted into an ordered full duplex communication channel that allows multiple logical connections to exist over a single TCP connection. This multiplexor allows gRPC to be used in our system seamlessly while also only requiring a single connection between peers.

8.4 gRPC

gRPC is a mature remote procedure call system originally developed at Google before being released to the public. It uses Protobuf (Protocol Buffers) for data serialization which allows for easy client-server communication necessary for the higher-order protocols. gRPC is used by many organizations including Netflix, Docker, and Spotify and has proven capable in demanding applications.

8.5 Connection Handshaking

In addition to the connection handshaking associated with Brontide, before any higher level protocol begins, two messages must be exchanged. The first message exchanges the Chain ID of each peer. If these values do not match, the connection is dropped. This has been built into this level of the protocol to prevent problems seen in other blockchain networks where peers attempt to sync the wrong chain. We chose instead to validate both peers are on the same network and terminate on failure as soon as possible. The second message communicates the port on which the remote node may be dialed back. The specific port that is passed in this location is the port on which the P2P protocol of a peer may be found. The reason this value is passed at this level is to facilitate the proper operation of discovery by forming a complete identity of a remote node. This identity includes the host, port, and public key of a remote node

8.6 Summary of Higher Order Protocols

At this time there are three higher order protocols in the communication stack. These are the BootNode protocol, the Discovery protocol, and the P2P protocol. These protocols have been divided into these classifications to allow differentiation of operations and security requirements. The BootNode protocol allows a peer who is first joining the network to discover peers in the overlay network using a persistent entry point. The Discovery protocol allows a peer to be connected to another peer for a single request that communicates other peers that a node may connect to. This protocol allows a node to build a peer list. Finally, the P2P protocol is a persistent full duplex channel between nodes in the P2P network. All protocols are based on gRPC after the initial handshake.

8.7 BootNode Protocol Summary

The BootNode protocol operates by allowing a remote host to connect to a persistent point of entry into the overlay P2P network. The bootnode servers are ignorant of the block chain and serve the singular purpose of allowing peering. These servers hold a cache of long term peers that have remained responsive to intermittent heartbeats over several days. These servers also hold an LRU cache that contains the most recently seen nodes up to cache eviction. Any node that connects to a bootnode server will make a single request that returns a list of peers that may be used for discovery and this list will contain items from both of the above mentioned caches. The bootnode server will terminate the connection after this single request is served.

8.8 Discovery Protocol Summary

The Discovery Protocol allows a node to find more peers. Peers are stored in one of sixteen buckets based on the first hex character of the public key of the peer. These buckets contain two sublists. One list stores active P2P connections while the other list stores known nodes from the discovery protocol requests. The algorithm attempts to fill the set of not connected buckets by dialing a random subset of peers from each bucket and requesting the closest

known peers to a random address. This is similar in nature to how Kademlia operates, but not identical. The returned peers are added to the list of possible peers to connect with. Once a node has enough peers that it may connect to, it begins dialing a random peer from each bucket until a user-defined number of peers have been connected to using the P2P protocol.

8.9 P2P Protocol Summary

The P2P protocol allows peers to gossip information about blockchain state, share consensus messages, and gossip transactions. This protocol is, like the other protocols, based on gRPC. The only difference is that all connections under this protocol are simulated connections that have been constructed from the Yamux multiplexing protocol. This allows the peers to act as both server and client in the P2P network while also leveraging the robust nature of gRPC in the process.

9 Data Structures

9.1 Compact Sparse Merkle Tries

In order to allow for the desired set of diverse verifiable data structures that are necessary in a blockchain system, we elected to select a general purpose tool that could fill most roles in our system without unacceptably sacrificing performance. Although more optimal structures may exist for some applications, forcing developers to build many complex forms of verifiable data structures complicates initial engineering and future ports to other languages. Thus, a single structure was selected that could be used in all current cases. This solution is based on a modified form of the open source Aergo State Trie. This is a Compact Sparse Merkle Trie that allows for compact proofs of inclusion/exclusion, stores a leaf at the optimal sub height, and operates as a base sixteen trie while actually being a true binary trie implementation. This polymorphic nature is derived from the manner in which the trie is built. The length of a generalized encoded proof of inclusion/exclusion requires $O(\log n)$ merkle proof keys, with an additional overhead of 64 bytes to communicate the key and value of the proof. The proof also requires $O(\log n)$ bits to communicate the positioning of the merkle proof keys in the trie. Thus, the total size is effectively $O(\log n)$.

The trie itself is a binary trie where all nodes are stored in height four sub tries. Each sub-trie may be loaded as a batched operation that accommodates a compact encoding through the use of bit fields that are similar to those used in the proofs. Thus, for a given height four sub trie, only nonempty nodes need be stored, and the bit field may be parsed to determine the positions these nodes should occupy in the actual trie. The process of updating the trie may be performed in a concurrent manner by applying all leaf transformations first and then concurrently calculating the hashes with a blocking operation on each height four sub-trie at the leaf levels. Thus, each sub-sub-trie may be calculated before the root of the sub-trie itself is calculated, and this concurrency may be arbitrarily extended across the trie.

9.2 State Trie

The State Trie is a CSMT that includes two types of objects and has the root value of the trie updated and stored in every block. The first object is the hash of a UTXO stored in the location of its UTXOID. The second type of object is the hash of a deposit nonce that is stored in the location that is equal to the deposit nonce. The first objects, the UTXO type objects, allow a compact proof of inclusion or exclusion to be formed for the UTXO at a given block height. This is useful for proofs of datastores, proofs to light clients, and allows fast syncing of the chain from a known good block, such as a snapshot. When a UTXO is created, it is added to the trie. When the UTXO is consumed, it is removed from the trie. Deposits work in an inverse manner. When a deposit is spent, it is added to the trie and shall remain there in perpetuity. This addition tracks the spending of deposits.

9.3 BlockHash Trie

The BlockHash Trie is a CSMT that stores the block hash of each block mined at a leaf value where the key is the zero padded block number. This value is included in every block, where the value in each block references the root of the trie after inclusion of the previous block. The structure is maintained in order to allow light clients to query blocks on a singular basis with proof of inclusion for each queried block. The proofs themselves are intended to start from a snapshot block and allow previous blocks from the snapshot to be proven as valid blocks from the entire chain. This system also has the benefit of allowing proofs of sidechain state to be formed much more readily in the Ethereum blockchain due to the ability to easily prove a block header as having a specific block hash as a fixed complexity procedure. Alternative systems must either track every block hash in the Ethereum blockchain, or large batches of contiguous blocks must be injected in order to prove state at an arbitrary block. We overcome these limitations through the proofs of inclusion and exclusion offered by the CSMT.

10 Economics

Miners are privileged actors who choose transactions to include in the sidechain blocks. In order to ensure honesty, there must be some method to punish nefarious validators when cryptographically-verifiable malicious behavior is observed. Therefore, all miners are required to stake a certain amount of Ether in a smart contract before being able to become a validator. This ensures the desired Nash equilibrium: it is in the miner's interest to correctly follow the procedure and receive compensation rather than to deviate from it and lose significant stake. This includes malicious behavior during the distributed key generation procedure (for example, failing to correctly share a secret) as well as when mining blocks (for example, signing two different proposals). All cryptographically-verifiable malicious behavior will be validated as such by an Ethereum smart contract. We note that submitting a false accusation is malicious behavior and will result in stake slashing.

In order to ensure security, we do not allow direct withdrawals at this time; this is due to security concerns and complexities of exit games. In particular, there is the possibility miners could create tokens out of thin air and then quickly move those tokens to the Ethereum

blockchain. By not allowing for direct withdrawals, those tokens created out of thin air cannot be removed. Creating such tokens would be malicious behavior by a miner who would then be punished accordingly. If necessary, the sidechain may be reorganized to negate such malicious behavior; however, it is not possible to invalidate blocks on Ethereum once they have been committed. Thus, atomic swaps are safer than direct withdrawals in the context of our security model and we take this approach.

We allow for deposits from Ethereum into the sidechain to occur through direct exchange. When a deposit occurs in the Ethereum MainNet, tokens are burned in the Ethereum Chain and an equal number of tokens become available after a minimum block wait time in the sidechain.

Miners are rewarded in two different ways. The first way miners are rewarded is upon the completion of an epoch when a snapshot is written into the Ethereum blockchain. This incentivizes miners to mine blocks and have the system progress, as mining more blocks will result in more rewards. Snapshots occur at a fixed interval of sidechain blocks as well as after a minimum required number of Ethereum blocks.

Validators are also rewarded for cleaning up stale state; that is, when a DataStore has expired, the miner can delete the DataStore and claim its deposit as a reward. Both of these rewards are in the form of MadNet tokens. This means that miners will be the source of token liquidity within the system, as they are required to have an Ethereum account as well as having a significant supply of tokens; furthermore, they determine which transactions are included in the blocks they propose and could include their atomic swaps. In order to store data, MadNet tokens are required to be burned. So long as the data exists, more tokens are burned to compensate the miners for being required to store the data.

In order to establish value stability during the initial period of low utilization, an incentivized mechanism of illiquidity has been established. This mechanism is designed to allow token holders to lock balances of tokens into a smart contract that will return yield to the owner that is expressed as a function of token utilization and the amount locked.

First, the amount of illiquid tokens is set. From there, we divide these tokens into different tranches; each tranche holds tokens for a predetermined time. The longer holding requirements will result in a greater rate of return. Should we determine that the system requires more tokens, tokens may be released, starting with the lowest yield tranche and working toward the highest yield. A bonus will be paid for early release as specified in the smart contract; the exact rate of return will be discussed below. We expect the number of tokens required to follow a normal distribution, so we use that to help distribute the tokens appropriately in the tranches. Below is a plot of the relative sizes of the tranches. The exact number of tranches is yet to be determined but there is no restriction on the allowable number. In the plot below, the tranches will be released from left to right, starting near 0 and ending near 1; see Fig. 16. The specified rate of return will increase from left to right.

A larger rate of return will occur should tokens need to be released into the market sooner than anticipated. The actual rate of return will be determined in relation to the expected holding and will vary smoothly with respect to time (the number of blocks); the maximum payout will be double the expected rate of return. All of this helps ensure a stable initial value for the token. A plot of the rate of return is shown below. Note the plot is the tokens paid out in addition to what was deposited. Here, if the tokens are released at one-half the holding time, the rate of return will be twice the agreed upon amount. Similarly, if the

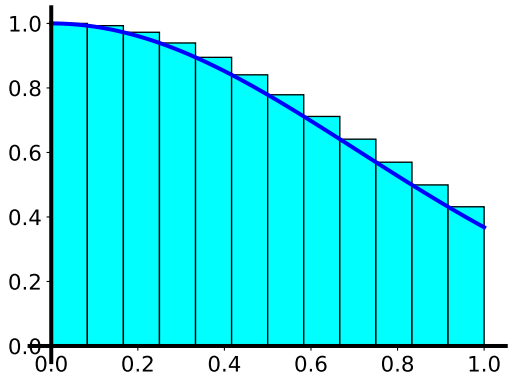

Figure 16: Relative size of an potential set of tranches to incentize illiquidity.

tokens are released at the originally agreed upon time, the rate of return will be the agreed upon amount (technically, a slightly greater amount); see Fig. 17. Thus, if tokens are quickly released, there will be only a small payout, but after an initial period of time, the rate of return will be at least the specified amount.

11 Layer Three

11.1 Scriptless Scripts and State Channels

An important implication of this work is one that was not fully considered at the time of conception. The realization came later when thinking about how the accusation and snapshot formation process operates. Namely, the inclusion of the abstraction of signature verification may be extended in further iterations to allow Schnorr signatures and other primitives. With this toolbox in place and the ability to extend the validation of a UTXO in an isolated manner, as is afforded by the design of the account abstraction system, more exotic primitives may be easily built. These include adaptor signatures, SE-snarks, and many other primitives that may be leveraged to accommodate the concept of scriptless scripts. Ultimately these constructs allow the system we have defined to be used as a restricted smart contract capable system without the need to verify the actual smart contract logic in the chain.

Further, other protocols may leverage these capabilities in concert with the existing ability to compactly prove state about the sidechain in the context of an Ethereum smart contract. The authors of this paper believe this ultimately may serve as an interesting primitive for constructing proofs of verifiable computation within an Ethereum smart contract. An example of this process that may be built based on the operation of the system without any modification follows.

