Plutus: Measuring message gossiping in P2P networks and providing incentives in cryptocurrencies

by

Rotem Hemo

B.Sc., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2018

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by . . . .
Yossi Gilad
Postdoctoral Researcher
Thesis Supervisor

Certified by . . . .
Nickolai Zeldovich
Professor
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chairman, Masters of Engineering Thesis Committee

Plutus: Measuring message gossiping in P2P networks and providing incentives in cryptocurrencies

by

Rotem Hemo

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we present Plutus, an efficient and game-theoretically proven incentive mechanism for Algorand, a proof-of-stake cryptocurrency. In order to operate, Algorand requires users to constantly propagate messages but has no mechanism to incentivize users to do so. Plutus solves this problem by keeping track of each message propagation path and rewarding the users who propagated messages using a lottery.

We implemented a prototype of Plutus on top of Algorand to measure the performance and overhead of Plutus. Experimental results show that with Plutus, Algorand's block confirmation time increases by only 7% and that there is no penalty on Algorand's scalability.

Thesis Supervisor: Yossi Gilad
Title: Postdoctoral Researcher

Thesis Supervisor: Nickolai Zeldovich
Title: Professor

Acknowledgments

If I have seen further it is by
standing on ye sholders of Giants

Sir Issac Newton

I am sincerely grateful to everyone who has volunteered their time to help in various stages of the process.

I would like to thank my advisor Nickolai Zeldovich, who taught me how to conduct research and always pushed me to achieve the highest standards, for the many hours poking around problems and the endless guidance with every problem I've encountered in my thesis.

I would also like to thank Yossi Gilad, who I was lucky to meet a few years ago at IBM Research. With a unique combination of an incredible work ethic, a brilliance of mind, and a kindness that is unmatched, Yossi was always there to answer any question and give advice.

I would also like to thank Georgios Vlachos for all the time he invested in helping me with the game theoretical analysis.

I would like to thank PDOS members, in particular Jon, Anish, and Atalay, who were always there to lend a hand, discuss security, or just grab a cup of tea.

I would also like to thank my dear friends — Suri, Noa, and Gal — who helped me ensure that this thesis is typo-free (let's hope they're right!) and to all my other friends who always make me feel home away from home.

Last, but definitely not least, I want to thank my parents and my siblings for their unconditional love, endless support of everything I do, and for showing me that physical distance is never a challenge.

Contents

1 Introduction **13**

1.1 Algorand 15

1.2 Plutus's Overview 16

1.3 Goals and Threat Model 16

1.4 Contributions 17

1.5 Outline 18

2 Related Work **19**

3 Design **21**

3.1 Algorand's Messages 22

3.2 Message logging 22

3.2.1 Message processing 22

3.2.2 Path verification 24

3.3 Lottery 25

3.3.1 Lottery selection 25

3.3.2 Lottery verification 27

4 Implementation **29**

5 Evaluation **31**

5.1 Method 31

5.2 Algorand as a baseline 33

5.3 Message signing and path verification 34

5.4 Lottery running and verification 36
5.5 Message Size 38

6 Game Theoretical Analysis **39**

7 Future work **43**
7.1 Performance 43
7.1.1 Cryptographic aggregators 43
7.1.2 Key Mapping 44
7.2 Additional Messages 44

8 Conclusion **45**

List of Figures

3-1 Message gossiping illustration. User M_0 issues a message, adds the destination public key PK_1 and signs the message. User M_1 repeats the process by adding the neighbors' PK, signing it and relaying it the next user. 23

5-1 Money distribution in our simulations. We assume 10^5 users with 10^9 coins in supply, distributed among the users using the power law distribution with $\alpha = 0.015$ (To simulate world's wealth distribution [20]). 32

5-2 Latency for one round of Algorand as a function of number of users 33

5-3 Comparison between Algorand's vanilla implementation and Plutus's latency for one round of Algorand as a function of the block size 34

5-4 Message path signing time as a function of the number of hops. Since we sign the hash of the path, the time is relatively constant. . . 35

5-5 Message path verification time as a function of the number of hops. The three different lines show the time it takes for different message size as issued by the original sender. As shown, the message size does not impact the verification time. 35

5-6 Lottery time as a function of the number of public keys. We sample a set of N public keys and select the winning keys with respect to their weight. 36

5-7 Lottery verification time as a function of the number of public keys. The verifier takes the set of winning keys and verifies that the lottery computation was done correctly. 37

5-8 Message size as a function of hops. The initial message size is 200Bytes. At every hop, the user adds its public key and signature to the message which increases the size of the message by roughly 96 bytes. . . 38

List of Tables

4.1 Approximate distribution of lines of code per Plutus's module . . . 29

6.1 Notations 39

Chapter 1

Introduction

Cryptocurencies act as decentralized public ledgers and can enable new applications such as smart contracts, efficient supply chain control, and reduced cost cross border transactions. However, cryptocurrencies, like most decentralized applications, use peer-to-peer networks to transfer messages and therefore need to incentivize their users to participate and operate the network.

In Proof-of-Work based cryptocurrencies, such as Bitcoin [19], the main task needed to maintain the network is block mining, where users need to repeatedly compute hashes. These cryptocurrencies incentivize their users to do so by providing them with a reward for every block they mine. However, the amount of energy required for these mining operations is expensive. Proof-of-Stake (PoS) cryptocurrencies offer an alternative solution: They are based on the amount of money users have in the network, avoiding wasteful computation. Algroand [10] is one such cryptocurrency.

Algorand has modest computational requirements compared to other cryptocurrencies as it does not require users to perform proof of work. The main cost of operating Algorand is in communication: It requires users to contribute a portion of their Internet connection bandwidth and electricity. Despite this need, at present, Algorand has no built-in incentive mechanism for incentivizing the users to do so.

This thesis presents Plutus, a novel and efficient incentive mechanism for Algorand. Plutus takes advantage of Algorands' consensus protocol by making it agree not only on users' transactions but also on the reward each user receives. These rewards are given to users who participated in the propagation of those messages, thereby incetivizing those users to propagate messages.

In designing Plutus, we had to navigate three primary challenges. First, Algorand is decentralized and as a result, there is no central entity that can record which user propagated which message. This makes it difficult to know which user to reward. Second, Algorand is a permissionless system where everyone can join and participate in the P2P network. Thus, the messages that users send go through the hands of the adversary. Therefore, Plutus must ensure that messages cannot be modified. Third, Plutus has to avoid *Sybil attacks* where an attacker creates multiple pseudonyms to "inflate" the number of messages it propagated.

Plutus addresses these challenges using the following techniques:

Message chaining - Every message, before arriving at a node, goes through several nodes in the peer-to-peer (P2P) network. In Plutus, each user "stamps" each message it propagates by using its private key to sign the message and the destination user's public key. As a result, every message contains an *immutable* and *cryptographically secure* path of users who participated in its propagation.

