Massachusetts Institute of Technology
Media Lab's Digital Currency Initiative
Sloan School of Management

UTreeXO

Authors: Benjamin Celermajer, Nicholas Swindell (Northeastern), Will Lopez-Cordero (MIT)

Keywords: UTreeXO, Wallets

Blockchain Lab Program
2018-2019 Working Groups Cycle
Taught by N. Narula, G. Gensler, S. Johnson, and M. Casey
Member Company: Digital Garage (DG)
Project Group: UTreeXO Wallet

15.S68 BLOCKCHAIN LAB

UTREEXO

Executive Summary

Cryptographic accumulators have been explored for decades, but in Bitcoin's 10 year existence, no viable solution has been identified to apply accumulators to produce a representation of the Bitcoin current state and reduce the data requirements of nodes. The primary reason for this is the need for proofs to identify membership to the accumulator, bridge nodes to communicate with non accumulator aware transactions and updating proofs in the accumulator as the state changes. UTreeXO is a solution developed by Tadge Dryja of the MIT Media Lab's Digital Currency Initiative that has designed a solution that seemingly has met a lot of these requirements in an efficient way.

UTreeXO is a dynamic hash based solution to reducing the data requirements of running a fully validating UTXO blockchain node (e.g. Bitcoin). Through a novel approach to swaping deleted leaves or branches with new leaves and branches within Merkle Trees, UTreeXO manages to reduce a Bitcoin full node data requirements from 3GB to under 1KB.

If Bitcoin full nodes can be run in under 1KB it significantly improves the ability of users to run their own full node on mobile devices, rather than downloading SPV wallets or third party provider wallets. This has significant implications for the distribution and decentralization of UTXO blockchains, with the potential to attract thousands of customers that otherwise would not have owned and operated full nodes. Further, by owning and operating a full node, users will benefit from improved security and privacy when compared to SPV or third party wallets.

It is worth noting that UTreeXO is still a code in development. There are significant hurdles that are required to be overcome before it is operational and functioning to its potential. Some of these include the existence of multiple bridge nodes to protect the network, the existence of many UTreeXO aware nodes to fully benefit from storing proofs, increased peer review of the UTreeXO code and greater stability of the code (still many improvements that can be made).

Contents

Introduction

The Bitcoin network was founded in January 2009 by person(s) who operated under the pseudonym “Satoshi Nakamoto”. As an opensource network, the Bitcoin code is constantly iterated and upgraded to improve its operations and efficiency. Bitcoin is currently on version 0.18 and its Github has over 20,000 commits from over 600 contributors. Despite all this work, one problem that has persistently challenged Bitcoin is its ability to scale.

Bitcoin’s most commonly discussed scaling challenge is its low transaction throughput. One of the key limitations that prevents higher throughput is that every transaction needs to be approved and validated by every full node on the network. For Bitcoin, by design the transaction limit has remained low due to the network’s 1MB block sizes. However, rather than seeing this as restricting the number of transactions, many consider it a mechanism to regulate the amount of data being broadcast to the network.

Bitcoin has many powerful attributes that make it an attractive digital commodity, perhaps none more so than the promise of self-managed digital custodianship. The most secure way to self-custody Bitcoin is to download a Bitcoin full node which has the information of every account of every user in the ecosystem. This information is stored as a collection of all the Unspent Transaction Outputs (UTXOs) that have been created since January 2009. Today, this amounts to approximately 200GB of data that must be downloaded. Under current circumstances, this is only going grow as Bitcoin’s price increases and becomes more usable as a medium of exchange. Thankfully, network users are not required to store the whole 200GB, but rather only the current state of the network which only shows who owns what right now (currently approximately 4GB but also growing).

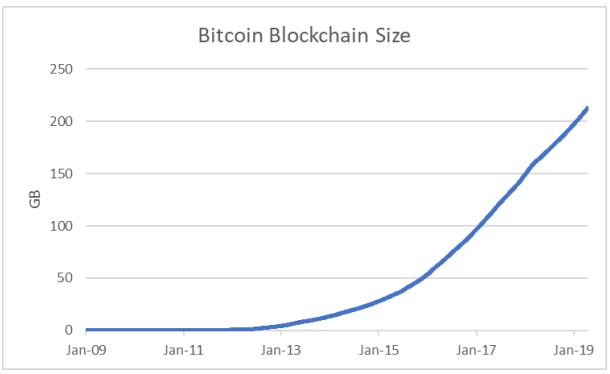


Figure 1 - Bitcoin blockchain size (blockchain.info)