State channels are a promising technology that suffer from a problem of requiring all participants to be online at all times. This problem may be addressed through a hybrid solution based on the construction of MadNetwork. Given the existence of MadNetwork, a

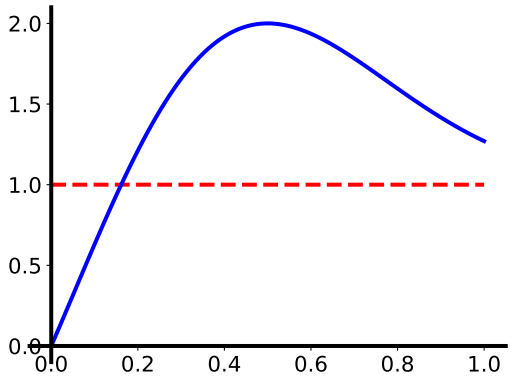

Figure 17: This plot shows how the specified rate of return will be multiplied by to show the actual rate of return. The maximum return occurs when the tokens in the tranche are released at one-half of the number of blocks they were originally held. This would occur because the system requires a large supply of tokens.

state channel may be constructed using a negotiated BLS multisignature account as described in the account abstraction section.

Let these two parties be named Alice and Bob. Let Alice and Bob have some motivation for continual exchange of value. Alice and Bob coordinated to negotiate a 2 out of 2 BLS multisignature.

Alice will be the first mover in this example, and so Alice will create a smart contract on Ethereum in which she writes the negotiated public key and deposits some value. Let this contract have knowledge of the snapshot contract for MadNetwork. Let this smart contract have three exit conditions. The first exit should be an exit condition that allows Alice to remove the deposit if Bob does not also place an equal deposit as Alice by some predetermined block number in the future. Let the other two exit conditions only be accessible after both parties have placed such a deposit. Let these other two exit conditions be an exit initiated by Alice or an exit initiated by Bob. Let an exit by Alice or Bob only occur within ten Ethereum blocks of a snapshot being recorded to the Ethereum blockchain by the validators of MadNetwork. Let this exit require a proof of state against the snapshot and only be valid for the most recent snapshot of MadNetwork.

Before Bob places a deposit, let Alice and Bob coordinate to create a DataStore in MadNetwork that holds a single 32 byte value. Let this value be the balance of Alices deposit in the Ethereum smart contract. Once this datastore has been constructed, let Bob place his deposit into the smart contract. Each time Alice and Bob wish to exchange value let Alice and Bob coordinate to sign a new transaction that overwrites the previous DataStore with a new value. This operation should not occur in any epoch until at least fifteen blocks have been recorded in the Ethereum blockchain since the last snapshot. This delay is to ensure the window of generating a valid exit proof for the smart contract is fully terminated before a new round of exchange may begin.

Let the value of the DataStore be the new balance of Alice after a mutually agreed upon

exchange of value. What governs such exchange is beyond the scope of this example. Let us simply say that they agree to exchange value based on information external to the system. For each state transition, both parties must sign a partial signature to a transaction that is written to the sidechain. Thus, the value is mutually agreed upon.

If at any point either party wishes to terminate the channel, the party may stop signing additional changes to the DataStore. At the termination of an epoch, the data of this DataStore may be proven in the Ethereum Blockchain in a compact manner using the parameters of the most recent snapshot. The proof of existence for the DataStore UTXO is just a Merkle Proof of Inclusion against the StateRoot of the snapshot block. The parsing logic of the DataStore is fairly trivial assuming the RawData field of the DataStore is a fixed size object. This assumption will allow fixed offset parsing of the canonical encoding of the DataStore object.

Once the existence of a UTXO is proven against a snapshot, the public key of the signer must be proven as equal to the 2 out of 2 BLS public key as stored in the smart contract. Assuming economic security of the sidechain is sufficient, the BLS signature of the DataStore need not be verified in the smart contract since this requirement is enforced in the sidechain validation logic. If this assumption is insufficient, our system is capable of validating a BLS signature in an Ethereum smart contract for less than 200K gas. This can likely be bounded more closely to 150K gas, but 200K is a safe upper bound for what we are describing.

Alice or Bob may initiate such a termination in any epoch in the designated window. Thus, the final state of the external system may be collapsed and proven in the context of the EVM. This state is the final state of the system which is, by definition, the last state that both parties have agreed to in advance. Given that this state is the balance of Alice, the smart contract may enforce that Alice may withdraw only this balance at the termination of the channel. Bob may only withdraw his own value minus the delta of Alices initial value.

The operation of parsing the object, validating the Merkle proof, and checking the data field as a simple uint256 value should cost around 700K gas for a Merkle Proof of 256 keys. This is the most complex Merkle Proof possible and actual costs are likely to be a fraction of this worst case value. Specifically, the cost should scale as $O(\log n)$ of the number of UTXOs that exist in the State Trie.

This construct is a toy example, but it does highlight a core benefit of the protocol. Namely, that the problems around state channel participant availability may be addressed by preventing an exit from a state both parties have not agreed to as the most recent state of the system. Other constructs may be devised for more complex cases. The fundamental idea is that MadNetwork may act as an out-of-band storage system that is cryptographically verifiable and allows proof of an item being the most recent object state.

11.2 AdLedger PKI

One application of MadNetwork is Transparent by Design Public Key Infrastructure (TPKI). This is discussed more fully in its own whitepaper, but we summarize the main ideas here.

The need for TPKI comes from the unfortunate but undeniable truth: the x.509 standard is broken beyond repair. This comes from ill-defined standards as well as haphazard attempts at patchwork extensions to salvage it. One main issue arises from certificate revocation, as it

is not possible to ensure that invalid certificates are rejected. A proposed solution is OCSP, but it is not sufficient due to its inability to handle the large number of requests.

To counteract this, we developed TPKI. Instead of issuing certificates with the lifetime of years that may become compromised, we issue short-lived certificates (validity around one day) that are regularly renewed. Having short-lived certificates ensures auto-revocation and requires proof of continued compliance for reissuance. Additionally, each certificate has well-defined properties and we do not allow arbitrarily chaining of certificates; this protects us against some known attacks. Furthermore, any certificate not present or ill-formed is considered invalid by default and should not be trusted. Root level certificates are not cross-validated (although they are self-signed) to ensure a well-defined notion of validity and revocation. Should a root certificate become compromised, a predetermined storage location in MadNetwork will hold the revocation key. This revocation public key, if present, will show that all users should reject subordinate certificates from this root certificate. We envision there being multiple Certificate Authorities with separate root certificates so that the revocation of one will not affect the others.

References

[1] A. Guttman. *Digital ad fraud losses worldwide 2018-2022*. Statista. Nov. 27, 2019. URL: https://www.statista.com/statistics/677466/digital-ad-fraud-cost/.

[2] Gora Adj and Francisco Rodríguez-Henríquez. *Square root computation over even extension fields*. Cryptology ePrint Archive, Report 2012/685. https://eprint.iacr.org/2012/685. 2012.

[3] Razvan Barbulescu and Sylvain Duquesne. *Updating key size estimations for pairings*. Cryptology ePrint Archive, Report 2017/334. https://eprint.iacr.org/2017/334. 2017.

[4] Paulo S. L. M. Barreto and Michael Naehrig. *Pairing-Friendly Elliptic Curves of Prime Order*. Cryptology ePrint Archive, Report 2005/133. https://eprint.iacr.org/2005/133. 2005.

[5] Connor Blenkinsop. *Blockchain's Scaling Problem, Explained*. Aug. 22, 2018. URL: https://cointelegraph.com/explained/blockchains-scaling-problem-explained.

[6] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short signatures from the Weil pairing". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2001, pp. 514–532.

[7] Jan Camenisch and Markus Stadler. "Proof systems for general statements about discrete logarithms". In: *Technical report/Dept. of Computer Science, ETH Zürich* 260 (1997).

[8] Sanjit Chatterjee, Darrel Hankerson, and Alfred Menezes. *On the Efficiency and Security of Pairing-Based Protocols in the Type 1 and Type 4 Settings*. Cryptology ePrint Archive, Report 2010/388. https://eprint.iacr.org/2010/388. 2010.

[9] Jennifer Derke. *Ads.cert v1.0: Signed Bid Requests*. Nov. 2018. URL: https://github.com/InteractiveAdvertisingBureau/openrtb/blob/master/ads.cert:%20Signed%20Bid%20Requests%201.0%20BETA.md.

[10] Pierre-Alain Fouque and Mehdi Tibouchi. "Indifferentiable hashing to Barreto–Naehrig curves". In: *International Conference on Cryptology and Information Security in Latin America*. Springer. 2012, pp. 1–17.

[11] Rosario Gennaro et al. "Revisiting the distributed key generation for discrete-log based Cryptosystems". In: *RSA Security'03* ().

[12] Rosario Gennaro et al. "Secure distributed key generation for discrete-log based cryptosystems". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1999, pp. 295–310.

[13] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. *An introduction to mathematical cryptography*. Vol. 1. Springer, 2008.

[14] Thomas Icart. "How to hash into elliptic curves". In: *Annual International Cryptology Conference*. Springer. 2009, pp. 303–316.

[15] Jonathan Katz et al. *Handbook of Applied Cryptography*. CRC press, 1996.

[16] Taechan Kim and Razvan Barbulescu. *Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case*. Cryptology ePrint Archive, Report 2015/1027. https://eprint.iacr.org/2015/1027. 2015.

[17] Alfred Menezes, Palash Sarkar, and Shashank Singh. *Challenges with Assessing the Impact of NFS Advances on the Security of Pairing-based Cryptography*. Cryptology ePrint Archive, Report 2016/1102. https://eprint.iacr.org/2016/1102. 2016.

[18] Peter L Montgomery. "Modular multiplication without trial division". In: *Mathematics of computation* 44.170 (1985), pp. 519–521.

[19] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. "New software speed records for cryptographic pairings". In: *International Conference on Cryptology and Information Security in Latin America*. Springer. 2010, pp. 109–123.

[20] Stephen O'Neal. *Who Scales It Best? Inside Blockchains' Ongoing Transactions-Per-Second Race*. Jan. 22, 2019. URL: https://cointelegraph.com/news/who-scales-it-best-inside-blockchains-ongoing-transactions-per-second-race.

[21] Stefan Rass and Daniel Slamanig. *Cryptography for security and privacy in cloud computing*. Artech House, 2013.

[22] Neal Richter. *Helping the industry prevent the sale of counterfeit inventory with ads.txt*. IAB. May 16, 2017. URL: https://iabtechlab.com/blog/helping-industry-prevent-sale-of-counterfeit-inventory-with-ads-txt/.

[23] Philipp Schindler et al. *ETHDKG: Distributed Key Generation with Ethereum Smart Contracts*. Cryptology ePrint Archive, Report 2019/985. https://eprint.iacr.org/2019/985. 2019.

[24] Mehdi Tibouchi. "Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2014, pp. 139–156.

[25] Kenton Varda. *Cap'n Proto*. 2020. URL: https://capnproto.org/.

[26] Kenton Varda. *Cap'n Proto*. 2020. URL: https://capnproto.org/news/.

[27] Riad S. Wahby and Dan Boneh. *Fast and simple constant-time hashing to the BLS12-381 elliptic curve*. Cryptology ePrint Archive, Report 2019/403. https://eprint.iacr.org/2019/403. 2019.

[28] Turner Wright. *Ethereum Now Rivals Bitcoin for Daily Value Transfers*. Apr. 16, 2020. URL: https://cointelegraph.com/news/ethereum-now-rivals-bitcoin-for-daily-value-transfers.