Weighted lottery - To discourage users from adding extra signatures to the message's path using different public keys, Plutus uses a technique inspired by Algorand. Plutus rewards users proportionally to their stake in the system. Using this technique, the expected user's reward will stay the same regardless of the number of signatures added.

We implemented a prototype of Plutus on top of Algorand and evaluated its design and performance. The experimental results show that Plutus increases Algorand's block confirmation time by 7% and can run the lottery with 20,000 public keys in 45ms.

We also analyze Plutus using techniques from game theory, showing that every user connected to the network will receive each message with high probability.

1.1 Algorand

Plutus is built on top of Algorand [10], which we briefly review here.

Algorand is a permissionless proof-of-stake blockchain. In Algorand, like in other cryptocurrencies, users communicate to agree on a sequential log of *transactions*. Like most cryptocurrencies, Algorand uses a P2P network to communicate. On average, each node is connected to 8 random peers where 4 are incoming connection and 4 are outgoing connections. Every time a node gets a message from one of the connections, it sends it through the other ones.

Every user, identified by its *Public Key*, can cryptographically sign and send transactions in the system. Users constantly aggregate these transactions into *blocks* and are selected from time to time to append these blocks to the ledger. Each block contains a list of transactions with their corresponding signature, a psuedorandom number Q (also called *round seed*, as described below) and a cryptographic hash of the previous block in the chain.

To append blocks to the ledger, Algorand works in *rounds* and the goal of each round is to propose and agree on the next block in the blockchain. Every round, one user[1] is selected as a *block proposer*, responsible for proposing the next block, and a few thousand users are selected to be *committee members*, responsible for verifying the proposed block so that it can be appended to the blockchain. If the block is invalid, committee members agree on a default, *empty block*, using a *byzantine agreement* scheme.

The frequency in which users are selected to be block proposers or committee members is directly proportional to their stake in the system. Every round, each user computes a psuedorandom value Q, which is unpredictable. Then, the user uses the Q value to seed and compute a *Verifiable Random Function (VRF)* [18] that is used to implement *cryptographic sortition*. A cryptographic sortition is a process that produces a sample of the users in the network, weighted by their stake. Running the cryptographic sorition privately, the user is the only one who

[1]In fact, a few tens are selected to be block proposers but only one eventually appends its block to the ledger. For simplicity, we describe it as if there is just one block proposer.

knows that it was selected. Weighing the users by their stake prevents an adversary from knowing which user to attack. To let the rest of the network know about the selection, the user sends a message with a proof of correctness produced by the VRF. To prevent adversarial manipulation, users in the round use the Q value from the previous round and are weighed by their stakes from the previous k^{th} block.

1.2 Plutus's Overview

Plutus extends Algorand in order to encourage users to relay messages. Plutus achieves this by rewarding users for propagating messages using two mechanisms - *Message logging* and *Lottery*.

Message logging. In Algorand, when a user issues a transaction message (for example) and sends it through a node to the network, the message reaches other nodes via intermediate nodes which act as *relayers*. A node can act as a relayer if it sends messages it gets to its peers in the network. Plutus requires the relayers to "stamp" every message they relay by signing it using their private key. This attaches to every message a list, also called *propagation path*, of relayers who propagated it.

Lottery. As mentioned above, every user in Algorand can be selected to be a block proposer and propose the next block. For this case, users aggregate transaction messages. Once a user is selected to be a block proposer, it composes the transaction messages into a proposed block. To reward the relayers for their contribution, the block proposer runs a lottery among the relayers of each message that was included in the proposed block.

1.3 Goals and Threat Model

Suppose that Alice is a user in Algorand and is running an Algorand node that is connected to Algorand's P2P network. Alice wishes to send a message to Bob, who is another user who runs a node (or a set of users who run multiple nodes). In this case, Plutus should achieve the following goals

- *Network Availability* - If Alice send a message to Bob, Bob should be able to receive the message with high probability.
- *Communication Efficiency* - Alice should be able to send the message to Bob with low cost in terms of latency and bandwidth.

Threat Model - Plutus is built on top of Algorand and as a result, we inherit Algorand's assumptions. Moreover, we assume that users are rational and would only make decisions that maximize their profit.

- *Honest Money Majority* - A fraction $h > \frac{2}{3}$ of the money in the system is held by honest users.
- *Cryptographic security* - An adversary has very high but bounded computational power. Specifically, an adversary cannot break standard cryptographic assumptions.
- *Adaptive corruptions* - An adversary can corrupt any user in the system at any given time. Although, even in this case our *Honest Money Majority* still has to hold.

1.4 Contributions

This thesis makes the following contributions:

- Design of a method to track a message's propagation path in peer-to-peer networks
- Design of Plutus, an incentive mechanism for Algorand
- Game theoretical analysis which shows that at *Nash equilibrium* each user receives a transaction with high probability
- An evaluation of Plutus's design on top of Algorand that demonstrates its performance

1.5 Outline

This thesis starts with a review of related work (§2) and continues with the design of Plutus (§3) followed by the details about the prototype implementation (§4). Then, it evaluates the performance (§5) and shows the theoretical analysis (§6). Finally, it discusses future work (§7) and concludes (§8).

Chapter 2

Related Work

This chapter discusses various attempts to develop incentive mechanisms for propagating messages in P2P networks in general and some attempts to develop incentive mechanism for cryptocurrencies in particular. The solutions are divided into three main approaches: First, direct incentives, similar to Plutus's approach, where the solution incentivizes the message propagation directly. Second, indirect approach, where the mechanism incentivizes a specific task at the blockchain layer and, as a side effect, encourages the user to propagate messages. Third, solutions that do not incentivize specific tasks but solve the problem by introducing special nodes for that purpose. We provide a brief description of each solution and analyze its limitations as compared to Plutus.

Direct incentives. Li *el at* [15] provides a general and theoretical technique for incentivizing message propagation in P2P networks. Their idea is similar to Plutus. The solution keeps track of the message's propagation path and reward the all the users along the path. Yet, this solution doubles the bandwidth requirement as it requires each relayer to reward the previews node downstream. If U_0 sends a message to U_n through some intermediate nodes, U_N should reward U_{n-1} and so forth. Plutus doesn't require the users to handle the reward themselves.

Yu *et al* [21] provide another general approach where each user connects to another user just by referrals. This way, the number of "free riders", users who consume resources but don't contribute, is reduced. This system is not resilient to Sybil attacks and also imposes a burden on new users who want to join the network.

Some other solutions [9, 13] analyze the classic P2P model and suggest optimization but not specific mechanisms.