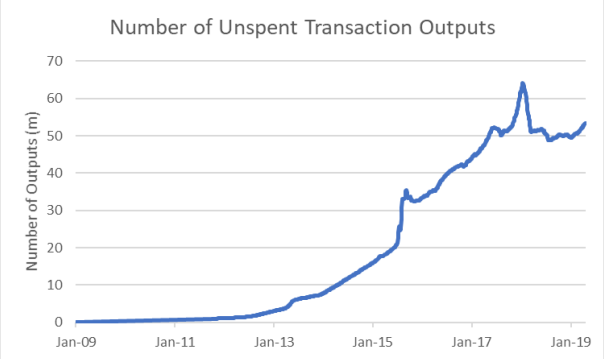


Figure 2 - Number of UTXOs in the Bitcoin current state (statoshi.info)

Whilst the size of the history is an issue, the much larger issue is the growing size of the current state. As the data storage requirements continue to grow, it inhibits users from running a full node on their mobile devices, PCs and other hardware. As the current state of the blockchain gets too big for users to store on their personal devices, it can place a strain on Bitcoin's (or any other blockchains) distribution of nodes. The consequence is higher centralization of validators and thus higher potential of collusion and network corruption (it is easier to collude between 10 Full nodes than 100).

UTreeXO plans to specifically address the issues plaguing the growing size of the current state. It does so through the use of a hash based dynamic accumulator that utilizes Merkle trees to create a representation of the UTXO set in under 1KB. With UTreeXO, users will not need to store the full set of UTXOs currently in the network, but rather hash(es) that represent the current state along with proofs of the UTXOs that relate to their wallets. For the Bitcoin network, the current 55m UTXO set is represented in a maximum of 27 Merkle roots (864B at 32B per hash).

Blockchain size solution review

Sharding

Sharding is a mechanism for partitioning databases and has been around since the 1990s. As the age of information was booming traditional databases were becoming tough to manage, costly to run and slow to query. Sharding was a method that separated large databases into smaller, faster and more manageable pieces called ‘data shards’. These data shards can then be distributed across more manageable and less expensive servers.

This theory has recently been applied to come blockchains with some success. To shard a blockchain there have been two primary approaches investigated:

- Sharding processing – the easier of the two solutions, each shard has its own transaction verifiers and block producers who still maintain the full current state of the blockchain. Once a shard produces a block, the information is collated by parallel subcommittees and shared with all the other shards. Whilst this potentially improves the rate of transactions of the blockchain, as all nodes are still required to hold the current state it does not solve data storage requirements.
- Sharding the current state – this more complex application requires shards of a complete state to be divided into several shards. In this application, each shard essentially acts as its own ‘mini-blockchain’. Transactions that occur between separate shards are required to undergo a process to ensure representation on both without risk of double spend. This implementation requires nodes to only store the current state of the shard they operate and thus does contribute towards reducing a node’s data storage requirements.

Child blockchains

As proposed by Ethereum founder Vitalik Buterin and Joseph Poon in their Plasma paper, this is a method where child blockchains (and even child blockchains of child blockchains) can be created from the main blockchain. Once rooted to their parent chain, a child chain would only need to store the transaction data relevant to them, ignoring information from every other child chain in the network and the parent chain it is a member of. The parent chain stores the state only needs to store the hashes of the child chain and not any of the transaction information.

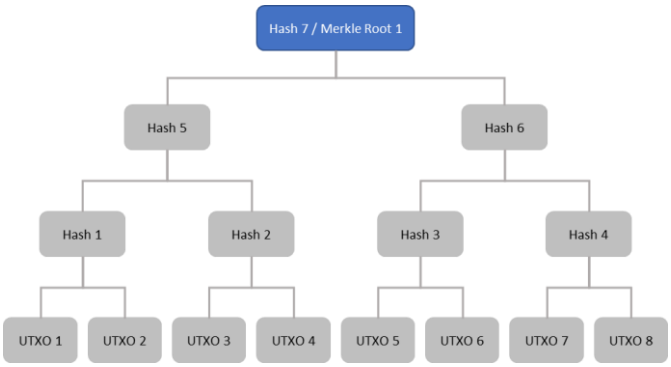
This significantly reduces the amount of data required to be stored on the main parent chain of the network.