A Transaction Object Specification

Listing 2: Cap'n Proto Transaction and UTXO Definition

```
using Go = import "/go.capnp";
@0xb99093b7d2518300;
$Go.package("capn");
$Go.import("github.com/MadHive/MadNet/application/capn");

const defaultDSPreImage :DSPreImage = (chainID = 0, index = 0x"00",
    issuedAt = 0, deposit = 0, rawData = 0x"00", owner = 0x"00");
const defaultDSLinker :DSLinker = (txHash = 0x"00",
    dSPreImage = .defaultDSPreImage);
const defaultVSPreImage :VSPreImage = (chainID = 0, value = 0,
    owner = 0x"00");
const defaultASPreImage :ASPreImage = (chainID = 0, value = 0,
    owner = 0x"00", issuedAt = 0, exp = 0);
const defaultTXInPreImage :TXInPreImage = (chainID = 0,
    consumedTxIdx = 0, consumedTxHash = 0x"00");
const defaultTXInLinker :TXInLinker = (
    tXInPreImage = .defaultTXInPreImage, txHash = 0x"00");

struct DSPreImage {
    chainID @0 :UInt32 = 0;
    # The chainID of the chain this object was created on.

    index @1 :Data = 0x"00";
    # The index of this data reference.

    issuedAt @2 :UInt32 = 0;
    # The Epoch during which this object was created.

    deposit @3 :UInt32 = 0;
    # The deposit given to this datastore.

    rawData @4 :Data = 0x"00";
    # The raw data associated with this data store.

    tXOutIdx @5 :UInt32 = 0;
    # The index at which this element appears in the transaction
    # output list.

    owner @6 :Data = 0x"00";
    # The hash of the public key of the owner of this object.
}

struct DSLinker {
    dSPreImage @0 :DSPreImage = .defaultDSPreImage;
```

```
    # The structure containing particular information for this object.

    txHash @1 :Data = 0x"00";
    # The hash of the transaction that created this object.
}

struct DataStore {
    dSLinker @0 :DSLinker = .defaultDSLinker;
    # Linking from object to txHash.

    signature @1 :Data = 0x"00";
    # Signature of the DSLinker
}

################################################################################

struct VSPreImage {
    chainID @0 :UInt32 = 0;
    # The chainID of this object.

    value @1 :UInt32 = 0;
    # The value stored in this object.

    tXOutIdx @2 :UInt32 = 0;
    # The index at which this element appears in the transaction
    # output list.

    owner @3 :Data = 0x"00";
    # The hash of the public key of the owner of this object.
}

struct ValueStore {
    vSPreImage @0 :VSPreImage = .defaultVSPreImage;
    # The structure containing particular information for this object.

    txHash @1 :Data = 0x"00";
    # The hash of the transaction that created this object.
}

################################################################################

struct ASPreImage {
    chainID @0 :UInt32 = 0;
    # The chainID of this object.

    value @1 :UInt32 = 0;
    # The value stored in this object.
```

```
    tXOutIdx @2 :UInt32 = 0;
    # The index at which this element appears in the transaction
    # output list.

    owner @3 :Data = 0x"00";
    # <sva><curve><hashlock><initial owner pubk hash><partner pubk hash>
    # The hash of the public key of the original owner of this object.

    issuedAt @4 :UInt32 = 0;
    # The Epoch during which this object was created.

    exp @5 :UInt32 = 0;
    # The Epoch during which this object will fall back to the original
    # owner if it is not claimed by the partner before this point.
    # For safety this should be at least three epochs after issuedAt.
}

struct AtomicSwap {
    aSPreImage @0 :ASPreImage = .defaultASPreImage;
    # The structure containing particular information for this object.

    txHash @1 :Data = 0x"00";
    # The hash of the transaction that created this object.
}

################################################################################

struct TXInPreImage {
    chainID @0 :UInt32 = 0;
    # Chain id on which this object was created.

    consumedTxIdx @1 :UInt32 = 0;
    # Index at which the consumed object was created in the tx named
    # by consumedTxHash or the max value of uint32 to signal a deposit
    # from Ethereum.

    consumedTxHash @2 :Data = 0x"00";
    # The hash of the transaction that created the object to be
    # consumed or the nonce of the deposit if input is a deposit from
    # Ethereum bc.
}

struct TXInLinker {
    tXInPreImage @0 :TXInPreImage = .defaultTXInPreImage;
    # The structure containing particular information for this object.
```

```
    txHash @1 :Data = 0x"00";
    # The hash of the transaction that is consuming this object.
}

struct TXIn {
    tXInLinker @0 :TXInLinker = .defaultTXInLinker;
    # Linking from object to txHash.

    signature @1 :Data = 0x"00";
    # Signature of linker.
}

################################################################################

struct TXOut {
    union {
        dataStore  @0 :DataStore;
        # The output if it is a datastore

        valueStore  @1 :ValueStore;
        # The output if it is a valuestore

        atomicSwap @2 :AtomicSwap;
    }
}

################################################################################

struct Tx {
    vin @0 :List(TXIn) = [];
    # Transaction input vector.

    vout @1 :List(TXOut) = [];
    # Transaction output vector.
}

################################################################################
```

B Consensus Object Specification

Listing 3: Cap'n Proto Consensus Object Specification

```
using Go = import "/go.capnp";
@0x85d3acc39d94e0f8;
$Go.package("capn");
$Go.import("github.com/MadHive/MadNet/consenus/capn");

const defaultRound :UInt32 = 0;
const defaultHeight :UInt32 = 0;
const defaultChainID :UInt32 = 0;
const defaultNumberTransactions :UInt32 = 0;
const defaultRClaims :RClaims = (chainID = .defaultChainID,
    height = .defaultHeight, round = .defaultRound,
    prevBlock = 0x"00");
const defaultBClaims :BClaims = (chainID = .defaultChainID,
    height = .defaultHeight, prevBlock = 0x"00",
    txCount = .defaultNumberTransactions, txRoot = 0x"00",
    stateRoot = 0x"00", headerRoot = 0x"00");
const defaultRCert :RCert = (rClaims = .defaultRClaims,
    sigGroup = 0x"00");
const defaultPClaims :PClaims = (bClaims = .defaultBClaims,
    rCert = .defaultRCert);
const defaultNRClaims :NRClaims = (rCert = .defaultRCert,
    rClaims = .defaultRClaims, sigShare = 0x"00");
const defaultProposal :Proposal = (pClaims = .defaultPClaims,
    signature = 0x"00");
const defaultNHClaims :NHClaims = (proposal = .defaultProposal,
    sigShare = 0x"00");
const defaultPreVote :PreVote = (proposal = .defaultProposal,
    signature = 0x"00");
const defaultPreCommit :PreCommit = (proposal = .defaultProposal,
    signature = 0x"00", preVotes = 0x"00");
const defaultPreVoteNil :PreVoteNil = (rCert = .defaultRCert,
    signature = 0x"00");
const defaultPreCommitNil :PreCommitNil = (rCert = .defaultRCert,
    signature = 0x"00");
const defaultNextRound :NextRound = (nRClaims = .defaultNRClaims,
    signature = 0x"00");
const defaultNextHeight :NextHeight = (nHClaims = .defaultNHClaims,
    signature = 0x"00", preCommits = 0x"00");
const defaultBlockHeader :BlockHeader = (bClaims = .defaultBClaims,
    sigGroup = 0x"00", txHshLst = 0x"00");

struct RClaims {
    chainID @0 :UInt32 = .defaultChainID;
    height @1 :UInt32 = .defaultHeight;
```

```
    round @2 :UInt32 = .defaultRound;
    prevBlock @3 :Data = 0x"00";
}

struct RCert {
    rClaims @0 :RClaims = .defaultRClaims;
    sigGroup @1 :Data = 0x"00";
}

struct BClaims {
    chainID @0 :UInt32 = .defaultChainID;
    height @1 :UInt32 = .defaultHeight;
    prevBlock @2 :Data = 0x"00";
    txCount @3 :UInt32 = .defaultNumberTransactions;
    txRoot @4 :Data = 0x"00";
    stateRoot @5 :Data = 0x"00";
    headerRoot @6 :Data = 0x"00";
}

struct PClaims {
    bClaims @0 :BClaims = .defaultBClaims;
    rCert @1 :RCert = .defaultRCert;
}

struct Proposal {
    pClaims @0 :PClaims = .defaultPClaims;
    signature @1 :Data = 0x"00";
    txHshLst @2 :Data = 0x"00";
}

struct PreVote {
    proposal @0 :Proposal = .defaultProposal;
    signature @1 :Data = 0x"00";
}

struct PreVoteNil {
    rCert @0 :RCert = .defaultRCert;
    signature @1 :Data = 0x"00";
}

struct PreCommit {
    proposal @0 :Proposal = .defaultProposal;
    signature @1 :Data = 0x"00";
    preVotes @2 :Data = 0x"00";
}

struct PreCommitNil {
```

```
    rCert @0 :RCert = .defaultRCert;
    signature @1 :Data = 0x"00";
}

struct NRClaims {
    rCert @0 :RCert = .defaultRCert;
    rClaims @1 :RClaims = .defaultRClaims;
    sigShare @2 :Data = 0x"00";
}

struct NextRound {
    nRClaims @0 :NRClaims = .defaultNRClaims;
    signature @1 :Data = 0x"00";
}

struct NHClaims {
    proposal @0 :Proposal = .defaultProposal;
    sigShare @1 :Data = 0x"00";
}

struct NextHeight {
    nHClaims @0 :NHClaims = .defaultNHClaims;
    signature @1 :Data = 0x"00";
    preCommits @2 :Data = 0x"00";
}

struct BlockHeader {
    bClaims @0 :BClaims = .defaultBClaims;
    sigGroup @1 :Data = 0x"00";
    txHshLst @2 :Data = 0x"00";
}
```

C Technical Cryptography

In this section we present in detail the specifics of the cryptography we will be using for MadNetwork. At times we will be verbose in our algorithmic details and design choices to allow for others to understand our decisions.

We will begin by comparing this work with the Ethereum Distributed Key Generation whitepaper [23] in Sec. C.1; our design is based this paper. Some of the implementation-specific details are covered in Sec. C.2. In Sec. C.3, we discuss the mathematics related to pairing-based cryptography; this is integral to our work as it is required for our group signatures. We describe the distributed key generation protocol in Sec. C.4; the specific method of shared secret encryption is described in Sec. C.5. We discuss how to construct group signatures in Sec. C.6. Group signatures from pairing-based cryptography require a hash-to-curve function, and we talk about our construction based on [10, 27] in Sec. C.7.

We follow [23] in our definition of (t, n)-thresholded system, where we need $t + 1$ actors for consensus. Unfortunately, this is different than what was used previously, where (t, n)-thresholded system meant t actors were needed to agree. We keep this difference for ease of comparison with the referenced paper.

C.1 Comparision with Ethereum Distributed Key Generation Paper

In [23], the authors presented the Ethereum Distributed Key Generation whitepaper. Here, participants work together to form a master public key (the public key for the group) based on verifiable secret sharing. The master secret key (the private key for the group) is the summation of the shared secrets correctly shared by valiadators. This system may proceed even with a limited number of Byzantine actors. In the end, participants may compute partial signatures of a message which may be combined to form a valid group signature.

While we follow the procedure of [23] in general, there are other concerns that must be taken into consideration. The MadNetwork will be a sidechain of Ethereum, and the keys we construct will be used to sign the blocks of our sidechain. By anchoring into Ethereum, we are able to use smart contracts to enforce compliance with the consensus algorithm as well as punish those who behave in a cryptographically-verifiable malicious way. Malicious behavior include submitting false keys, submitting false proofs, signing invalid messages, and similar actions. This is different from the original paper where everything happened on Ethereum and participants who acted maliciously during the key derivation stage would still be allowed to proceed because honest actors would be able to work together to derive the correct information. In that setting, the focus was having a (t, n)-thresholded system whereby $t+1$ actors are required to work together to sign messages for the group. Here, t and n are preassigned. In our case, we specifically desire a Byzantine-fault tolerant thresholded system, whereby we require $t = \lceil 2n/3 \rceil - 1$. Even though we have the same t values for $n = 3k$ and $n = 3k - 1$ (thereby potentially allowing one malicious validator to be ejected without forcing a required restart), we will restart the DKG process whenever malicious behavior is cryptographically proven. By forcing a restart upon pernicious action, validators are discouraged from malicious activity as well as lose the opportunity to earn block rewards. We also restart when validators fail to submit the required information. In this case when

malicious intent cannot be proven due to the possiblity of technical failure, a minor fine will be given due to time cost of the other participants.

During the DKG process, all of the material necessary to recover a participant's secret is available provided enough actors work together. As mentioned in [23], this is useful in order to allow for the DKG to continue even if certain participant's fail to cooperate, but the problem is that with this secret information it would be possible for a large enough malicious subset (specifically, a majority greater than two-thirds) to recover a secret and produce valid signatures from participant P_i proving malicious behavior that is not, in fact, perpetrated by P_i. This is of serious concern because participants stake tokens on the basis of being validators on our blockchain and receiving block rewards for their computation and are threatened with losing stake should they behave nefariously.

Requiring at least 67% percent of the validators to work together in order to produce stake-burning results is better than a 51% Attack which can occur in Proof-of-Work blockchains but it leaves something to be desired; we would like for honest validators to be immune to the previously-mentioned behavior even if there is only one honest participant. To combat this, we will require all messages to be signed by a participant's Ethereum's private key. This will allow all messages to be validated against the Ethereum public key and will be safe so long as the Ethereum private key is secure. In this way, any secret information leaked during the distributed key generation will not enable Byzantine actors to produce cryptographic proof of malicious behavior against honest participants.

C.2 Specific Implementation Details

At this point in time, the Ethereum Virtual Machine (EVM) only allows for certain elliptic curve operations to be performed inexpensively from precompiled contracts. In order to keep gas costs low, we will rely heavily on these contracts. The precompiled contracts are ECAdd, ECMul, and PairingCheck; see Alg. 8 for specifics. While they have the names addition and scalar multiplication, we will primarily be using multiplicative notation in this whitepaper; the exception will be when we talk about hash-to-curve algorithms in Sec. C.7. Furthermore, we may not explicitly reference ECAdd and ECMul when using them in our algorithms for space considerations, although we will use comments to make this clear.

We require elliptic curves with pairing-based cryptography. The source code for curve `bn256` is implemented for Ethereum here[1]. This code implements the Optimal Ate pairing from [19] and is based on Barreto-Naehrig curves [4], a family of pairing-friendly curves. A more recent version of Cloudflare's library can be found here[2]. Cloudflare's version includes a hash-to-curve function based on [10], which is the same paper we base our hash function on. Unfortunately, the newer version uses a different prime; the Ethereum library uses a 254-bit prime while the current Cloudflare library uses a 256-bit prime. We modified the Ethereum code and included functions for computing modular square roots as well as a hash-to-curve algorithm. We plan to make these updates available to others, as they will be useful whenever cryptographic signatures are used. Our code includes an implementation for the hash-to-curve algorithms described in [10]; see Sec. C.7.