Indirect incentives. *Solida* [3] provides incentives to their committee members and block proposers. Plutus relies on a similar idea but rewards every participant who helped create the block. Furthermore, although Solida makes use of Byzantine Agreement instead of Nakamoto consensus [19], it still employs PoW which requires significant computation.

SmartCast [14] is another incentive compatible cryptocurrency that creates a layer on top of the blockchain to reward and punish participants using smart contracts. Plutus doesn't require the support of smart contracts and employs basic cryptography.

SpaceMint [12] presents another approach, using Proof-of-Space as its consensus protocol. Similar to Plutus, this approach makes a use of game theory to prove the soundness of their scheme. However, using Proof-of-Space requires allocation of a non-trivial amount of memory from the user which limits the usability of such consensus protocol; Plutus does not have such issues.

Special purpose. Other cryptocurrencies [4, 7] do not try to incentivize specific tasks, but instead try to solve the problem specifically by introducing special nodes. This solution works but makes the network centralized and vulnerable to DOS attacks. For example, Dash [7] separates the network into two different sets of users - master nodes, and regular nodes. Furthermore, another limitation of this approach is feasibility. At this time, in order to be able run a master node, one must buy 1000 DASH coins (at the time of this writing worth approximately 450,000USD [1]). As a final example, Neo [6] creates another parallel coin to the main one to incentivize a special type of "Consensus Nodes" that run the network.

Chapter 3

Design

Plutus is composed of two main components: a *message logging* and a *lottery*. In the message logging component, each relayer, upon receiving a message from the network, verifies, signs and propagates the message to its network peers. In the lottery component, every time a user is selected to be the next block proposer, it runs a lottery among the relayers' public keys and creates new transactions that reward the relayers for their work.

Plutus also ensures some level of immutability of the paths in order to prevent a potential adversary from manipulating the paths in their favor and is resilient to Sybil attacks, an attack where an adversary creates a large number of pseudonymous identities.

This chapter discusses Plutus's design in further details and specifically addresses the following questions -

1. How does Plutus log message propagation paths?

2. How does Plutus prevent adversaries from changing message propagation paths?

3. How does Plutus provide the relayers with rewards?

4. How does Plutus prevent Sybil behavior?

3.1 Algorand's Messages

Algorand has three type of messages which need to be propagated:

- Transaction - a message that describes a transaction in the system.
- Block proposal - a message that contains the block that was proposed by the block proposer.
- Byzantine agreement - a message that contains the vote of a specific committee member.

For simplicity, we focus in this thesis only on transaction messages. Developing techniques for the other messages is left for future work and discussed in chapter 7.

3.2 Message logging

Plutus's message logging process logs messages' propagation paths and comprises two steps. In the first step, *message processing*, users sign the message that they relayed. In the second step, *path verification*, users verify that the path they received is valid.

3.2.1 Message processing

Message processing is the fundamental procedure in Plutus as it provides the framework through which Plutus incentivizes users. Every user in the network is required to relay messages in order to receive rewards. Because peer-to-peer networks are distributed, there is no simple way to globally keep track of what messages each user sent and to whom. To address this problem, each user, upon receiving a message *m*, signs the message with their own private key, and then sends it to their peers. See algorithm 1 and figure 3-1.

```
procedure ProcessMessage(sk, neighbors, m)
for neighbor in neighbors do
    m' ← m + neighbor.pk
    SendMessage(neighbor, pk, $Signed_{sk}$(m'))
end
```

Algorithm 1: Psudocode for processing and gossiping an incoming message. The procedure receives the sender's secret key *sk*, the list of neighbors *neighbors*, and the message *m*. Then, for each one of the user's neighbors, the sender adds the neighbor's public key and sends the message signed by its own private key.

This allows every user to know a "path"[1] (or sequence of users) that each message went through before arriving to her/him, allowing the block proposer to run the lottery and to distribute rewards to the message's relayers.

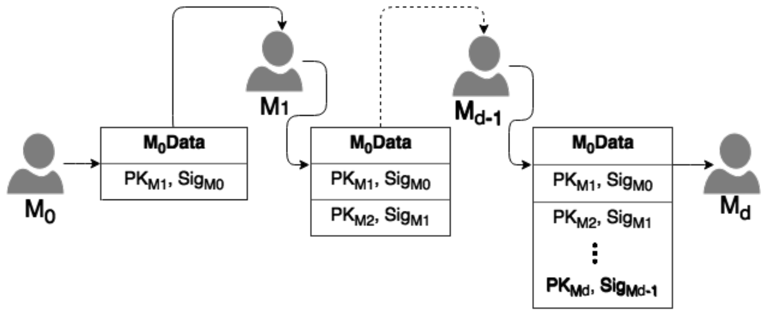


Figure 3-1: Message gossiping illustration. User M_0 issues a message, adds the destination public key PK_1 and signs the message. User M_1 repeats the process by adding the neighbors' PK, signing it and relaying it the next user.

Keeping track of a message's propagation path is challenging because it goes through an adversary's nodes who can try to manipulate it in multiple ways. For example, one possible attack on this procedure consists of deleting the signatures of previous relayers. In order to mitigate this attack and to make the path par-

[1]By the nature of P2P networks, every message arrives to every user from multiple peers. Each user in Plutus uses just the first copy of a messages it receives and ignores the rest. This may create an "unfair" situation where users with faster connections are rewarded more than users with slower ones. This situation is acceptable because Plutus favors the network performance over a single user's benefit.

tially [2] immutable to changes by an adversary, we modify the standard P2P *gossip* procedure. Instead of sending the same message to all of its peers, the relayer signs not only the message but also the message destination's public key. This creates a partially immutable path and prevents an adversary from creating arbitrary changes.

Another potential vector of attack by adversaries is adding an arbitrary number of signatures to the path. Plutus discourages this behavior by running the lottery proportionally to the relayer's stake in the network. With the proportional lottery, the user gets on expectation the same reward – regardless of the how the user's stake is distributed among its keys. (For detailed explanation, see section 3.3.1)

3.2.2 Path verification

Before proposing a block, the block proposer receives a set of different messages with their corresponding relayers' public keys and their signatures from the message issuer to block proposer. In order to keep only valid messages, the block proposer needs to verify the path of each message. To do so, the block proposer invokes the procedure *VerifyPath* (as shown in Algorithm 2). This procedure takes the path of signatures p, and verifies each signature against its signer's public key. In addition, the procedure checks that the last public key belongs to the block proposer.

[2] An attacker can still cut paths in the middle by creating a cycle. However, this cannot increase the expected reward of the attacker and therefore by our rationality assumption, a rational attacker has no incentive to do so.

```
procedure VerifyPath($p$)
$m, sig, pk \leftarrow p$;
while $p.next \neq NULL$ do
    if $VerifySig(m, sig, pk) \neq OK$ then
        return False
    end
    $p \leftarrow p.next$;
    $m, sig, pk \leftarrow p$;
end
return True
```

Algorithm 2: Psudocode for verifying signatures in a chain. Each signature on the Message path is verified against its public key.