Accumulators

Cryptographic accumulators were proposed in 1993 by Benaloh and de Mare (REFERENCE) as an alternative to digital signatures when designing secured distributed protocols. A cryptographic accumulator is an algorithm that collects a finite set of inputs and creates a single succinct output. Whilst accumulators are one-way functions in which no single input can be determined from the output, every input has an efficient witness set or inclusion proof that certifies membership in the accumulator. It follows that finding a witness for a value not in the accumulator set is computationally infeasible.

Hash based accumulators

The simplest accumulator is a Merkle Tree, which is a structure that allows for efficient and secure verification of information in a large data set. A Markle Tree takes a set of 2^n inputs that are hashed together to create a single output, the Merkle Root. Each pair of nodes in the tree are hashed together at each level until there is only one hash left, as demonstrated in figure 3.



3 - Merkle Tree

Dynamic Accumulators

More recently, Camenisch and Lysyanskaya[1] proposed the more complex and practical dynamic accumulator, where members can be added and deleted from the accumulator. This increases the potential applications as the accumulator set can be adjusted without requiring knowledge of the existing set of inputs.

RSA Accumulators

RSA accumulators consist of proof batching and aggregation techniques that can lead to improvements in the accumulator's proof size as well as reduce the required node communication requirements of stateless blockchains.

Through batching and aggregation of proofs, RSA accumulators reduce the overhead on verifiers of the network who can batch and verify *n* proofs faster than they can verify a single proof *n* times[2].

One problematic practical consideration for RSA accumulators is that proof updates cannot be aggregated. As proofs change, RSA accumulator nodes are required to update all proofs one at a time, meaning that the Bitcoin network with ~55m UTXOs, would require 55m RSA operations, or potentially worse if one operation is required per change in the accumulator.

UTreeXO – A hash-based solution to blockchain data storage

Merkle Tree's have long been applied in cryptography as a mechanism of representing a large data set as a single output hash. Since hash outputs are unique (in the vast majority of scenarios) to the set of inputs that create them, proof of membership to the data set can be provided with knowledge of the hash pairings that create the Merkle Root. The size of the output Merkle Root is dependent on the hashing algorithm but generally speaking is orders of magnitude smaller than the large data set of inputs.

UTreeXO is a UTXO hash-based accumulator that takes the current state of Bitcoin as the set of inputs and creates a Merkle Root(s) to represent the complete data set. Since the Bitcoin

[1] http://groups.csail.mit.edu/cis/pubs/lysyanskaya/cl02a.pdf
[2] https://eprint.iacr.org/2018/1188.pdf

current state is rarely exactly 2^n UTXOs, UTreeXO is not a constant size like the RSA accumulators. Rather, numerous Merkle Roots are required to represent the current state. However, due to the hashing algorithm and structure of Merkle Trees, the number of Merkle Roots required for any given set of inputs grows logarithmically at O(log(n)), inferring that the tree never becomes prohibitively large. The result is that UTreeXO stores has a 'forest' of trees where each individual tree is different in size and has 2^n number of leaves.

For Bitcoin, each 32B UTXO input is hashed with another UTXO to create a 32B output, this process continues until there is only a set of Merkle Roots left that represent the whole current state. Currently with ~55m UTXOs in the network, the maximum number of Merkle Roots required is 27. At 32B per Merkle Root, UTreeXO allows for the Bitcoin current state to be represented in 864B.

Cryptographic solution – A Dynamic Merkle Tree

Merkle Trees have not been considered an appropriate mechanism for accumulating dynamic data sets as there have been no solutions to date that allow for the substitution of deleted inputs with added outputs. The implications of this are that all deleted inputs leave holes in the Merkle Tree and new inputs are just added to the end of the Merkle Tree. Over time the tree grows into a large, non-efficient tree where each addition or deletion results in modification of internal tree nodes which corrupt Merkle paths. This results in the production of inefficient witness sets that make it difficult to prove inclusion.

One of the novel and innovative cryptographic solutions that has been developed by Tadge Dryja for UTreeXO is the ability to replace deleted Merkle leaves and branches with smaller Merkle Trees. By doing so, as new transactions are created, added onto the Merkle tree and swapped in place for deleted Merkle branches, the size of the tree remains relatively small.

Merkle Tree proofs

Merkle Tree proofs allow for verification that a piece of data is included in the Merkle Tree without revealing any of the other pieces of data that comprise the tree (i.e. a proof of inclusion). This is one of the critical aspects that contribute to the efficiency of accumulators. Without the ability to demonstrate inclusion, users would have to revert back to storing the full

set of data to verify inclusion. Figure 4 demonstrates how a proof is generated, here to prove the existence of UTXO 3, one must have information about UTXO 4, Hash 1 and Hash 6. The hash of UTXO 3 with UTXO 4 create Hash 2, the hash of Hash 2 with Hash 1 create Hash 5 and the hash of Hash 5 with Hash 6 create the unique Merkle Root, thus demonstrating that UTCO 3 must be in the dataset. Since all verifiers are required to store the Merkle it is simple for them to match the proofs root with their Merkle root.