[1] https://github.com/ethereum/go-ethereum/tree/master/crypto/bn256/cloudflare
[2] https://github.com/cloudflare/bn256

Algorithm 8 Precompiled Ethereum Contracts for Elliptic Curves

function ECADD(a_1,a_2) ▷ $a_1, a_2 \in \mathbb{G}_1$
    return $a_1 + a_2$ ▷ Elliptic curve addition
end function

function ECMUL(a,k) ▷ $a \in \mathbb{G}_1, k \in \mathbb{F}_p$
    return $[k]\, a$ ▷ Elliptic curve addition
end function

function PAIRINGCHECK(a_0,b_0,$\cdots$,a_{k-1},b_{k-1}) ▷ $a_i \in \mathbb{G}_1, b_i \in \mathbb{G}_2$
    $t = 1$ ▷ $t \in \mathbb{F}^*_{p^{12}}$
    for $i = 0$; $i < k$; i++ **do**
        $t = t \cdot e\,(a_i, b_i)$
    end for
    if $t = 1$ **then**
        return `true`
    else
        return `false`
    end if
end function

The underlying field operations perform arithmetic modulo a prime number. This can be difficult to perform quickly [15, Chapter 14]. The `bn256` library uses Montgomery encoding [18] for efficiency; we will give an overview here although one reference is [15, Chapter 14.3.2]. The primary advantage of using the Montgomery encoding is that modular multiplication requires only multiplication with a potential subtraction; in particular, it does not require division.

If we want to perform multiplication modulo p, let $R > p$ such that $\gcd(R, p) = 1$ and it is convenient to perform modular arithmetic with respect to R. Given $x \in \mathbb{Z}_p$, the Montgomery encoding of x is

$$\tilde{x} \equiv xR \mod p. \tag{C.1}$$

Given two encoded values $\tilde{a}$ and $\tilde{b}$, Montgomery multiplication allows us to compute $\widetilde{ab}$:

$$\widetilde{ab} = \tilde{a}\tilde{b}R^{-1} \mod p. \tag{C.2}$$

Here, $R^{-1}R = 1 \mod p$, so R^{-1} is the multiplicative inverse of R with respect to p. An efficient implementation of Montgomery multiplication allows us to use it both for encoding and decoding, and this is what the `bn256` library does.

We now look at the elliptic curves in the pairing-based cryptography. Specifically, we have the elliptic curve $E/\mathbb{F}_p$ where

$$E : y^2 = x^3 + ax + b \tag{C.3}$$

and constants

$$
\begin{aligned}
p &= \texttt{30644E72 E131A029 B85045B6 8181585D 97816A91 6871CA8D 3C208C16 D87CFD47} \\
&= 36u^4 + 36u^3 + 24u^2 + 6u + 1 \\
q &= \texttt{30644E72 E131A029 B85045B6 8181585D 2833E848 79B97091 43E1F593 F0000001} \\
&= 36u^4 + 36u^3 + 18u^2 + 6u + 1 \\
u &= 4965661367192848881 \\
a &= 0 \\
b &= 3 \\
g_1 &= (1, 2) \\
h_1 &= (h_{1,x}, h_{1,y}) \\
h_{1,x} &= \texttt{081D36DE F693881E DFC5614E AE25BB5C 228A7142 A36EE533 47B09434 1D541F2C} \\
h_{1,y} &= \texttt{2CD20C36 14D407F3 39B9BB25 EF23979C D2EE1E45 310EB0C5 023A3F5F D52D8B11}
\end{aligned}
\tag{C.4}
$$

From here, we see $p = 3 \mod 4$ and $p = 1 \mod 6$. If $E(\mathbb{F}_p)$ is the group of points on $E/\mathbb{F}_p$ acting under addition, then $E(\mathbb{F}_p) = \langle g_1 \rangle$ and $|E(\mathbb{F}_p)| = q$; here, g_1 the standard generator. From the construction of BN curves, q is prime, which implies that any nontrivial element of $\mathbb{G}_1$ is a generator. During the distributed key generation protocol, we will need another generator h_1 for $E(\mathbb{F}_p)$ such that $\operatorname{dlog}_{g_1} h_1$ is unknown, which ensures no one is able to bias the underlying probability distribution of the master public key. Here, we set

$$
h_1 = \textsc{HashToG1}(\texttt{[]byte("MadHive Rocks!")}), \tag{C.5}
$$

where HashToG1 is the hash-to-curve function from Alg. 16 developed in Sec. C.7. In the future h_1 may change on each iteration of the DKG protocol, but it is constant at this point.

We let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be our nondegenerate bilinear map over groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$. We let $\mathbb{G}_1 = E(\mathbb{F}_p)$, $\mathbb{G}_2 \subseteq E(\mathbb{F}_{p^{12}})$, and $\mathbb{G}_T \subseteq \mathbb{F}^*_{p^{12}}$; e is the Optimal Ate pairing [4]. By design, we have $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T| = q$. For efficient implementation, we will need to look at the twist curve $E'/\mathbb{F}_{p^2}$ where

$$
E' : y^2 = x^3 + b'. \tag{C.6}
$$

In this case, $b' = b/\xi$ and the specific choice of $\xi \in \mathbb{F}^*_{p^2}$ will be discussed below. We then define

$$
\begin{aligned}
\psi : E'(\mathbb{F}_p^2) &\to E(\mathbb{F}_p^{12}) \\
(x', y') &\mapsto \left(z^2 x', z^3 y'\right),
\end{aligned}
\tag{C.7}
$$

and we see that ψ is an injective group homomorphism. Here, we use the definition $\mathbb{F}_{p^{12}} \equiv \mathbb{F}_{p^2}[X] / (X^6 - \xi)$, where $\xi \in \mathbb{F}_{p^2}$ is a nonsquare noncube and $z \in \mathbb{F}_{p^{12}}$ is one of the roots of $X^6 - \xi$. That $\xi \in \mathbb{F}^*_{p^2}$ exists follows from the fact $p = 1 \mod 6$; for more information see [4, Lemma 1]. It is this homomorphism ψ which allows for efficient computation, because this allows most of our arithmetic operations to occur in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$; in particular, signatures

are elements of $E(\mathbb{F}_p)$ and we represent public keys as elements of $E'(\mathbb{F}_{p^2})$. The only time arithmetic in $\mathbb{F}_{p^{12}}$ is required is when we compute the Optimal Ate pairing.

We now discuss the specific parameters for the twist curve E':

$$
\begin{aligned}
\xi &= i + 9 \\
h_{2,x,i} &= \texttt{198E9393 920D483A 7260BFB7 31FB5D25 F1AA4933 35A9E712 97E485B7 AEF312C2} \\
h_{2,x} &= \texttt{1800DEEF 121F1E76 426A0066 5E5C4479 674322D4 F75EDADD 46DEBD5C D992F6ED} \\
h_{2,y,i} &= \texttt{090689D0 585FF075 EC9E99AD 690C3395 BC4B3133 70B38EF3 55ACDADC D122975B} \\
h_{2,y} &= \texttt{12C85EA5 DB8C6DEB 4AAB7180 8DCB408F E3D1E769 0C43D37B 4CE6CC01 66FA7DAA} \\
h_2 &= \left(h_{2,x,i} i + h_{2,x}, h_{2,y,i} i + h_{2,h}\right). \qquad \text{(C.8)}
\end{aligned}
$$

The values of ξ and h_2 are from the `bn256` implementation used in Ethereum. We specify the i component before the non-i component following the library convention.

For BN curves like the one we are using, there is no known efficiently computable isomorphism $\varphi : \mathbb{G}_2 \to \mathbb{G}_1$ [21, Chap 2.2.7], which makes our setting Type 3 [8].

As noted in the Cloudflare `bn256` source code repository, the original implementation had approximately 128 bits of security, but this has been reduced due to recent work [16]. The exact security level has been discussed here [3, 17], with [3] listing the security level at 100 bits. While this is below the desired 128-bit level, it is believed to be currently out of reach. Even so, with this in mind we take precautions by enforcing regular key rotation in order to ensure the security of the system. The decrease in security is partially mitigated by the fact that validators use the Ethereum private key to sign messages throughout the consensus algorithm.

C.3 Mathematical Background and Cryptographic Definitions

We let $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ be cyclic groups of order q. Let $g_1, h_1 \in \mathbb{G}_1$ and $h_2 \in \mathbb{G}_2$ be generators and require that the discrete logarithm $\text{dlog}_{g_1} h_1$ is unknown. The groups we use were described in Sec. C.2. We let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficiently computable nondegenerate bilinear pairing. In our case, signatures will be elements of $\mathbb{G}_1$ while public keys will be elements of $\mathbb{G}_2$. See Table 1 for additional functions that will be used in our algorithms.

We are building a sidechain on top of Ethereum. All of the validators will be required to have an Ethereum public key. The DKG algorithm requires us to index the participants from 1 to n. To do so, we order the participants with respect to their sorted Ethereum public keys. Our algorithm will need an open broadcast channel; this will take place via smart contracts on the Ethereum network.

At times we will want to ensure

$$e\left(h_1^{\alpha}, h_2\right) \stackrel{?}{=} e\left(h_1, h_2^{\beta}\right). \qquad \text{(C.9)}$$

Due to how PairingCheck is defined, we will need to check the equivalent

$$e\left(h_1^{\alpha}, h_2^{-1}\right) \cdot e\left(h_1, h_2^{\beta}\right) \stackrel{?}{=} 1. \qquad \text{(C.10)}$$

$\texttt{u2b}(u)$	Convert the unsigned integer u to its bit representation
$\texttt{b2u}(\texttt{b})$	Convert the bit representation $\texttt{b}$ to its unsigned integer
$\texttt{s2b}(s)$	Convert a string s to its bit representation
$\texttt{CTEq}(x, y)$	Returns 1 when x and y are equal and 0 otherwise; runs in constant time
$\chi_p(a)$	Computes the quadratic character of a with respect to p
$H : \{0,1\}^* \to \{0,1\}^{256}$	Cryptographically secure hash function
$H_{2C} : \{0,1\}^* \to \mathbb{G}_1$	Cryptographically secure hash-to-curve function; see Sec. C.7
$\texttt{a}\|\|\texttt{b}$	Concatenation of $\texttt{a}$ and $\texttt{b}$

Table 1: List of helper functions and notation.

If $h^\alpha = (x, y)$, then $(h^\alpha)^{-1} = (x, -y)$; this holds for all elliptic curves. For ease of notation, we set $\bar{h} = h^{-1}$. It may be convenient to store both h_2 and $\bar{h}_2$.

C.4 Distributed Key Generation Protocol

We desire a Byzantine Fault Tolerant consensus algorithm. So, we let $\mathcal{P}$ be the total collection of participants with $|\mathcal{P}| = n$. We set the threshold $t = \lceil 2n/3 \rceil - 1$ in our (t, n) secret sharing protocol. Thus, it takes $t + 1$ users to reconstruct a secret, which corresponds to strictly greater than two-thirds of the participants. We assume there is an open broadcast channel between all participants. Encryption will be provided through Diffie-Hellman style shared secret encryption; this will be discussed in Sec. C.5. The group shared secret, henceforth called the *master secret key*, will be the sum of the shared secrets of each group member who correctly shared his secret. Once there are $t + 1$ valid partial signatures, these will be combined to form a group signature.

As stated above, although the final master public key will reside in $\mathbb{G}_2$, because the precompiled contracts currently available in the Ethereum Virtual Machine only allow addition and scalar multiplication in $\mathbb{G}_1$ (multiplication and exponentiation in our multiplicative notation), we will primarily use computations in $\mathbb{G}_1$ and anything required in $\mathbb{G}_2$ will be confirmed via a PairingCheck call.

C.4.1 Participant Setup

Each participant $P_i \in \mathcal{P}$ begins by selecting a secret key $\text{sk}_i \in \mathbb{Z}_q$ with public key $\text{pk}_i = g_1^{\text{sk}_i}$. The public-private key pair $\langle \text{pk}_i, \text{sk}_i \rangle$ will be used for secure communication over the insecure broadcast channel; it will not be used for signing any messages.