3.3 Lottery

To provide relayers with rewards for their work, the block proposers run a lottery. In the lottery, Plutus randomly draws a subset of public keys from the set of all relayers' public keys that participated in propagating the messages that compose the block. In order to allow secure verification of the computation done by the block proposer, the random draw uses Verifiable Random Functions (VRF). The lottery is thus composed of two main components: *lottery selection* and *lottery verification*.

3.3.1 Lottery selection

The main procedure performing lottery selection, as invoked by the block proposer, is shown in Algorithm 3. The procedure takes a context *ctx*, which captures the current state of the ledger, the block proposer's secret and public keys *sk*, *pk*, the lookback parameter k, the current round *seed*, and total stake in the network W. The lottery selection process extracts the relayers' public keys from the k^{th} previous block set of transactions into *pks*. Then, to enable verification of the lottery, the process employs the VRF and computes its hash with the block proposer public key *pk* concatenated with the *seed*. To get a new random number for every key, the process hashes each public key form *pks* with the previous hash and checks if the user was selected for a reward by comparing the the normalized hash value to the user's weight in the network.

```
procedure LotterySelection($ctx, sk, pk, k, seed, W$)
// Getting the public keys from the set of transacctions
$pks \leftarrow getPublicKeysFromBlockByIndex(ctx.lastBlockIndex - k)$;
// Computing the public key VRF using the current seed
$\langle hash, \pi \rangle \leftarrow VRF_{SK}(pk||seed)$;
$winners \leftarrow \{\}$;
for $pk \in pks$ do
    $hash \leftarrow H(hash||pk)$;
    $pr \leftarrow \frac{hash}{2^{hashlen}-1}$ ;
    $w \leftarrow ctx.GetMoneyByPublicKey(pk)$ ;
    if $pr < \frac{w}{W}$ then
        $winners.add(pk)$ ;
    end
end
return winners, $\pi$
```

Algorithm 3: Psudocode for Lottery Selection. The procedure extracts the set of public keys from the *k*th previous block and computes the hash and proof using the VRF. Then, it checks which of the public keys were selected by normalizing the hash and comparing it to the fraction of the user's money in the network.

Mitigating Sybil attacks In order to discourage Sybil attacks, we set the probability of a user receiving the reward to be proportional to the user's stake in the network. This way, the user has no incentive to split their money to different keys and add additional signatures to the path. For example, let u be a user in the network with stake w_u. The expected profit once the user signature ends in the block is

$$Profit = Reward \cdot \frac{w_u}{W}$$

where w_u is the stake of the user and W is the total money in the system. Now, assume the user decides to create n new public keys. The user will need to split their stake among the new public keys $w_u = w_1 + w_2 + \ldots + w_n$. Thus, their expected profit stays the same.

$$Profit = \sum_{i=1}^{n} Reward \cdot \frac{w_i}{W} = Reward \cdot \frac{w_u}{W}$$

3.3.2 Lottery verification

After a new block is proposed together with the winning keys, the verifiers verify the block. To verify that the winning public keys were properly selected, they invoke the *LotteryVerification* procedure as shown in Algorithm 4. This procedure takes context ctx, the block proposer's pk together with its VRF proof π, the look-back parameter k, the list of winners public keys, the current round's $seed$, and total money in the system W. Then, the procedure verifies that the *LotterySelection* process was computed properly.

```
procedure LotteryVerification($ctx$, $pk_{bp}$, $k$, $winners$, $seed$,$\pi$, $W$)
// Getting the public keys from the set of winning transactions
$pks \leftarrow getPublicKeysFromBlockByIndex(ctx.ProposedBlock)$;
// Verifying the VRF proof and hash
if $\neg VerifyVRF_{pk_{bp}}(\pi, seed)$ then
    return False
end
$hash \leftarrow H(\pi)$ ;
$index \leftarrow 0$ ;
for $pk \in pks$ do
    $hash \leftarrow H(hash \| pk)$ ;
    if $pk = winners[index]$ then
        $w \leftarrow ctx.GetMoneyByPublicKey(winners[index])$ ;
        $pr \leftarrow \frac{hash}{2^{hashlen}-1}$ ;
        // Verifying the lottery selection
        if $(pr \geq \frac{w}{W})$ then
            return False ;
        end
        $index \leftarrow index + 1$;
    end
end
return True;
```

Algorithm 4: Psudocode for verifying the Lottery Selection. We first verify the VRF's proof and its hash. Given the first test passes, we check that the normalized hash was, indeed, lower than the fraction of the user's money in the network.

Chapter 4

Implementation

We implemented a prototype of Plutus on top of Algorand's [10] implementation using C++ 11. We kept Algorand's SHA-256 as the hash function and the VRF implementation as outlined in Goldberg *et al* [11]. In total, we added and modified nearly 800 lines of code (see table 4.1). In particular, we added the lottery component and refactored multiple parts of the code in order to integrate Plutus. The parts include the blockchain, message format, message handling, and the most significantly - the network.

Component	Lines of Code
Lottery	201
Signature handling	53
Network code refactoring	532
Total	786

Table 4.1: Approximate distribution of lines of code per Plutus's module

In Algorand's original implementation, there are two main modules that handle the message sending: a P2P node module and a network module. These two modules are separate and do not exchange data besides requests to send and to receive messages. Furthermore, the P2P node module is not "aware" of the node's network peers because these are handled by the network module. Since Plutus requires each message to include the public key of its destination, we refactored the

P2P module to have an API request that would allow the network to retrieve the information needed to add the target's public key and sign the message.

In addition, since Plutus adds the target's public key, we had to modify the *gossip* method to be a "multi-unicast". In particular, we modified the *gossip* method to be able to wrap each message sent with the destination's public key, sign it and send it to a specific peer.

Chapter 5

Evaluation

In order to understand Plutus's performance, we evaluate Plutus to answer the following questions -

1. How does Plutus perform in comparison to the vanilla Algorand system?

2. Does the lottery mechanism impose a large overhead on the block proposer and verifiers?

3. What effect does Plutus have on message size?

5.1 Method

To answer these questions, we measure Plutus' overhead and performance in two configurations. First, we evaluate Plutus' different sub-components using micro-benchmarks on a single machine on MOC cloud service [2] using a single 24 Intel(R) Xeon E3-12 v2 (Ivy Bridge, IBRS) 2.4Ghz cores and 128GB RAM machine. Second, we evaluate the performance and latency of Plutus in a distributed manner. We deploy our prototype of Plutus on Amazon's EC2 using 100 AWS m4.xlarge Virtual Machines(VMs), each of which has 8 cores, 16GB RAM, and up to 1Gbps network throughput. To measure the performance with a large number of users, we run multiple users, where each user is a process, on the same VM. By default, we run 50 users per VM and users propose 1MByte block. To simulate commodity links, we cap the bandwidth of each user to 20Mbps.