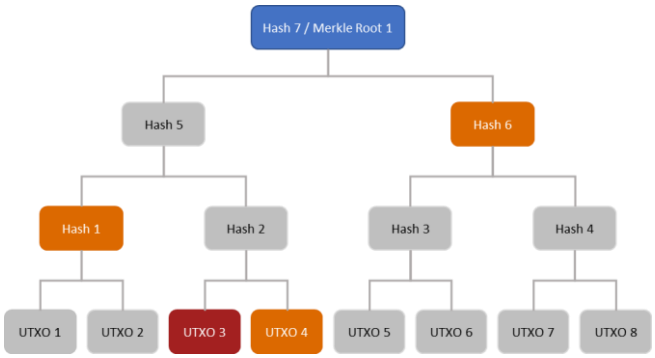


Figure 4 - Proof of inclusion

Adding leaves

As UTXOs are created in the blockchain they enter the current UTXO set and are added to the end of the current Merkle tree. As the number of UTXOs in the network increases so too does the size of the Merkle tree and the number of possible Merkle roots required. However, the relation between number of UTXOs and size is not linear but rather represented by a log function where O(log(n)) represents the maximum number of Merkle roots required for the set of n UTXOs.

When a new leaf is added to the Merkle tree, if the tree it is added to had an odd number of leaves the accumulator will create a hash of the odd leaf with the new leaf and continue hashing up the tree until it creates an odd number of hashes at its horizontal level (Figure 5).

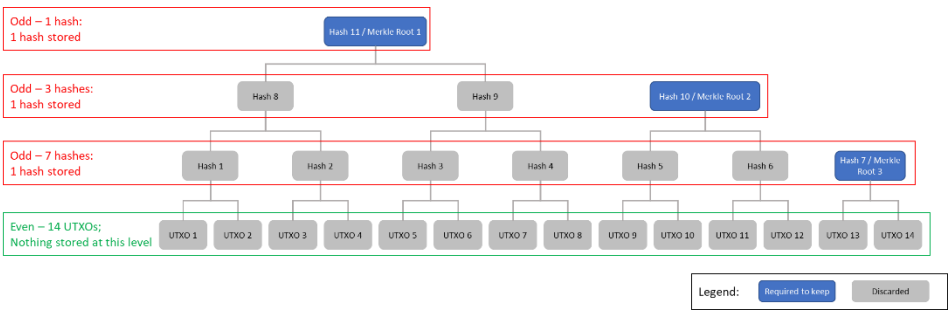


Figure 5 - Hashing of new leaves

Deleting leaves

As UTXOs are spent, they are removed from the data set and thus the Merkle Tree. This leaves holes in the Merkle Tree and can void proofs that previously existed (demonstrated in Figure 8). As mentioned previously, this has previously hindered Merkle Tree adoption in dynamic accumulators as it leaves imperfect tree structures, thus making proofs of inclusion inefficient.

However, UTreeXO has a novel multi algorithm solution where deleted tree leaves and nodes can be replaced to ensure all trees are always complete. The three algorithmic operations that enable this function are ‘twin’, ‘swap’ and ‘root’.

- Twin – If deleted siblings are right next to each other, they can be skipped, and just their parent hash can be deleted.