C.4.2 Verifiable Secret Sharing

Participant P_i chooses a secret $s_i \in \mathbb{Z}_q$ to share with the other participants. To do this, choose a secret polynomial $f_i : \mathbb{Z}_q \to \mathbb{Z}_q$ with

$$f_i(x) = c_{i0} + c_{i1}x + c_{i2}x^2 + \cdots + c_{it}x^t, \tag{C.11}$$

where $c_{i0} = s_i, c_{i1}, \cdots, c_{it}$ are chosen uniformly in $\mathbb{Z}_q$. Setting

$$C_{ik} = g_1^{c_{ik}} \quad k \in \{0, \cdots, t\}\,, \tag{C.12}$$

we have the corresponding public polynomial $F_i : \mathbb{Z}_q \to \mathbb{G}_1$:

$$F_i(x) = C_{i0} C_{i1}^x \cdots C_{it}^{x^t}. \tag{C.13}$$

The shared secret from P_i to P_j is $s_{i\to j} = f_i(j)$ and

$$\overline{\mathtt{s}}_{i\to j} = \textsc{Encrypt}(\mathrm{sk}_i, \mathrm{pk}_j, j, s_{i\to j}) \tag{C.14}$$

refers to a particular encryption scheme we discuss in Sec. C.5. Participant P_i will broadcast the message

$$\{\overline{\mathtt{s}}_{i\to 1}, \overline{\mathtt{s}}_{i\to 2}, \cdots, \overline{\mathtt{s}}_{i\to i-1}, \overline{\mathtt{s}}_{i\to i+1}, \cdots, \overline{\mathtt{s}}_{i\to n}, C_{i0}, C_{i1}, \cdots, C_{it}\} \tag{C.15}$$

over the broadcast channel. We note this message does not include the secret $\overline{\mathtt{s}}_{i\to i}$.

Once participant P_j receives the message from P_i, he sets

$$\hat{s}_{i\to j} = \textsc{Decrypt}(\mathrm{sk}_j, \mathrm{pk}_i, j, \overline{\mathtt{s}}_{i\to j}). \tag{C.16}$$

P_j then determines if

$$g_1^{\hat{s}_{i\to j}} \stackrel{?}{=} F_i(j). \tag{C.17}$$

If we have equality, then $\hat{s}_{i\to j} = s_{i\to j}$. Otherwise, P_i incorrectly shared his secret.

C.4.3 Malicious shares

We now suppose that $\overline{\mathtt{s}}_{i\to j}$ is incorrect; that is, we do not have equality in Eq. (C.17). In order to prove this to be the case, everyone needs to be able to prove that the encrypted secret $\overline{\mathtt{s}}_{i\to j}$ is incorrect. To do this, P_j must publish and prove the shared secret k_{ij}; this is required in order to ensure bad actors do not submit false proofs against honest actors.

Proving k_{ij} is the shared secret is based on showing

$$\mathrm{pk}_j = g_1^{\mathrm{sk}_j} \quad \text{and} \quad k_{ij} = \mathrm{pk}_i^{\mathrm{sk}_j} \tag{C.18}$$

without sharing the secret key sk_j; that is, we wish to show $\mathrm{dlog}_{g_1}(\mathrm{pk}_j) = \mathrm{dlog}_{\mathrm{pk}_i}(k_{ij})$ while keeping their common value (P_j's secret key sk_j) secret. To do this, we use a zero-knowledge proof; see Alg. 9 for constructing the zk-proof and Alg. 10 for proof verification. One reference for zk-proofs involving discrete logarithms is [7].

Thus, P_j would compute

$$\pi(k'_{ij}) = \mathrm{DLEQ}(g_1, \mathrm{pk}_j, \mathrm{pk}_i, k'_{ij}, \mathrm{sk}_j) \tag{C.19}$$

and publish $\langle k'_{ij}, \pi(k'_{ij})\rangle$, where k'_{ij} is claimed shared secret. This allows anyone to use DLEQ-VERIFY to determine its validity. If

$$\text{DLEQ-}\textsc{verify}(g_1, \mathrm{pk}_j, \mathrm{pk}_i, k'_{ij}, \pi(k'_{ij})) = \texttt{true}, \tag{C.20}$$

then $k'_{ij} = k_{ij}$, the shared secret. Using this, everyone can decrypt $\overline{\mathbf{s}}_{i\to j}$ by

$$\hat{s}_{i\to j} = \textsc{DecryptSS}(k_{ij}, j, \overline{\mathbf{s}}_{i\to j}, b) \tag{C.21}$$

and determine if

$$g_1^{\hat{s}_{i\to j}} \stackrel{?}{=} F_i(j). \tag{C.22}$$

If the DLEQ proof $\pi(k'_{ij})$ shows k'_{ij} is the shared secret between P_j and P_i and we do not have equality in Eq. (C.22), then P_i is acted maliciously and should be removed. There are two other possibilities: k'_{ij} is not the shared secret, or k'_{ij} is the shared secret and we have equality in Eq. (C.22). In both cases, P_j acted maliciously and should be removed. Thus, when P_j submits a claim that P_i failed to share a secret, either P_j's or P_i's stake will be slashed.

In practice, P_j will submit P_i's broadcast message to an Ethereum smart contract along with purported shared secret k'_{ij} and proof $\pi(k'_{ij})$, and the smart contract would determine its validity and burn stake as appropriate.

C.5 Shared Secret Encryption

As mentioned above, the shared secret from P_i to P_j is $s_{i\to j} = f_i(j)$. To encrypt this, we need to compute their shared secret:

$$k_{ij} = \mathrm{pk}_i^{\mathrm{sk}_j} = \mathrm{pk}_j^{\mathrm{sk}_i} = g_1^{\mathrm{sk}_i \mathrm{sk}_j}. \tag{C.23}$$

Encryption and decryption are based on the idea of a one-time pad; in particular, we use outputs of cryptographic hashes of the x-coordinate of the shared secret along with the index of the participant receiving the message as our "one-time pad". This does not meet the technical definition of a one-time pad as it is usually defined (one standard reference is [15]), but the idea is similar. By including the index of the intended recipient in the hash function, each symmetric encryption key is unique. See Alg. 11 for details.

C.6 Group Signatures

C.6.1 Constructing the Master Public Key

The goal of our consensus algorithm is to enable a Byzantine-fault tolerant subgroup to cryptographically sign on behalf of the entire group without requiring every individual group member to sign. This is enabled by signature aggregation in the appropriate way.

We let $\mathcal{Q}$ be the collection of qualified actors who correctly shared their secrets and $\mathcal{R} \subseteq \mathcal{Q}$ such that $|\mathcal{R}| = t + 1$; thus, $\mathcal{R}$ is a Byzantine-fault tolerant subgroup. As discussed in [11, 12, 23], in order to ensure that no bad actors gain any information about the master public key and not be able to change its underlying probability distribution, we require $h_1 \in \mathbb{G}_1$ such that $\mathrm{dlog}_{g_1} h_1$ is unknown. We also let $h_2 \in \mathbb{G}_2$ be a generator.

The individual shared secrets s_i allow us to define the *master secret key* msk:

$$\mathrm{msk} = \sum_{P_i \in \mathcal{Q}} s_i. \tag{C.24}$$

Algorithm 9 Zero-knowledge proof of discrete-logarithm equality.

function DLEQ(x_1,y_1,x_2,y_2,α) ▷ Construct zk-proof that $y_1 = x_1^\alpha$ and $y_2 = x_2^\alpha$.
  $w \in_R \mathbb{Z}_q$ ▷ $x_i, y_i \in \mathbb{G}_1$ with $|\mathbb{G}_1| = q$.
  $t_1 = x_1^w$
  $t_2 = x_2^w$
  $\texttt{x1} = \texttt{u2b}(x_1)$
  $\texttt{y1} = \texttt{u2b}(y_1)$
  $\texttt{x2} = \texttt{u2b}(x_2)$
  $\texttt{y2} = \texttt{u2b}(y_2)$
  $\texttt{t1} = \texttt{u2b}(t_1)$
  $\texttt{t2} = \texttt{u2b}(t_2)$
  $\texttt{c} = H(\texttt{x1}||\texttt{y1}||\texttt{x2}||\texttt{y2}||\texttt{t1}||\texttt{t2})$
  $c = \texttt{b2u}(\texttt{c})$
  $r = w - \alpha c \mod q$
  $\pi = (c, r)$
  return π
end function

Algorithm 10 Zero-knowledge verification of discrete-logarithm equality proof.

function DLEQ-VERIFY(x_1,y_1,x_2,y_2,π) ▷ Determine validity of proof from `DLEQ`
  $(c, r) = \pi$
  $t'_1 = x_1^r y_1^c$
  $t'_2 = x_2^r y_2^c$
  $\texttt{x1} = \texttt{u2b}(x_1)$
  $\texttt{y1} = \texttt{u2b}(y_1)$
  $\texttt{x2} = \texttt{u2b}(x_2)$
  $\texttt{y2} = \texttt{u2b}(y_2)$
  $\texttt{t1p} = \texttt{u2b}(t'_1)$
  $\texttt{t2p} = \texttt{u2b}(t'_2)$
  $\texttt{cp} = H(\texttt{x1}||\texttt{y1}||\texttt{x2}||\texttt{y2}||\texttt{t1p}||\texttt{t2p})$
  $c' = \texttt{b2u}(\texttt{cp})$
  if $c = c'$ **then**
    return `true`
  else
    return `false`
  end if
end function

Algorithm 11 Encryption and decryption functions

function ENCRYPT(sk,pk,j,s) ▷ Encrypt secret s to participant j; pk is j's public key
    $k = \text{pk}^{\text{sk}}$ ▷ k is the shared secret
    $\texttt{kX} = \texttt{u2b}(k_x)$ ▷ Convert x coordinate of shared secret to bytes
    $\texttt{j} = \texttt{u2b}(j)$
    $\texttt{s} = \texttt{u2b}(s)$
    $\texttt{HKj} = H(\texttt{kX}||\texttt{j})$
    $\overline{\texttt{s}} = \texttt{s} \oplus \texttt{HKj}$
    return $\overline{\texttt{s}}$
end function

function DECRYPT(sk,pk,j,$\overline{\texttt{s}}$) ▷ Decrypt secret $\overline{\texttt{s}}$ to participant j; sk is j's secret key
    $k = \text{pk}^{\text{sk}}$ ▷ k is the shared secret
    $\texttt{xK} = \texttt{u2b}(k_x)$ ▷ Convert x coordinate of shared secret to bytes
    $\texttt{j} = \texttt{u2b}(j)$
    $\texttt{HKj} = H(\texttt{kX}||\texttt{j})$
    $\texttt{s} = \overline{\texttt{s}} \oplus \texttt{HKj}$
    $s = \texttt{b2u}(\texttt{s})$
    return s
end function

function DECRYPTSS(k,j,$\overline{\texttt{s}}$) ▷ Decrypt secret $\overline{\texttt{s}}$ to participant j
    $\texttt{kX} = \texttt{u2b}(k_x)$ ▷ Convert x coordinate of shared secret to bytes
    $\texttt{j} = \texttt{u2b}(j)$
    $\texttt{HKj} = H(\texttt{kX}||\texttt{j})$
    $\texttt{s} = \overline{\texttt{s}} \oplus \texttt{HKj}$
    $s = \texttt{b2u}(\texttt{s})$
    return s
end function

This gives us the *master public key* mpk:

$$\begin{aligned} \text{mpk} &= h_2^{\text{msk}} \\ &= \prod_{P_i \in \mathcal{Q}} h_2^{s_i}. \end{aligned} \tag{C.25}$$

Because everyone in $\mathcal{Q}$ correctly shared his secret, a Byzantine-fault tolerant subgroup $\mathcal{R}$ can correctly obtain the secret s_i via Lagrange interpolation:

$$s_i = \sum_{P_j \in \mathcal{R}} s_{i \to j} R_j$$
$$R_j = \prod_{\substack{P_k \in \mathcal{R} \\ k \neq j}} \frac{k}{k-j}. \tag{C.26}$$

This would allow us to recover the secret s_i should P_i fail to share $h_1^{s_i}$ below; however, we take a stricter response and would view failure to share as malicious activity leading to stake slashing.

We now proceed to compute mpk. Let

$$\pi(h_1^{s_i}) = \text{DLEQ}(g_1, g_1^{s_i}, h_1, h_1^{s_i}, s_i) \tag{C.27}$$

be the zk-proof that $h_1^{s_i}$ is P_i's portion of the master public key (technically part of mpk^* as defined below). Because $C_{i0} = g_1^{s_i}$ is public knowledge and P_i correctly shared his secret s_i, it is possible to publicly verify $h_1^{s_i}$. Additionally, P_i will publish $h_2^{s_i}$ so that we can ensure

$$\textsc{PairingCheck}(h_1^{s_i}, \bar{h}_2, h_1, h_2^{s_i}) = 1. \tag{C.28}$$

This will be called by a smart contract. Thus, failure of P_i to publish $h_1^{s_i}$, a valid proof $\pi(h_1^{s_i})$, and the corresponding $h_2^{s_i}$ amounts to misbehavior which will lead to a fine.