For the micro-benchmarks, we model our network and currency to have 10^5 users with 10^9 coins in supply. To simulate the world's wealth distribution [20], we assume that the money is distributed according to the power law distribution with $\alpha = 0.015$. (See figure 5-1). For the distributed case, we assign equal share of coins to each user; the equal distribution of coins maximizes the number of messages each user needs to process. The graphs in the distributed simulations show the time it takes to agree on a block, including the minimum, median, maximum, 25th, and 75th percentile times across all users. To measure the overhead of Plutus, we use Algorand's original results as a baseline.

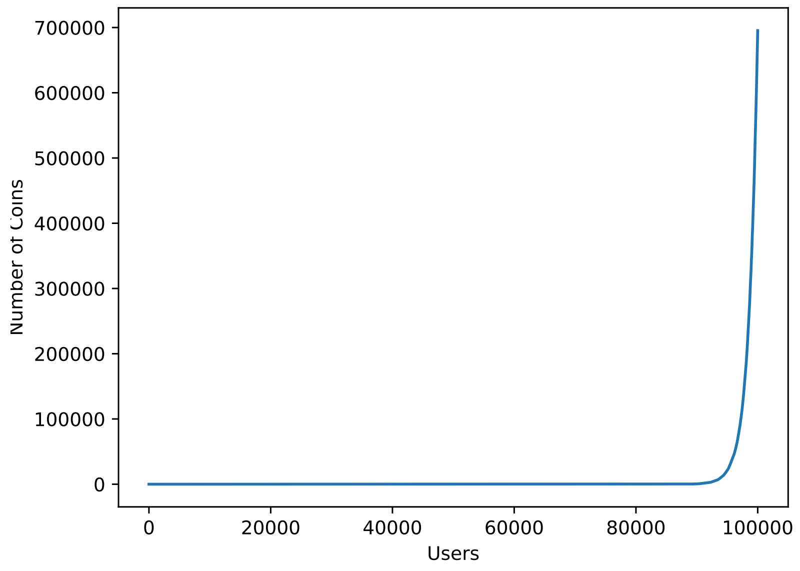
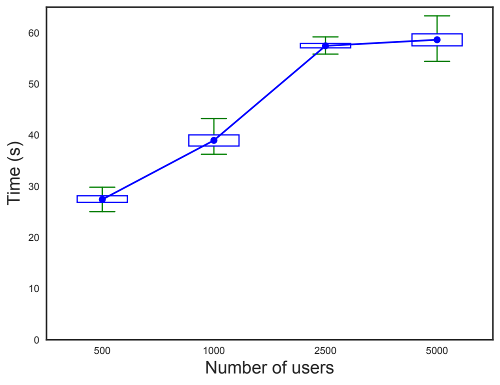


Figure 5-1: Money distribution in our simulations. We assume 10^5 users with 10^9 coins in supply, distributed among the users using the power law distribution with $\alpha = 0.015$ (To simulate world's wealth distribution [20]).

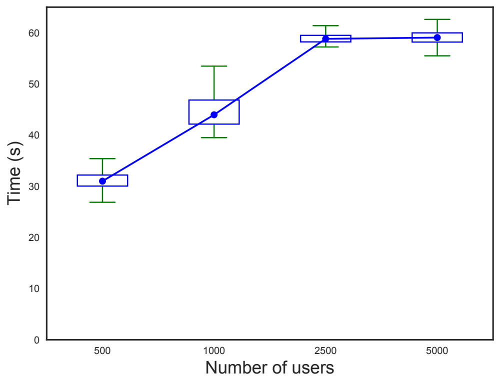
5.2 Algorand as a baseline

To answer our first question regarding Plutus's performance in comparison to the vanilla Algorand system, we compare our implementation of Plutus with Algorand vanilla's implementation. For accuracy of results, we match the original implementation to ours by removing one optimization—sending batched messages. This change results in higher latency compared to the results demonstrated in the paper originally describing Algorand. Our analysis shows that Plutus incurs approximately 7% increase in latency over Algorand's vanilla implementation.

From our first experiment, Figure 5-2 shows results with number of users varying from 500 to 5000 (by varying the number of VMs from 20 to 100). The results show that Plutus keeps the results of Algorand and incurs a small overhead in terms of latency.



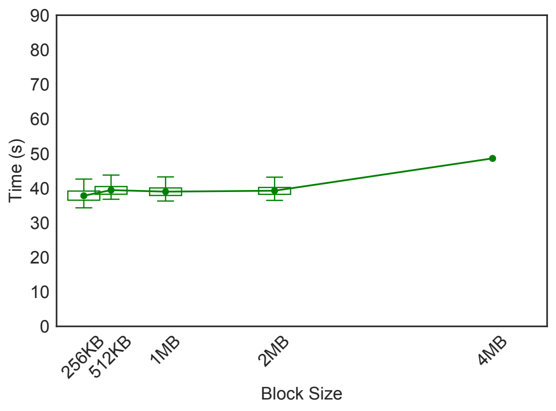
(a) Vanilla implementation



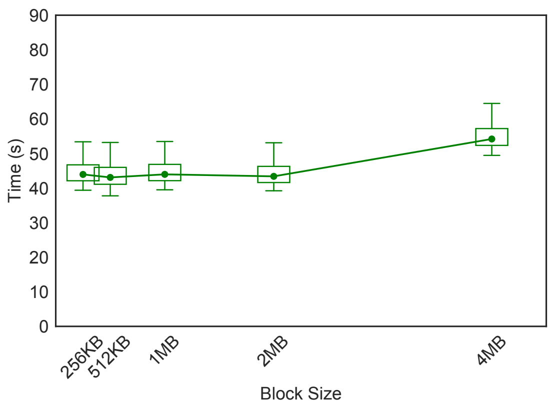
(b) Algorand with Plutus

Figure 5-2: Latency for one round of Algorand as a function of number of users

In a second experiment, we deployed 1,000 users on our VMs (50 per machine). Figure 5-3 shows results with varying block size, including a side by side comparison of the two implementations. The figures show that the overhead of Plutus is small and constant, unaffected by block size.