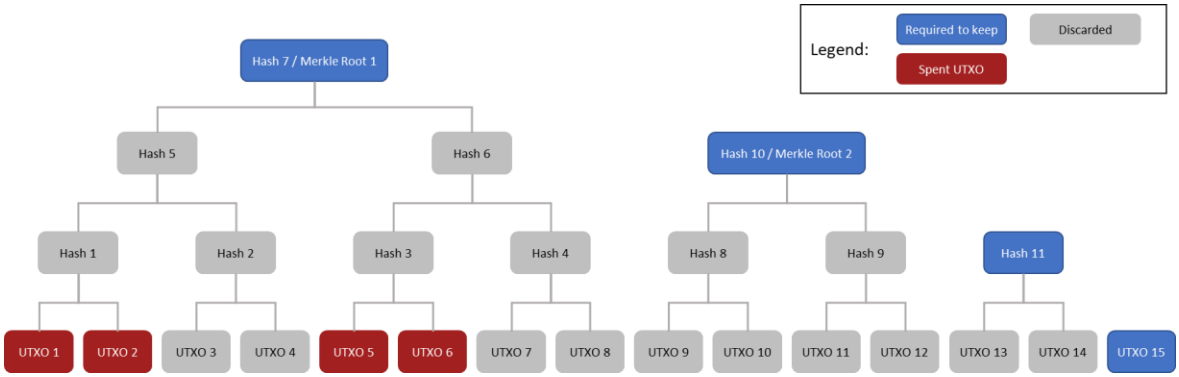


Figure 6 - Twin operation

- Swap – If deleted nodes at the same height are not next to each other in the tree, they can be moved so that they are. This allows for the newly paired sibling nodes to be hashed and have their parent hash deleted. By doing so, rather than deleting two ‘sub trees’ at a given height, *x*, you can delete one ‘sub tree’ at height $x+1$. This functionality is only enabled because UTreeXO participants can identify the location of deletions from the information from their unique proof of inclusion.

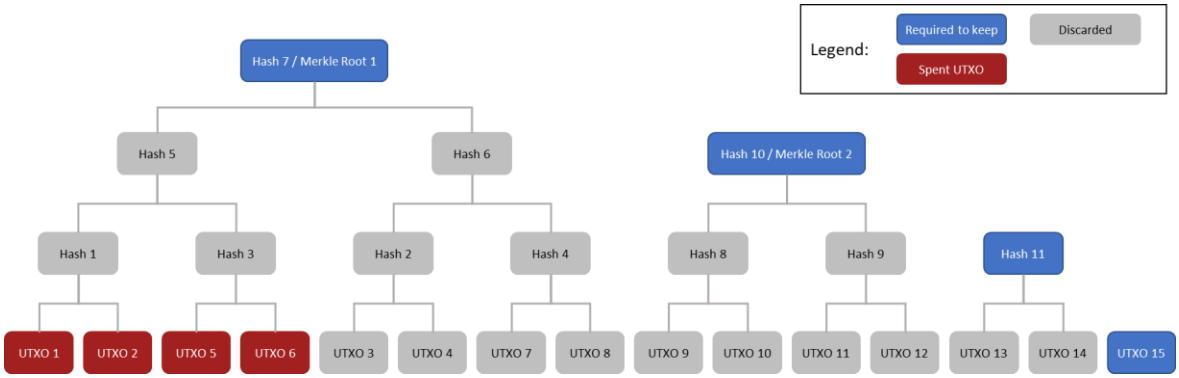


Figure 7 - Swap operation

- Root – This operation identifies the height of a deleted ‘sub tree’ and replaces it with a complete tree within the accumulator that is the same height.