The Ethereum smart contract will store $h_1^{s_i}$ from all participants and broadcast $h_1^{s_i}$, $\pi(h_1^{s_i})$, and $h_2^{s_i}$. From here, any participant will be able to submit

$$\text{mpk} = \prod_{P_i \in \mathcal{Q}} h_2^{s_i} \tag{C.29}$$

to the smart contract. Because $\{h_1^{s_i}\}_{i \in \mathcal{Q}}$ are stored and valid, the contract can construct

$$\text{mpk}^* = \prod_{P_i \in \mathcal{Q}} h_1^{s_i} \tag{C.30}$$

and call

$$\textsc{PairingCheck}(\text{mpk}^*, \bar{h}_2, h_1, \text{mpk}) \tag{C.31}$$

to ensure mpk is valid. The master public key can then be stored publicly and used for group signature verification.

C.6.2 Constructing Group Signatures

At this point, we have successfully constructed the master public key mpk for $\mathcal{Q}$ and distributed the master secret key msk among the members of $\mathcal{Q}$. We now turn our attention to constructing group signatures from partial signatures.

Each participant $P_j \in \mathcal{Q}$ has a portion of the master secret key; this is portion is called the *group secret key*:

$$\text{gsk}_j = \sum_{P_i \in \mathcal{Q}} s_{i \to j}. \tag{C.32}$$

This is possible because we proved that every participant in $\mathcal{Q}$ correctly shared his secret share. We note that that $s_{j \to j}$ is included in the sum for gsk_j even though the encrypted form was not publicly shared. Naturally, there is the corresponding *group public key*:

$$\text{gpk}_j = h_2^{\text{gsk}_j}. \tag{C.33}$$

Here, gpk_j is P_j's portion of the master public key and will be broadcast to all users. Cryptographic proof that gpk_j is valid will be discussed in the next section.

The threshold property along with the previous definitions give us the following result:

$$\text{msk} = \sum_{P_j \in \mathcal{R}} \text{gsk}_j R_j. \tag{C.34}$$

These R_j factors only depend on $\mathcal{R}$. It follows that

$$\text{mpk} = \prod_{P_j \in \mathcal{R}} \text{gpk}_j^{R_j}. \tag{C.35}$$

This will allow partial signatures to be combined into a valid group signature.

We now assume that $\mathcal{Q}$ wants to sign message M. We let $H_{2C}(M) \in \mathbb{G}_1$ be the result of a hash-to-curve algorithm; these will be discussed in Sec. C.7. In this case, participant $P_j \in \mathcal{R}$ computes the partial signature

$$\sigma_j = [H_{2C}(M)]^{\text{gsk}_j}. \tag{C.36}$$

For security, we should confirm

$$\textsc{PairingCheck}(\sigma_j, \bar{h}_2, H_{2C}(M), \text{gpk}_j) = 1 \tag{C.37}$$

to ensure we have a valid signature. If gpk_j is stored and we can call the hash-to-curve function H_{2C}, then the only inputs will be the message M and signature σ_j.

It is easy to compute the group signature from the partial signatures:

$$\sigma = \prod_{P_j \in \mathcal{R}} \sigma_j^{R_j}. \tag{C.38}$$

The R_j constants are defined in Eq. (C.26); as mentioned above, the R_j depend upon $\mathcal{R}$ and can be computed by anyone. Because $\sigma_j \in \mathbb{G}_1$, signature aggregation could be carried out by the EVM if desired; the gas cost would come from calls to ECAdd, ECMul, and modular exponentiation, but this would be expensive. We now prove this is the signature corresponding to the master public key:

$$
\begin{aligned}
e(\sigma, h_2) &= \prod_{P_j \in \mathcal{R}} e\left(\sigma_j^{R_j}, h_2\right) \\
&= \prod_{P_j \in \mathcal{R}} e\left([H_{2C}(M)]^{\mathrm{gsk}_j}, h_2^{R_j}\right) \\
&= \prod_{P_j \in \mathcal{R}} e\left(H_{2C}(M), \left[h_2^{\mathrm{gsk}_j}\right]^{R_j}\right) \\
&= e(H_{2C}(M), \mathrm{mpk}).
\end{aligned} \tag{C.39}
$$

It follows that

$$
\textsc{PairingCheck}(\sigma, \bar{h}_2, H_{2C}(M), \mathrm{mpk}) = 1. \tag{C.40}
$$

This allows every member of $\mathcal{R}$ to compute the signature for the entire group while not requiring anyone to share his signing key. This is of utmost importance for security.

C.6.3 Malicious Group Public Key Shares

We now look at how to ensure the broadcast value of gpk_j is valid.

Along with the P_j's group public key $\mathrm{gpk}_j \in \mathbb{G}_2$, there is the corresponding version in $\mathbb{G}_1$:

$$
\mathrm{gpk}_j^* = g_1^{\mathrm{gsk}_j}. \tag{C.41}
$$

Note the base is g_1 and not h_1 as in the case of mpk^* in Eq. (C.30). Participant P_j will publish a signature σ_j of a predetermined message M for security purposes. An initial check will ensure

$$
\textsc{PairingCheck}(\sigma_j, \bar{h}_2, H_{2C}(M), \mathrm{gpk}_j) = 1; \tag{C.42}
$$

any validator who provides an invalid signature is clearly malicious.

Because P_j correctly shared his secret, gpk_j^* is public knowledge, as

$$
\mathrm{gpk}_j^* = \prod_{P_i \in \mathcal{Q}} F_i(j). \tag{C.43}
$$

Thus, it could be reconstructed inside a smart contract. From there, we will ensure

$$
\textsc{PairingCheck}(\mathrm{gpk}_j^*, \bar{h}_2, g_1, \mathrm{gpk}_j) = 1. \tag{C.44}
$$

Requiring submission of gpk_j^* and the associated proof could be required at the time of submission. This would be costly, though, and will instead allow the other participants to prove malicious behavior. We would prefer the entire process to be as inexpensive as possible should there be no malicious actors.

It will be expensive to carry out this proof of malicious action. The reason is that in order to compute gpk_j^*, the message containing the shared secrets and public coefficients from every

Precompile	Gas Cost
ECAdd	150
ECMul	6000
PairingCheck	113000
ModExp	13056

Table 2: Gas cost of important precompiled contracts on the Ethereum Virtual Machine.

participant will need to be entered into a smart contract in order to obtain the right hand side of Eq. (C.43). Because of this, there is another, cheaper way to prove malicious behavior provided a Byzantine-fault tolerant group correctly submits gpk_j with valid signatures σ_j and we discuss this now.

First, a participant can go through all possible subgroups $\mathcal{R} \subseteq \mathcal{Q}$ searching for a subset whose signatures can be combined to form a valid group signature. Once a valid subset $\mathcal{R}$ is found, we can then choose $P_i \in \mathcal{Q} \setminus \mathcal{R}$ in order to determine if $\{P_i\} \cup (\mathcal{R} \setminus \{P_v\})$ can form a valid signature; if it does not, then P_i is malicious. Here $P_v \in \mathcal{R}$ is a fixed. In this case, two lists of participants would be entered into a smart contract: a list of valid participants followed by a list of invalid participants. It is straightforward to show the list of valid participants form a valid group signature. From there, malicious participant's signatures are included one at a time to show that they form invalid signatures, giving cryptographic proof of malicious behavior. Naturally, it would be a malicious action if false participant lists were submitted.

We now look at the costs of both methods. The major costs will be calling the precompiled contracts ECAdd, ECMul, PairingCheck, and ModExp (modular exponentiation); see Table 2 for the specific gas costs. Note that the cost for PairingCheck comes from our assumption that we are testing 2 pairings, while the cost for ModExp comes from the fact that all of our arguments are 256-bit (32-byte) unsigned integers. These costs come from EIP-198[3] and EIP-1108[4].

We begin with the standard method. First, to compute gpk_j^*, we must compute $F_i(j)$:

$$F_i(j) = C_{i0} C_{i1}^{j} C_{i2}^{j^2} \cdots C_{it}^{j^t}. \tag{C.45}$$

The cost of computing $F_i(j)$ is dominated by t calls to ECMul. This will be done for each element in $\mathcal{Q}$, so the main computation is nt calls to ECMul in addition to a call to PairingCheck. Thus, we see

$$\text{Cost of Standard Proof} \sim 113000 + 4000n^2. \tag{C.46}$$

For $n = 20$, this corresponds to 1.7M gas; the gas limit is 10M.

We now look at the cost of the group method. We will ignore the initial cost of ensuring a valid group signature from $\mathcal{R}$. The computation consists of forming

$$\sigma = \prod_{P_j \in \mathcal{R}_k^*} \sigma_j^{R_j}, \tag{C.47}$$

[3] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-198.md
[4] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1108.md

where

$$\mathcal{R}_k^* = (\mathcal{R} \setminus P_v) \cup \{P_k\} \tag{C.48}$$

for $P_v \in \mathcal{R}$ fixed and $P_k \in \mathcal{I}$, the set of participants who incorrectly shared gpk_j. From there, we would call

$$\textsc{PairingCheck}(\sigma, \bar{h}_2, H_{2C}(M), \text{mpk}) \tag{C.49}$$

to ensure that this is an invalid signature, and the initial check confirming $\mathcal{R}$ forms a valid signature implies that P_k is malicious.

Each group signature σ will require forming R_j and computing $\sigma_j^{R_j}$. After computing R_j, we require one ECMul call to compute $\sigma_j^{R_j}$. The expensive part turns out to be forming R_j, and we focus on its straightforward computation. We recall from Eq. (C.26) that

$$R_j = \prod_{\substack{P_k \in \mathcal{R} \\ k \neq j}} \frac{k}{k-j}. \tag{C.50}$$

Naively, we would need to compute t finite field inversions, which corresponds to t ModExp calls, to compute R_j. This must be performed $t+1$ times to form σ; we must also call ECMul $t+1$ times but this is negligible. Finally, we must call

$$\textsc{PairingCheck}(\sigma, \bar{h}_2, H_{2C}(M), \text{mpk}); \tag{C.51}$$

this call must fail for invalid signatures. Thus, we have

$$\text{Cost of Naive Group Proof} \sim 113000 + 5800n^2. \tag{C.52}$$

For $n = 20$, this corresponds to 2.4M gas. This is more expensive than the previous version, but we can do better.

The main cost comes from the finite field inversions, which require a call to ModExp. To fix this, we precompute these inverses and include them in the function call. At the beginning of the call, we check to make sure that the submitted inverses are valid; if they are valid, we proceed with the accusation, and if they are invalid, we stop. Everything else is the same as before; see Alg. 12 for the complete description. For each accusation, the main cost is $t+1$ ECMul calls as well as one pairing check. Now, these same computations must be included for the initial check of a valid group signature, but we will ignore that for now. Thus, we have the cost to prove malicious action for one participant:

$$\text{Cost of Efficient Group Proof} \sim 113000 + 4000n. \tag{C.53}$$

For $n = 20$, this corresponds to 200K gas. This is more efficient than computing gpk_j^* via smart contract.

Due to the costs, validators would always prefer to use the group method over submitting all of the sent messages. Even so, submitting all messages will always work, while in the group setting we require a Byzantine-fault tolerant set of honest validators.

Algorithm 12 Accusation against malicious gpk_j shares using group signatures.

```
1: function GrpAccGPKj(invArray, honestIndices, dishonestIndices)
2:     if Length(honestIndices) < t + 1 then
3:         return                ▷ Require t + 1 validators to make a valid signature
4:     end if
5:     validInvs = CheckInverses(invArray)
6:     if validInvs ≠ true then
7:         return                ▷ Did not submit valid multiplicative inverses
8:     end if
9:     IndexArray = honestIndices[0 : t + 1]        ▷ Include first t + 1 participants
10:    σ = AggregateSignatures(IndexArray)
11:    validSig = PairingCheck(σ, h̄_2, H_2C(M), mpk)
12:    if validSig ≠ true then
13:        return            ▷ honestIndices do not form a valid group signature
14:    end if
15:    for i = 0; i < Length(dishonestIndices); i++ do
16:        IndexArray[t] = dishonestIndices[i]
17:        σ = AggregateSignatures(IndexArray)
18:        validSig = PairingCheck(σ, h̄_2, H_2C(M), mpk)
19:        if validSig ≠ false then
20:            return            ▷ dishonestIndices[i] submitted valid signature
21:        end if
22:    end for
23: end function
```

Algorithm 13 Determine if submitted inverses are correct

```
1: function CheckInverses(invArray)
2:     validInvs = true          ▷ Assuming the form [1^{-1}, 2^{-1}, ··· , (n − 1)^{-1}]
3:     for i = 0; i < Length(invArray); i++ do
4:         k = i + 1
5:         k_inv = invArray[i]
6:         r = k_inv · k  mod q
7:         if r ≠ 1 then
8:             validInvs = false
9:             break
10:        end if
11:    end for
12:    return validInvs
13: end function
```

Algorithm 14 Compute group signature

function AGGREGATESIGNATURES(INDEXARRAY, INVARRAY)
    $\sigma = \text{IDENTITY}$ ▷ Set to identity element of $\mathbb{G}_1$
    for `idx` $= 0$; `idx` $< t + 1$; `idx`++ **do**
        $j = \text{INDEXARRAY}[\texttt{idx}]$
        $R_j = \text{COMPUTERJ}(j, \text{INDEXARRAY}, \text{INVARRAY})$
        $t = \text{ECMUL}(\sigma_j, R_j)$ ▷ σ_j is j's stored signature
        $\sigma = \text{ECADD}(\sigma, t)$
    end for
    return σ
end function

Algorithm 15 Compute R_j for group signature

function COMPUTERJ(j, INDEXARRAY, INVARRAY)
    $R_j = 1$
    for `idx` $= 0$; `idx` $<$ LENGTH(INDEXARRAY); `idx`++ **do**
        $k = \text{INDEXARRAY}[\texttt{idx}]$
        $t = k$
        if $k = j$ **then**
            continue
        else if $k > j$ **then**
            $\alpha = k - j$
        else
            $t = (-1) \cdot t \mod q$
            $\alpha = j - k$
        end if
        $t_{\text{inv}} = \text{INVARRAY}[\alpha - 1]$
        $t = t \cdot t_{\text{inv}} \mod q$
        $R_j = R_j \cdot t \mod q$
    end for
    return R_j
end function

C.7 Hash-to-curve Functions

The contents of this section should not be considered a full proof of security but are for implementation reference. Formal treatment of the security proofs for hash-to-curve functions are handled in referenced works.