(a) Vanilla implementation



(b) Algorand with Plutus

Figure 5-3: Comparison between Algorand's vanilla implementation and Plutus's latency for one round of Algorand as a function of the block size

5.3 Message signing and path verification

Every relayer must sign and verify message paths. We measure the time it takes to sign the hash of the message paths with varying message sizes. Figure 5-4 shows the time it takes to sign the message as a function of the number of hops (i.e. the number of previous relayers) it passed before arriving at the current node. As expected, because we sign the hash of the message, the time of the signature is approximately constant and negligible.

Figure 5-5 shows the time it takes to verify every message path as a function of its length and the original message size. With a P2P network size of 10,000 nodes (as in the Bitcoin network, for example), we expect the number of hops to be $\log_8(10,000) = 4.4$ [1]. In this case, figure 5-5 shows that the time it takes is about $200\mu s$.

[1]In a graph with degree d, we expect the average path to contain $log_d(Nodes)$ hops [8]

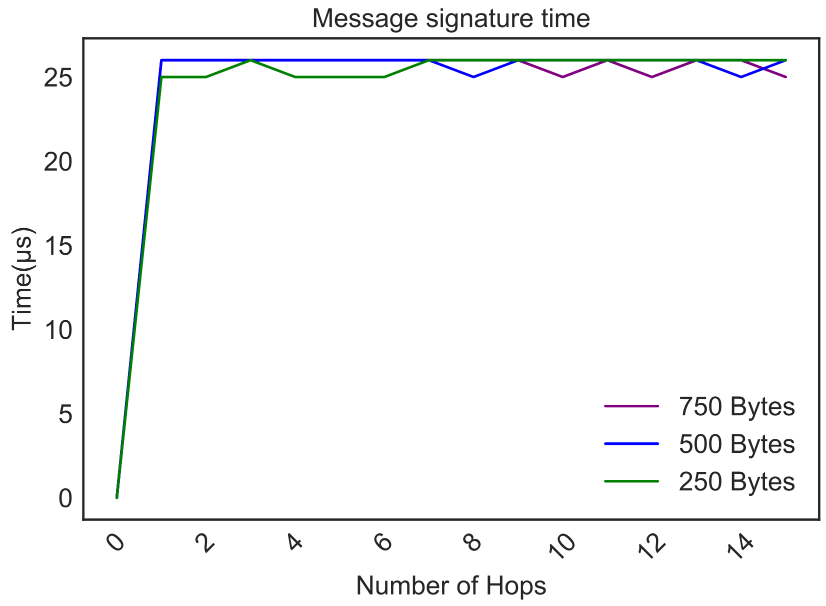


Figure 5-4: Message path signing time as a function of the number of hops. Since we sign the hash of the path, the time is relatively constant.

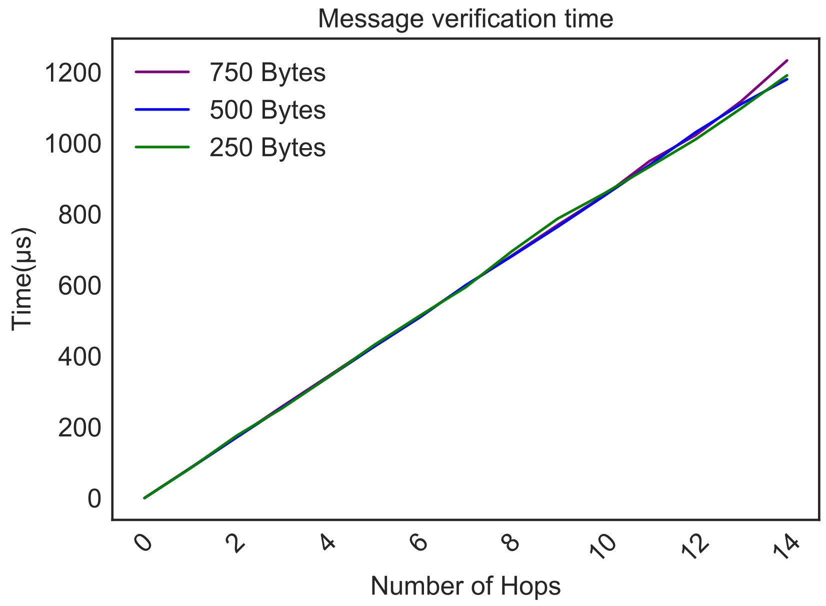


Figure 5-5: Message path verification time as a function of the number of hops. The three different lines show the time it takes for different message size as issued by the original sender. As shown, the message size does not impact the verification time.

5.4 Lottery running and verification

Plutus imposes one additional task on the block proposer: running the lottery and rewarding the winners. In order to measure the time it takes to run the lottery, we uniformly sample a number of keys from all the keys in the system and run the lottery among them. Figure 5-6 shows the amount of time it takes to run the lottery and to create the list of winning transaction as a function of the number of public keys. In the average block of size $1MB$ and average transaction size of $800Bytes$ (with a path comprised of 8 relayers), the expected number of transactions in a block is approximately 1200. With that many transactions, the lottery is expected to run among 4800 different keys. The figure shows that this takes an additional $10ms$ of computation time from the block proposer node. Even with large block size, of $4MB$ for example, the lottery runs in just $40ms$.

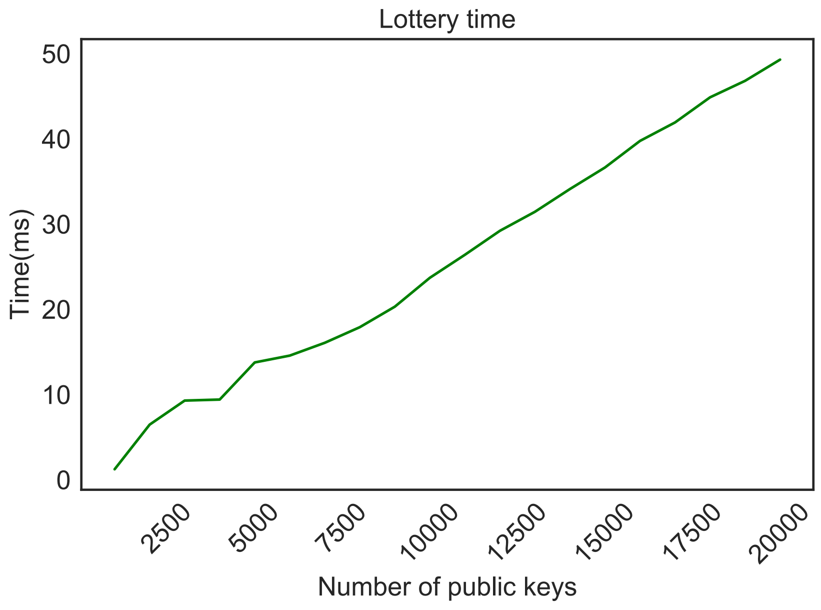


Figure 5-6: Lottery time as a function of the number of public keys. We sample a set of N public keys and select the winning keys with respect to their weight.