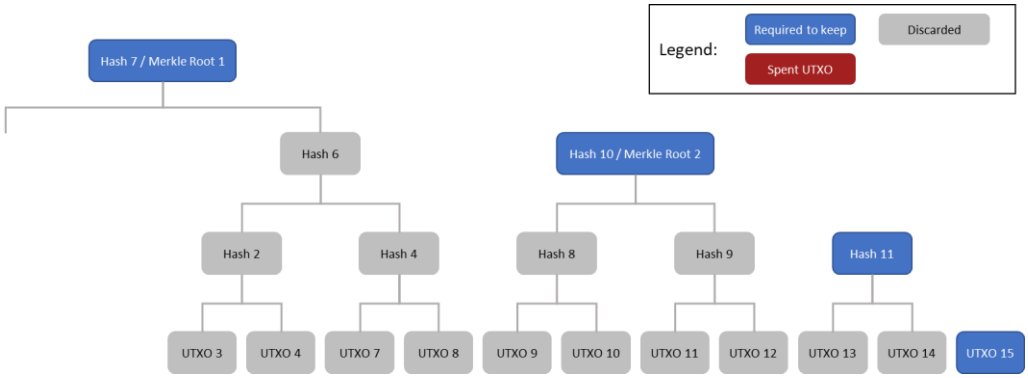


Figure 8 - Leaf / root deletion

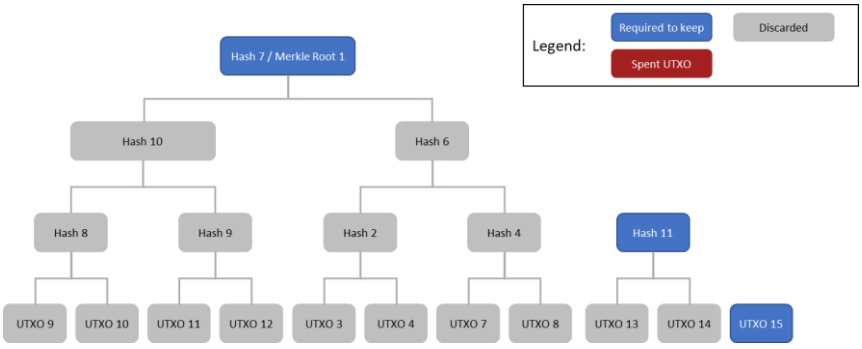


Figure 9 - Root operation

Operationalizing UTreeXO

UTreeXO has been developed and designed as a data scaling solution for any UTXO blockchain, specifically with Bitcoin in mind. Due to the distributed nature of Bitcoin's governance and high security standards, it has a highly conservative and slow moving roadmap where any proposed upgrades to Bitcoin Core need to undergo a long process of peer review before achieving a 100% majority approval or risk a hard fork.

One of the design features of UTreeXO is that it can exist in parallel with Bitcoin Core and does not require any change to the Bitcoin Core code base, nor does it put any additional information in the blockchain (i.e. proofs are not stored in blocks).

However, there are varying ways of implementation that would be more effective than launching a parallel network in isolation:

- A soft fork would lead to the most optimal implementation of UTreeXO as it would require all participants to upgrade their nodes / clients in order to recognize and store proofs. Since the value of the network to users increases with the number or nodes that store the UTreeXO Merkle Roots and proofs this is the most optimal implementation.
- A peer-to-peer update could also be committed which allows users to manually select to run a UTreeXO aware node in the configuration settings. Similarly to the most simple implementation where no change is made to the Bitcoin Core code, this would result in a separate UTreeXO network that would operate side by side with Bitcoin.

UTreeXO Architecture

Provided the aforementioned difficulty of gaining traction for a soft fork of Bitcoin, it is most likely that initially UTreeXO operates as a parallel network that participants can opt into. This creates a communication issue between UTreeXO aware nodes and Bitcoin nodes.

UTreeXO aware nodes

A UTreeXO aware node can run every single operation that a Bitcoin Core node runs. The key difference is that rather than being required to store the full state of UTXOs, nodes are only

required to store the Merkle Root set that represents the network as well as their own UTXOs and the related proofs to demonstrate that they exist.

UTreeXO aware nodes are still fully validating nodes and are required to verify all transactions independently before adding or deleting UTXOs to the current state and creating a new set of Merkle roots.

For UTreeXO aware nodes to transact, they are also required to send all information that a current Bitcoin node would communicate, they primary difference is that also attach the proof of UTXO inclusion. As Bitcoin nodes become aware of these transactions, they just ignore the proof and process the transaction like any other. The primary issue of running a parallel network is UTreeXO aware nodes understanding information when Bitcoin nodes transact. Since Bitcoin nodes do not attach proofs UTreeXO aware nodes do not know which UTXO has been spent and cannot conduct the twin, root, swap operation and create a new Merkle Root.

Bridge node

To resolve this a ‘bridge node’ is required that stores the full current state of UTXOs, the full UTreeXO Merkle Tree and the Merkle Root but unlike UTreeXO aware nodes, the bridge node does not delete the data set or any proofs once the Merkle Root is created. As the bridge node maintains the full state, once Bitcoin Network transactions are received it can attach the required proof and propagate it to the UTreeXO network, thus enabling UTreeXO aware nodes to update their states.

Since the Bridge node is required to maintain the full UTXO set and the complete Merkle Tree of proofs there is an unavoidable data storage overhead. Currently this is approximately 15-20% of a Bitcoin full node.

Whilst UTreeXO could operate with one bridge node, the system would have vulnerabilities in a scenario where a Bridge node stopped communicating correct information to UTreeXO aware nodes or went offline and stopped communicating all together. This would render the UTreeXO network useless as any transaction in the Bitcoin Network would not have the associated proofs required by UTreeXO aware nodes and thus they would not be able to conduct the required operations to create the Merkle Root.

As such, to secure the network it is recommended that numerous bridge nodes operate globally.

Having said this, the bridge node cannot communicate bad information to UTreeXO aware nodes since nodes are still fully validating.

Architecture

UTreeXO can be implemented in parallel to the Bitcoin Network with the existence of a (few) bridge node to attach the required proofs for UTreeXO aware nodes to operate.