We turn our attention to the hash-to-curve functions implemented in our system. The contents of this section are described hereafter. First, we review a mechanism often employed to solve this problem. Then, we briefly discuss why this solution is not ideal in our setting. Finally, we introduce the basis of our implementation and proceed into the definition of the algorithms used in our system after a brief review of some required mathematical operations. All algorithms may be found at the end of this section.

The seemingly standard, non-deterministic method to perform a hash-to-curve operation in the elliptic curve setting is to use the "MapToGroup" method as presented in the original BLS short signature paper [6]. In this method values are hashed into $\mathbb{F}_p$ by modular arithmetic with a concatenated counter. This counter starts at a fixed value, and is incremented until a value is found that creates a valid point on the specified elliptic curve. While this may be sufficient in some instances, we prefer deterministic methods due to the need for bounded computational overhead. In our system, the need for bounded computational overhead arises from a desire to allow an Ethereum smart contract to perform the hash-to-curve operations with bounded gas consumption. Deterministic methods also have the benefit of minimizing side-channel attacks in those algorithms that require such protection.

The hash-to-curve implementation selected allows the problem space to be divided into independent problems such that their solutions may be composed. Specifically, we first hash to the base field (hash map $\mathfrak{h} : \{0,1\}^* \to \mathbb{F}$) and then find a deterministic map from the base field to the elliptic curve (function $f : \mathbb{F} \to E(\mathbb{F})$). This approach allows for a separation of concerns and has become a standard approach to the problem of hashing into an elliptic curve [10, 14, 27]. Although this strategy does offer a simplified view of the problem, mapping from the base field to the elliptic curve is nontrivial. Additionally, it is frequently the case that f is not surjective, but we can overcome this limitation to obtain a surjective hash-to-curve algorithm under easily-satisfied conditions [10, 24]. Specifically, we use domain-separation in order to obtain independent hash functions. These independent hash functions allow us overcome a non-surjective mapping. We will fully address this concern later in this document.

Our discussion of hashing functions follow [10, 27]. We ask the reader to note that in the remaining work of this section, we view the group $E(\mathbb{F}_p)$ additively unless otherwise specified. We highlight this fact because this is different from the multiplicative notation used previously in this document. Before we define our implementation of the hash-to-curve algorithms employed, we feel a review of the mathematics would benefit the reader. Thus, we first review the mathematics necessary to perform the hash-to-curve operations. After this review, we present the algorithm for hashing to $\mathbb{G}_1$ and then present the algorithm for hashing to $\mathbb{G}_2$.

For reference, the full HashToG1 algorithm can be found in Alg. 16 and the full algorithm for HashToG2 can be found in Alg. 21.

C.7.1 Inverses, Square Roots, and Legendre Symbols in $\mathbb{F}_p$

In this section we review some finite field mathematics that are important in our hash-to-curve setting.

We begin by reviewing inversion in $\mathbb{F}_p$. First, we recall $\mathbb{F}_p^* \equiv \mathbb{F}_p \setminus \{0\}$, the nonzero elements of our finite field. For $a \in \mathbb{F}_p^*$, we have Euler's formula:

$$a^{p-1} = 1 \mod p. \tag{C.54}$$

This implies $a^{-1} \mod p = a^{p-2}$. We acknowledge more efficient methods for computing modular inverses are possible, but exponentiation can easily be performed in constant time, which is a goal of our implementation. This concludes our discussion of computing inverses in $\mathbb{F}_p$.

We now review the mechanisms for computing Legendre symbols. We recall that $a \in \mathbb{F}_p^*$ is a *quadratic residue* if there is $x \in \mathbb{F}_p$ such that $x^2 = a \mod p$; otherwise, a is a called a *quadratic nonresidue*. This allows us to define the Legendre symbol [13]:

$$\chi_p(a) \equiv \begin{cases} 1 & \text{if there } a \text{ is a quadratic residue modulo } p. \\ -1 & \text{if there } a \text{ is a quadratic nonresidue modulo } p. \\ 0 & \text{if } p \mid a \end{cases} \tag{C.55}$$

The Legendre symbol has a simple formula:

$$\chi_p(a) = a^{\frac{p-1}{2}}. \tag{C.56}$$

This formula holds even when $a = 0$. This concludes our discussion of computing the Legendre Symbol in $\mathbb{F}_p$.

Computing square roots is more challenging. We now review the mechanism of computing square roots. In our case $p = 3 \mod 4$, so there is a simple formula to solve $x^2 = a$:

$$x = a^{\frac{p+1}{4}}. \tag{C.57}$$

This relies on the assumption that a is a quadratic residue, so $a^{\frac{p-1}{2}} = 1$. This concludes our discussion of computing the square roots in $\mathbb{F}_p$.

Taking inverses, Legendre symbols, and square roots gives us the necessary tools to hash to $\mathbb{G}_1$. This ends the preliminary work necessary to develop HashToG1.

C.7.2 Inverses, Square Roots, and Legendre Symbols in $\mathbb{F}_{p^2}$

The previous section reviewed the mathematics of inverses, square roots, and Legendre Symbols in $\mathbb{F}_p$. We must be able to perform the same operations in $\mathbb{F}_{p^2}$ if we wish to hash to $\mathbb{G}_2$. While computing square roots and inverses may be familiar to the reader in $\mathbb{F}_p$, the operations require special handling in $\mathbb{F}_{p^2}$. We review these mechanisms now.

We first address the problem of computing inverses in $\mathbb{F}_{p^2}$. Our discussion and methods follow [2], and we present their general results applied to our particular case. As mentioned above, the setting of our elliptic curve is $p = 3 \mod 4$. This implies -1 is a quadratic nonresidue. Thus, we have the following isomorphism:

$$\mathbb{F}_{p^2} \simeq \mathbb{F}_p[i]/\left(i^2+1\right). \tag{C.58}$$

This is the analogous to constructing the complex numbers $\mathbb{C}$ from the real numbers $\mathbb{R}$. Our implementation uses this construction. We may use this construction to compute an efficient inversion as follows:

$$(a_0 + a_1 i)^{-1} = \frac{a_0 - a_1 i}{a_0^2 + a_1^2}. \tag{C.59}$$

The main computational cost of this operation is the inversion of an element in $\mathbb{F}_p$. This concludes our discussion of computing inverses in $\mathbb{F}_{p^2}$.

We now begin our discussion of computing the Legendre Symbol in $\mathbb{F}_{p^2}$ If $a = a_0 + a_1 i \in \mathbb{F}_{p^2}$, then a is a quadratic residue in $\mathbb{F}_{p^2}$ if and only if $a_0^2 + a_1^2$ is a quadratic residue in $\mathbb{F}_p$. The main computational cost of this operation is the computation of the Legendre symbol of an element in F_p. From the above we have the Legendre symbol in $\mathbb{F}_{p^2}$, which we denote as LegendreFP2 or $\chi_{p^2}(\cdot)$; this particular algorithm is presented in Alg. 19. This concludes our discussion of computing the Legendre Symbol in $\mathbb{F}_{p^2}$.

We will now look at computing square roots in $\mathbb{F}_{p^2}$. The main idea is to find $b \in \mathbb{F}_{p^2}$ and odd s such $b^2 a^s = 1$. In this case, we see

$$\begin{aligned}\left[ba^{\frac{s+1}{2}}\right]^2 &= b^2 a^{s+1} \\ &= a.\end{aligned} \tag{C.60}$$

If we set

$$\begin{aligned} b &= \left(1 + a^{\frac{p-1}{2}}\right)^{\frac{p-1}{2}} \\ s &= \frac{p-1}{2}, \end{aligned} \tag{C.61}$$

then when $b \neq 0$, we have $b^2 a^s = 1$. When $b = 0$, we have $a^{\frac{p-1}{2}} = -1$. In this case, our square root is $ia^{\frac{p+1}{4}}$. This procedure is formally presented in Alg. 20. The main computational cost of this operation is two exponentiations in $\mathbb{F}_{p^2}$. This concludes our discussion of computing square roots in $\mathbb{F}_{p^2}$ and the preliminary review of those operations necessary to develop HashToG2.

C.7.3 Hashing to Base

In this section we will describe the hash-to-base operation. We begin by discussing the construction of a random oracle into $\mathbb{F}_p$ using a single cryptographic hash function with domain separation. We then bound the deviation from uniformity in this operation.

In the following H is a 256-bit hash function. In our implementation we use Keccak256. Note this is the Sha3 variant used by Ethereum that differs from the NIST approved Sha3 hash function due to a difference in the handling of padding.

Let H act as a random oracle. We map from H to $\mathbb{Z}$ by interpreting the output of a cryptographic hash function as a big endian unsigned integer. This can be seen in lines 4 and 5 of HashToBase as the function named `b2u`. Although we may naively map from $\mathbb{Z}$ to $\mathbb{Z}_p$ by simply taking the ouput of H modulo p, this would be insecure in our setting due to the nonuniformity of this operation. In order to ensure the mapping from $\mathbb{Z}$ to $\mathbb{Z}_p$ is more uniformly distributed, domain separation is utilized. Specifically, we use domain separation in order obtain independent hash functions from H through concatenation of constants with the message being hashed. These independent hash functions allow us to create a secure 512-bit random number from a single cryptographic hash function. The full explanation of this operation is below.

Let $\text{HashToBase} : \{0,1\}^* \times \{0,1\}^8 \times \{0,1\}^8 \to \mathbb{F}_p$ denote our random oracle into the underlying field $\mathbb{F}_p$. In the HashToBase operation the first component corresponds to the underlying message being hashed, while the last two elements provide the necessary domain separation. See Alg. 17 for the full implementation.

Due to the fact p is prime and not a power of 2, there will be some nonuniformity in the resulting distribution of HashToBase. We investigate this nonuniformity now.

First, we assume that $H : \{0,1\}^* \to \mathbb{Z}_N$ is a random oracle and $p \in \{1, 2, \cdots, N-1\}$. We want to determine how much $H(m) \mod p$ deviates from uniformity. We restrict ourselves to the case when $p \nmid N$. Let

$$N = qp + r \tag{C.62}$$

with $0 \leq r < p$ and $q \geq 1$. Because $p \nmid N$, we have $r \geq 1$. Let X be uniformly distributed on $\mathbb{Z}_N$ and set $X_p = X \mod p$. Furthermore, we let U_p be the uniform distribution modulo p. Then

$$\mathcal{P}\left(X_p \in \{0, \cdots, r-1\}\right) = \frac{q+1}{N} \tag{C.63}$$

and

$$\mathcal{P}\left(X_p \in \{r, \cdots, p-1\}\right) = \frac{q}{N}. \tag{C.64}$$

Here, $\mathcal{P}$ denotes the probability of an event occurring. We now determine the deviation of X_p from the uniform distribution U_p:

$$\begin{aligned}
\Delta\left(X_p, U_p\right) &\equiv \sum_{k=0}^{p-1} \left|\mathcal{P}(X_p = k) - \mathcal{P}(U_p = k)\right| \\
&= \sum_{u=0}^{r-1} \left|\frac{q+1}{N} - \frac{1}{p}\right| + \sum_{u=r}^{p-1} \left|\frac{q}{N} - \frac{1}{p}\right| \\
&= r\frac{qp + p - N}{pN} + (p-r)\frac{N - qp}{pN} \\
&\leq \frac{p}{N}.
\end{aligned} \tag{C.65}$$

From this, we see that if p is a k-bit prime and $N = 2^{k+\ell}$, then the deviation from uniformity is $\leq 2^{-\ell}$.