The committee members who verify the blocks in Algorand need to verify that the lottery was done properly by the block proposer. Figure 5-7 shows the time it takes to verify the block proposer computation as a function of the number of public keys. As shown, Plutus can verify the lottery in just a few milliseconds.

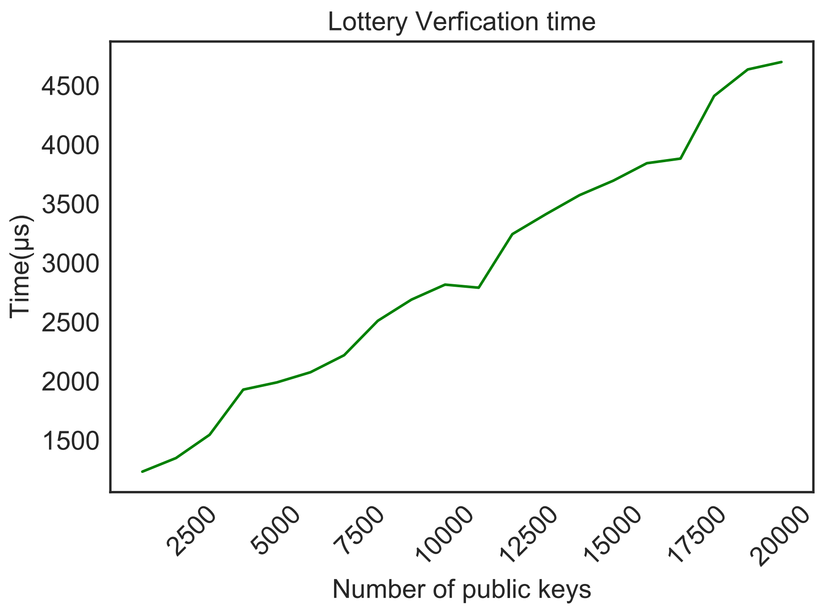


Figure 5-7: Lottery verification time as a function of the number of public keys. The verifier takes the set of winning keys and verifies that the lottery computation was done correctly.

5.5 Message Size

The main impact of Plutus on performance is that Plutus incurs a large overhead on message size. Because the relayer adds a public key and a signature to each message, the message size is expected to grow by *Signature size* + *Public Key size* = 64 + 32 = 96 *Bytes* at every hop. Figure 5-8 shows that this is indeed the case. Furthermore, we expect the number of hops to grow logarithmically with respect to the number of nodes in the network. For example, as mentioned above, in a network of 10,000 nodes, we expect to have about 4 hops. Therefore, each message is expected to have an additional 300 bytes.

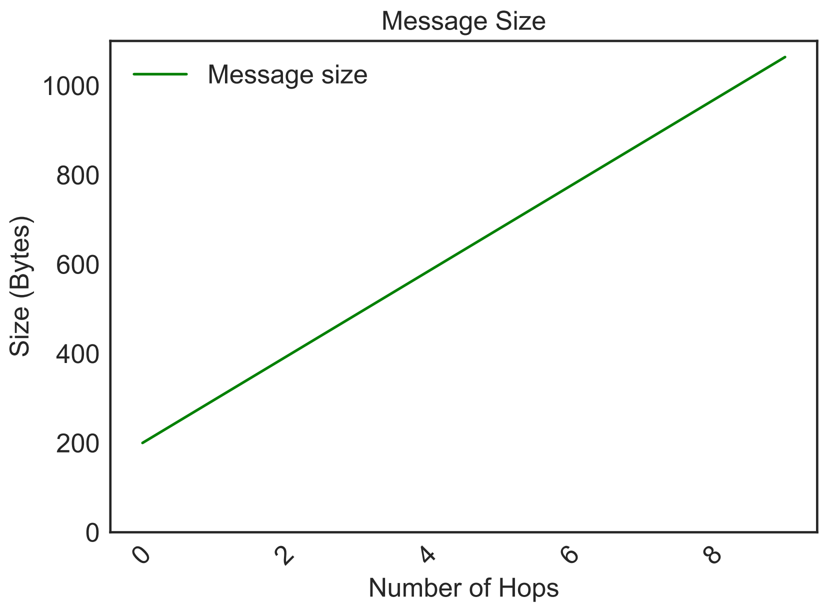


Figure 5-8: Message size as a function of hops. The initial message size is 200Bytes. At every hop, the user adds its public key and signature to the message which increases the size of the message by roughly 96 bytes.

Chapter 6

Game Theoretical Analysis

To study whether Plutus encourages users to relay messages, we use game theory and model the scheme as a static game to analyze the behavior of the nodes. We use the *Nash equilibrium* results of the game to show that for every transaction, each user receives it with probability $> \frac{1}{2}$.

The model of the game is described as follows.

Notations

Table 6.1: Notations

W	Total money in the system
f	The minimum fraction of money in the system that each user needs to have in order to be a user in the game
N	The number of users who have at least fW coins
$P = P_1, \ldots, P_N$	The set of users in the game
w_i	User's P_i stake in the system
R	The maximum reward paid per transaction
c	The cost for relating a single message

Simplifying assumptions

1. The connection graph is fixed.

2. The cost of relaying a single message for user P_i is c.

Users. The game has a set of N users P, where each user has sufficient stake w_i and the total stake in the system is W. There exist another user P_s, which might not belong to P, that issues a transaction t and sends it to User $P_t \in P$.

Strategies. The transaction t, along with its propagation path, is sent to a user P_i for the first time. P_i can choose its action $S_j \in \{Relay, Drop\}$ for each of its neighbors P_j.

Utilities. User P_i gets its utility by subtracting the cost of relaying c from the expected reward R. The expected reward is computed by the product of the probability of P_j being selected to be a block proposer, the probability of the P_i to win the lottery, the probability of P_j putting P_i's transaction in the proposed block, and the reward R.

$$u_i = \left\{ \begin{array}{ll} p_{i,j}\frac{w_i}{W}\frac{w_j}{W}R - c, & \text{for } 0 \leq i,j < N \text{ and } S_i = \text{Relay} \\ 0, & \text{for } S_i = \text{Drop} \end{array} \right\}$$

where $p_{i,j}$ is the probability that user P_j puts P_i's transaction in its proposed block, upon receiving.