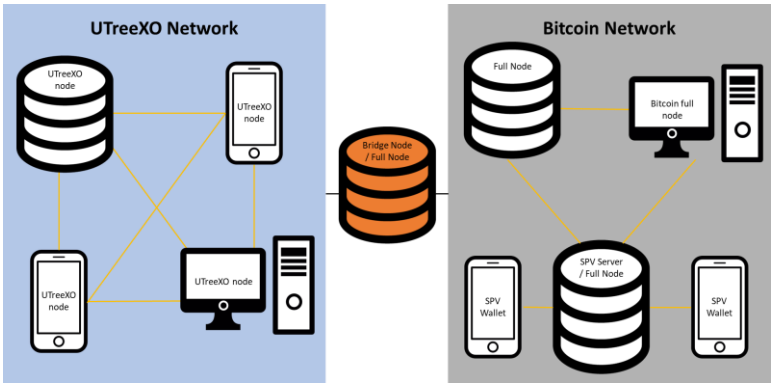


Figure 10 - UTreeXO parallel implementation with Bitcoin

UTreeXO Advantages

UTreeXO's greatest contribution to Bitcoin is the ability to allow anyone to run a fully validating Bitcoin node in under a kilobyte, rather than three gigabytes. By doing so, it enables smartphones, old computers and other small mobile devices to do what is currently limited to high powered computers and servers.

There are two generic categories of beneficiaries from UTreeXO:

1. **Individual Bitcoin users** – currently the most popular methods to mobile storage solutions are third party providers or SPV wallets. Both of these come with significant security and privacy concerns for users.

   Privacy – Third party providers each have their own security policies which vary in effectiveness and users largely unknowingly adopt these security procedures without much due diligence. As such there are numerous instances of hacks and loss of user funds from third party custody. Since SPV wallets are not validating, in special

circumstances like a consensus change, the SPV wallet can be tricked into thinking that non-upgraded nodes had the correct longest chain and thus validate through them.

Security – Since both third party providers and SPV wallet users provide information of the UTXOs they own, whilst an identity cannot be explicitly determined, crypto forensics can lead to identification of owners and value of ownership.

2. **Bitcoin Network** – since UTreeXO makes it more feasible for individuals to run fully validating nodes it vastly increases distribution of nodes and thus network security. The more validating nodes on the network the more secure and harder to corrupt the Bitcoin Network becomes.

UTreeXO Disadvantages

To obtain the current state of the UTreeXO network, users are required to download the full history of UTXOs and all of the associated proofs. Whilst all of this can be thrown away after it has been witnessed, there is a significant amount of data required to be downloaded to obtain the current state. To complete this function for Bitcoin it currently requires nodes to download ~240GB, for UTreeXO, with the additional proof information that needs to be generated, this increases to ~270GB. As Bitcoin's blockchain continues to grow, the 15-20% overhead will also continue to grow and require significant time for new users to obtain the current state.

Another disadvantage of UTreeXO is that is requires large network effects to operate most efficiently. If the majority of users are not UTreeXO aware nodes, then UTreeXO users will receive many transactions without proofs. As such, a bridge node is required to attach proofs to transactions sent without proofs (i.e. Bitcoin Network transactions). As the network continues to grow, bridge nodes have the potential to become so large that they require servers to run and can potentially become centralized which increases the vulnerability of the system (for 1,000 bridge nodes to simultaneously fail is less likely than for 5 to fail).

UTreeXO code repository current state

The MIT Digital Currency Initiative is home to the UTreeXO repository[3]. The two structs, Pollard and forest, are the main components of the code. Forest is for bridge nodes and contains the entire Merkle Tree / UTreeXO structure along with all of its data and can produce proofs. Pollard on the other hand can hold and produce partial accumulators and verify proofs. The structure is defined the way it is because UTreeXO nodes are only require to store the Merkle root output from the accumulator which the Pollard struct efficiently produces. Bridge nodes however are required to store the full Merkle tree structure and since Pollard structs are not as efficient as forest structs for holding the entire Merkle tree / UTreeXO structure, UTreeXO utilized both structs in its code. However, it is still being explored if constructing a the bridge node can be done in a more efficient way with Pollard, thus reducing the code down to only one struct.

The code is still in development by Tadge Dryja but is publicly available on the github. The code is written in golang, or go, an open source language developed at google and designed for modern programming that takes advantages of machines with multiple cores.