In our case, p is a 254-bit prime and we concatenate the output of independent hash functions in order to have a uniformly distributed 512-bit output. From the above it may be seen that HashToBase produces outputs whose deviation from uniformity is less than 2^{-258}. This deviation is sufficiently small as to not be of concern. Further work is required to more formally prove this assumption, but this work is not included at this time.

C.7.4 Base to $\mathbb{G}_1$

In this section we discuss the construction of a hash-to-G1 algorithm. We begin by noting the non-surjective nature for many functions $f : \mathbb{F}_p \to E(\mathbb{F}_p)$. Then, we cite a known solution to this problem and provide necessary mathematics to understand the fundamental operation that overcomes the problem. Next, we discuss the actual implementation. Lastly, we discuss specific exclusion of three points from the allowable outputs of this algorithm for security purposes.

Let us now suppose that we have a hash function $\mathfrak{h} : \{0,1\}^* \to \mathbb{F}_p$ and a deterministic map $f : \mathbb{F}_p \to E(\mathbb{F}_p)$. As has been previously stated, in many situations [10, 14, 27] f is not surjective. That is, there are points on the elliptic curve $E(\mathbb{F}_p)$ (sometimes a nontrivial fraction) which cannot be reached by f. Even so, if we use domain separation to construct independent hash functions, $\mathfrak{h}_1, \mathfrak{h}_2 : \{0,1\}^* \to \mathbb{F}$, then

$$F(\mathfrak{m}) = f(\mathfrak{h}_1(\mathfrak{m})) + f(\mathfrak{h}_2(\mathfrak{m})) \tag{C.66}$$

is indistinguishable from a random oracle on $E(\mathbb{F}_p)$ under certain restrictions on f. See [10, 24] for details.

We now turn our attention to determining $f : \mathbb{F}_p \to E(\mathbb{F}_p)$. Finding a map $f : \mathbb{F}_p \to E(\mathbb{F}_p)$ is involved. Our BN curve has the form

$$\begin{aligned} E : y^2 &= g(x) \\ &= x^3 + b. \end{aligned} \tag{C.67}$$

From [10], it possible to show there are $x_1, x_2, x_3, y \in \mathbb{F}_p$ such that

$$g(x_1)g(x_2)g(x_3) = y^2. \tag{C.68}$$

When $y \neq 0$, quadratic reciprocity implies that $g(x_i)$ is square for some i; that is, for some i we have $(x_i, \sqrt{g(x_i)}) \in E(\mathbb{F}_p)$. For uniqueness, we choose the smallest i. We will use the construction of [10] to determine such points with some modifications based on work in [27]. The exact algorithm can be found in Alg. 18.

One of the main difficulties is determining i in such a way as to not leak information; because of this, we do not wish to rely upon if statements. In [10], they suggested the function

$$\psi(r_1, r_2) = [(r_1 - 1)\, r_2 \mod 3] + 1. \tag{C.69}$$

This function works under the assumption that modular arithmetic always returns positive integers. This is not always the case in programming languages; in particular, it does not hold in Go (Golang), the language we use to implement these algorithms. To circumvent this, we use the following function, which is implemented in Line 15 of Alg. 18:

$$(r_1 - 1)(r_2 - 3)/4 + 1. \tag{C.70}$$

In [10], the authors encountered an issue when $t = 0$. Their solution was to define

$$\textsc{BaseToG1}(0) = \left(\frac{-1 + \sqrt{-3}}{2}, \sqrt{1 + b}\right). \tag{C.71}$$

Wahby and Boneh [27] suggest another method, which we implement, in order to have a more efficient algorithm. This implementation also affords the benefit of not needing to handle the case of $t = 0$ separately. This can be seen in Line 7 of Alg. 18 where we define α as the inverse of a value which depends on $t \in \mathbb{F}_p$. When $t = 0$, we compute $\alpha = 0$ because we compute inversions via exponentiation. Thus, the computational convention $0^{-1} = 0 \mod p$ is established. This leads to the same result as in Eq. (C.71) without special handling.

We have cryptographic insecurity when $\textsc{HashToG1}(\mathtt{m}) = g_1^{\alpha}$ for known α. Note we briefly switch back to multiplicative notation at this time. In practice, it is likely this insecurity will only be known when $\alpha \in \{-1, 0, 1\}$ or is sufficiently close to these values. We assume we may not fix simple proximity to these values, and thus only explicitly address the case of $\alpha \in \{-1, 0, 1\}$. For clarity, this notation specifies the hash function outputs of the identity element, the generator, or the generator's inverse. Due to the concerns around the use of these points, we will not allow these individual cases. In order to enforce this requirement we include a $\textsc{SafePointCheck}$ in $\textsc{HashToG1}$; see Line 7 in Alg. 16. This function checks the point returned from the hash function for equivalence with the identity element OR equivalence of the point's x coordinate with 1. If either of these conditions is true, an error is raised. Although the probability of a hash mapping to these points is small, this error must be handled appropriately to prevent an attacker from causing unexpected errors in a remote system due to specially crafted messages.

C.7.5 Base to $\mathbb{G}_2$

We now take up the slightly more challenging possibility of computing a hash function to $\mathbb{G}_2$. Fortunately, we may utilize many of the previously described operations. Thus, we do not repeat those explanations and only discuss those operations that differ. We would like to remind the reader that the results in Sec. C.7.2 allow us to compute Legendre symbols and square roots in $\mathbb{F}_{p^2}$.

Using the derivation of [27, Section 3], we set $u_0 = 1$ as in $\mathbb{G}_1$ to obtain

$$\begin{aligned} x_1 &= \frac{\sqrt{-3} - 1}{2} - \frac{t^2\sqrt{-3}}{t^2 + g'(1)} \\ x_2 &= -1 - x_2 \\ x_3 &= 1 - \frac{(t^2 + g'(1))^2}{3t^2}, \end{aligned} \tag{C.72}$$

for $t \in \mathbb{F}_{p^2}$. Here, we have

$$E' : y^2 = g'(x) = x^3 + b', \tag{C.73}$$

where $b' = b/\xi = 3/(i+9)$. With this choice, $-g'(1)$ is a residue in $\mathbb{F}_{p^2}$. This ensures that $t \in \{0, \pm\sqrt{-g'(1)}\}$ implies x_1 is a valid point on the curve; thus, all inputs result in valid outputs. See Alg. 22 for the algorithm. Thus, our hash functions to $\mathbb{G}_1$ and $\mathbb{G}_2$ are essentially the same.

At this point, we have successfully mapped into E'; however, our goal is to map into $\mathbb{G}_2$. From [4], we know $|E'(\mathbb{F}_{p^2})| = q(2p - q)$. This gives a cofactor $r = 2p - q$ because $p \nmid q$. We take care of this by clearing the cofactor. See Alg. 21 for the full hash algorithm.

Algorithm 16 Hash to $\mathbb{G}_1$

function HashToG1(m) ▷ Hash-to-G1 algorithm for BN curves
    $t_0 = \text{HashToBase}(\texttt{m}, \texttt{0x00}, \texttt{0x01})$
    $t_1 = \text{HashToBase}(\texttt{m}, \texttt{0x02}, \texttt{0x03})$
    $h_0 = \text{BaseToG1}(t_0)$
    $h_1 = \text{BaseToG1}(t_1)$
    $h = \text{ECAdd}(h_0, h_1)$
    SafePointCheck(h) ▷ Ensure $h \notin \{\mathcal{O}, g_1, -g_1\}$
    return h
end function

Algorithm 17 Hash to the base field F_p

function HashToBase(m,i,j)
    $\texttt{s0} = H(\texttt{i}||\texttt{m})$
    $\texttt{s1} = H(\texttt{j}||\texttt{m})$
    $s_0 = \texttt{b2u}(\texttt{s0})$
    $s_1 = \texttt{b2u}(\texttt{s1})$
    $c = 2^{256} \mod p$ ▷ Precomputed constant
    $t_0 = s_0 \cdot c \mod p$
    $t_1 = s_1 \mod p$
    $t = t_0 + t_1 \mod p$
    return t
end function

Algorithm 18 Base to $\mathbb{G}_1$

function BaseToG1(t)
  $p_1 = \left(-1 + \sqrt{-3}\right)/2$ ▷ Precomputed constant
  $p_2 = \sqrt{-3}$ ▷ Precomputed constant
  $p_3 = 1/3$ ▷ Precomputed constant
  $p_4 = g(1)$ ▷ $g(1) = 1 + b$; Precomputed constant
  $s = (p_4 + t^2)^3$
  $\alpha = \left[t^2 (p_4 + t^2)\right]^{-1}$ ▷ $\alpha = 0$ when $t = 0$
  $x_1 = p_1 - p_2 t^4 \alpha$ ▷ On curve when $\alpha = 0$
  $x_2 = -1 - x_1$
  $x_3 = 1 - p_3 s \alpha$
  $a_1 = x_1^3 + b$
  $a_2 = x_2^3 + b$
  $r_1 = \chi_p(a_1)$
  $r_2 = \chi_p(a_2)$
  $i = (r_1 - 1)(r_2 - 3)/4 + 1$
  $c_1 = \texttt{CTEq}(1, i)$
  $c_2 = \texttt{CTEq}(2, i)$
  $c_3 = \texttt{CTEq}(3, i)$
  $x = c_1 x_1 + c_2 x_2 + c_3 x_3$
  $y = \texttt{sign0}(t)\sqrt{x^3 + b}$
  return (x, y)
end function

Algorithm 19 Legendre symbol for $\mathbb{F}_{p^2}$; Alg. 8 from [2]

function LegendreFP2(a) ▷ $a = a_0 + a_1 i \in \mathbb{F}_{p^2}$
  $\alpha = a_0^2 + a_1^2$
  return $\chi_p(\alpha)$
end function

Algorithm 20 Square Root in $\mathbb{F}_{p^2}$; Alg. 9 from [2]

function SqrtFP2(a) ▷ $a \in \mathbb{F}_{p^2}$; assuming a is quadratic residue
  $t = a^{\frac{p-3}{4}}$
  $y = ta$
  $\alpha = ty$
  if $\alpha = -1$ **then**
    $x = iy$
  else
    $b = (1 + \alpha)^{\frac{p-1}{2}}$
    $x = by$
  end if
  return x
end function

Algorithm 21 Hash to $\mathbb{G}_2$

function HashToG2($\mathtt{m}$) ▷ Hash-to-G2 algorithm for BN curves
  $t_{0,0} = \text{HashToBase}(\mathtt{m}, \mathtt{0x04}, \mathtt{0x05})$
  $t_{0,1} = \text{HashToBase}(\mathtt{m}, \mathtt{0x06}, \mathtt{0x07})$
  $t_{1,0} = \text{HashToBase}(\mathtt{m}, \mathtt{0x08}, \mathtt{0x09})$
  $t_{1,1} = \text{HashToBase}(\mathtt{m}, \mathtt{0x0a}, \mathtt{0x0b})$
  $t_0 = t_{0,0}i + t_{0,1}$
  $t_1 = t_{1,0}i + t_{1,1}$
  $h_0 = \text{BaseToTwist}(t_0)$
  $h_1 = \text{BaseToTwist}(t_1)$
  $h = \text{ECAdd}(h_0, h_1)$
  $g = \text{ECMul}(h, r)$ ▷ Clearing cofactor $r = 2p - q$
  return g
end function

Algorithm 22 Base to E'

function BaseToTwist(t)
  $p_1 = \left(-1 + \sqrt{-3}\right)/2$ ▷ Precomputed constant
  $p_2 = \sqrt{-3}$ ▷ Precomputed constant
  $p_3 = 1/3$ ▷ Precomputed constant
  $p_4 = g'(1)$ ▷ $g'(1) = 1 + b'$; Precomputed constant
  $s = \left(p_4 + t^2\right)^3$
  $\alpha = \left[t^2\left(p_4 + t^2\right)\right]^{-1}$ ▷ $\alpha = 0$ when $t \in \{0, \pm\sqrt{-p_4}\}$
  $x_1 = p_1 - p_2 t^4 \alpha$ ▷ On curve when $\alpha = 0$
  $x_2 = -1 - x_1$
  $x_3 = 1 - p_3 s \alpha$
  $a_1 = x_1^3 + b'$
  $a_2 = x_2^3 + b'$
  $r_1 = \chi_{p^2}(a_1)$
  $r_2 = \chi_{p^2}(a_2)$
  $i = \left(r_1 - 1\right)\left(r_2 - 3\right)/4 + 1$
  $c_1 = \mathtt{CTEq}(1, i)$
  $c_2 = \mathtt{CTEq}(2, i)$
  $c_3 = \mathtt{CTEq}(3, i)$
  $x = c_1 x_1 + c_2 x_2 + c_3 x_3$
  $y = \mathtt{sign0}(t)\sqrt{x^3 + b'}$
  return (x, y)
end function