Definition 1. An incentive scheme is *cooperative* if, at *Nash equilibrium*, each user receives a transaction t with probability $p > \frac{1}{2}$

Theorem 1. *Let the parameters R, c and f be such that $\frac{1}{2}f^2R - c > 0$. At Nash equilibrium, each user receives t with probability $> \frac{1}{2}$*

Proof. At Nash equilibrium, let A be the set of users who receive t with probability $> \frac{1}{2}$ and B the set of users who receive t with probability $\leq \frac{1}{2}$.
Assume by contradiction that at Nash equilibrium, at least one user receives t with probability $\leq \frac{1}{2}$, so B is non-empty. User P_t receives t with probability 1, so A is also non-empty.
Let P_A and P_B be two users in A and B that are connected to each other (such users exist because the graph is connected). P_A does not always relay t to P_B, otherwise P_B would also receive t with probability $\geq \frac{1}{2}$.

We will show that User P_i has an incentive to change its strategy to always $S = Relay$, therefore the game cannot be in equilibrium and we're done.

We've shown above that P_A relays to P_B with probability $p_A < 1$. Let us compute his additional utility from always propagating.

$$u_m = (1 - p_A) \cdot \left(p_{A,B} \cdot \frac{w_A}{W} \frac{w_B}{W} R - c\right)$$

$(1 - p_A)$ is clearly positive so it is enough to show that $\left(p_{A,B} \cdot \frac{w_A}{W} \frac{w_B}{W} R - c\right)$ is positive. $p_{A,B} > \frac{1}{2}$ because P_B receives the transaction t only from P_A for at least half the time, and $\frac{w_A}{W} \frac{w_B}{W} \geq f^2$ by definition.
Therefore,

$$u_m = (1 - p_A) \cdot \left(p_{A,B} \cdot \frac{w_A}{W} \frac{w_B}{W} R - c\right) \geq (1 - p_A)\left(\frac{1}{2} f^2 R - c\right) > 0$$

$\square$

Chapter 7

Future work

This thesis focused on designing an incentive mechanism for Algorand which encourages the propagation of transaction messages. Yet, there is more work to be done to improve Plutus as described below.

7.1 Performance

Plutus's main bottleneck is the overhead it imposes on the message size, stemming from the addition of the sender's public key and additional signature. In order to reduce the message size substantially, one can make a use of *Cryptographic aggregators* and *Key Mapping*.

7.1.1 Cryptographic aggregators

Boneh *et al* [5] introduced the notion of cryptographic aggregators. Cryptographic aggregators are cryptographic primitives that allow the aggregation of multiple signatures into one. Given n signatures, signed by n distinct users, it is possible to compress all of these signatures into one short signature. This single signature will be sufficient to convince the verifier that these n distinct user indeed signed the n distinct messages.

A survey by Malina *et al* [17] of several such constructions shows that efficient constructions, such as LOSSW by Lu *et al* [16], allow for sequential aggregation, reducing the size of the signatures to only $2kp$ where kp is the size of one element.

7.1.2 Key Mapping

In order to reduce the size of public keys in the messages, one can introduce a sequential numbering to the public keys on the blockchain to create a 1-1 mapping between a public key and its sequential number. Formally, $f(pk) \to i$ s.t. $i \in [1, N]$ where N is the number of keys in the system. That way, only the sequential number of the key and not the key itself can be store in a message. Assuming 64bit integers, introducing the sequential numbering will cause an improvement of 4X in size.

7.2 Additional Messages

In this thesis, we describe only how to encourage the propagation of transactions messages but these are not the only messages in Algorand. For completeness, there is also a need to encourage the propagation of Block proposal and Byzantine Agreement messages.

Chapter 8

Conclusion

This thesis introduces Plutus, an efficient, game theoretically proved, and Sybil resilient mechanism to measure work of nodes in P2P networks and to provide incentives in Algorand to encourage users to propagate messages. Plutus keeps track of messages' propagation history paths and provides relayers with incentives using a verifiable lottery. Plutus also has low overhead and doesn't affect the scalability of Algorand's consensus protocol. We implemented and evaluated Plutus on top of Algorand. Experimental results showed that Plutus increases Algorand's block confirmation time by only 7%.

Bibliography

[1] Cryptocoin price index and market cap - worldcoinindex. https://www.worldcoinindex.com/.

[2] Mass open cloud: An open cloud exchange public cloud. https://massopen.cloud/.

[3] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 95. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[4] Anonymous. *PIVX: Private Instant Verified Transactions*. https://pivx.org/what-is-pivx/white-papers/.

[5] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 416–432. Springer, 2003.

[6] Fabio C. Canesin, Yak Jun Xiang, Jeremy Lim, Ethan Fast, Joshua Lowenthal, lllwvlvwlll, Alan Fong, and Erik van den Brink. Neo white paper. http://docs.neo.org/en-us/.

[7] Evan Duffield and Daniel Diaz. *Dash: A PrivacyCentric CryptoCurrency*. https://dashpay.atlassian.net/wiki/spaces/DOC/pages/1146949/Site+map.

[8] Paul Erdos and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[9] Huibin Feng, Shunyi Zhang, Chao Liu, Junrong Yan, and Ming Zhang. P2p incentive model on evolutionary game theory. In *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM'08. 4th International Conference on*, pages 1–4. IEEE, 2008.

[10] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[11] Sharon Goldberg, Moni Naor, Dimitrios Papadopoulos, and Leonid Reyzin. Nsec5 from elliptic curves: Provably preventing dnssec zone enumeration with shorter responses. *IACR Cryptology ePrint Archive*, 2016:83, 2016.

[12] Trond Hønsi. Spacemint-a cryptocurrency based on proofs of space. Master's thesis, NTNU, 2017.

[13] Pan Hui, Kuang Xu, Victor OK Li, Jon Crowcroft, Vito Latora, and Pietro Lio. Selfishness, altruism and message spreading in mobile social networks. In *INFOCOM Workshops 2009, IEEE*, pages 1–6. IEEE, 2009.

[14] Abhiram Kothapalli, Andrew Miller, and Nikita Borisov. Smartcast: An incentive compatible consensus protocol using smart contracts. In *International Conference on Financial Cryptography and Data Security*, pages 536–552. Springer, 2017.

[15] Cuihong Li, Bin Yu, and Katia Sycara. An incentive mechanism for message relaying in unstructured peer-to-peer systems. *Electronic Commerce Research and Applications*, 8(6):315–326, 2009.

[16] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 465–485. Springer, 2006.

[17] L. Malina, J. Hajny, and V. Zeman. Trade-off between signature aggregation and batch verification. In *2013 36th International Conference on Telecommunications and Signal Processing (TSP)*, pages 57–61, July 2013.

[18] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 120–130. IEEE, 1999.

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. https://bitcoin.org/bitcoin.pdf.

[20] Credit Suisse. Global wealth report 2017. *Zurich: Crédit Suisse. https://publications. credit-suisse. com/tasks/render/file*, 2017.

[21] Bin Yu and Munindar P Singh. Incentive mechanisms for peer-to-peer systems. In *International Workshop on Agents and P2P Computing*, pages 77–88. Springer, 2003.