There are many optimizations that can be added. For instance, an optimal caching strategy exists for a UTreeXO Initial Block Download, or IBD. This is due to the fact that the entire txo insertion and removal schedules are known ahead of time. This is comparable to trying to plan an optimal coat hanging strategy while knowing the arrival and leave time of everyone at an event ahead of time.

At the time of this writing, 34 commits have been made to the utreexo repository and 2 contributors are listed. The code is still in its early stages and as an open source contribution to UTXO blockchains, ideally as awareness increases we start to see more peer review and contributions which would contribute to the code's strength and implement some of the previously mentioned optimizations.

[3] https://github.com/mit-dci/utreexo

Creating a wallet

When investigating the current UTreeXO codebase to scope the project and understand what was needed to launch a UTreeXO mobile application we identified four key functional pieces that needed to be developed:

1. A messaging client for bridge nodes to communicate with UTreeXO aware nodes to facilitate the transfer of proofs.
2. UTreeXO signature verification that allows verification of the right to spend UTXO.
3. A cyrptographic key management solution.
4. Mobile application software (i.e. UI/UX).

While the network is bootstrapped to UTreeXO, initially at least a large majority of transaction will be generated and propagated from the existing Bitcoin network. This means that no proof data will be attached to the transactions. As such, UTreeXO aware nodes need to look to bridge nodes to receive the required proofs information for each UTXO. This is the purpose of the messaging client. For this we decided that JSON would be a suitable lightweight data interchange format to complete this task. The steps from Bitcoin non-UTreeXO aware node to UTreeXO aware node are as follows:

- The Bitcoin non-UTreeXO aware node propagates the transaction data that includes the relevant UTXO data but no proof data.
- A bridge node recognizes this and receives the UTXO data and matches it to the UTreeXO Merkle Tree and attaches the relevant proofs.
- The bridge node then communicates this information to the UTreeXO aware nodes so that they can recognize the transaction has occurred.
- When the new block comes out, UTreeXO aware nodes and bridge nodes update the Merkle tree so that old UTXOs are deleted, new ones added, a new root added.
- UTreeXO aware nodes can then discards the leaves and are only required to keep the Merkle root and any proofs associated with their UTXOs.

For signature verification, we decided to use the uspv folder from the lit repository. It is designed for SPV functionality, and although it is not ready for production yet, we wanted to

keep as much code in golang, or go, as possible, as that is what the utreexo library is written in.

We choose BlueWallet (github.com/BlueWallet/BlueWallet.git) to develop our mobile application and key management solution. At first, we wanted to develop a new wallet and front-end using go, however, we decided that it was simpler for our purposes to find an up to date wallet built with React Native and build UTreeXO functionality into it. The required changes from the existing code centred around what the wallet needs to store; SPV wallets are only required to store block headers where-as UTreeXO wallets are required to hold the accumulator hashes and their personal UTXO proofs.

Code contributions

Currently to run the UTreeXO initial block download there are several manual steps that need to be taken, each taking up to several hours. We have written a script (Figure 11) in order to streamline the process so that there are no time delays between downloading Bitcoin TXO information and UTreeXO executable programs. The script runs an initial block download using a utreexo accumulator. The file requires TXO data from testnet or mainnet, and will install go if it is not already on your machine. The simulation code is in the cmd folder of the UTreeXO (github.com/mit-dci/utreexo.git) library.

```
print_options() {
        echo "--genproofs or --genhist"
        echo "if go is installed ignore OPTIONS: --yes if need to download testnet t:
        echo "OPTIONS:"
        echo -e "  --linux install linux version"
        echo -e "  --win install windows version"
        echo -e "  --mac install macOS version"
}

if [ "$1" == "--help" ]; then
        print_options
        exit 0
fi

VAR=$1

run_ibdsim() {
    go get "github.com/mit-dci/utreexo/cmd/ibdsim"
    cd "$HOME/go/src/github.com/mit-dci/utreexo/cmd/ibdsim"
    go build
    if [ "$VAR" == "--genproofs" ]; then
        echo "hi"
        ./ibdsim --ttlfn=$HOME/ttl.testnet3.txos --genproofs=true
    fi
    if [ "$VAR" == "--genhist" ]; then
        ./ibdsim --ttlfn=$HOME/ttl.testnet3.txos --genhist=true
    fi
    exit 0
}
```

Figure 11 - UTreeXO download script

The messaging client code proved to be too much to write during the semester, but we were able to work with Tadge Dryja to scope the piece and communicate this to our member company Digital Garage who are developing the code now. The key steps required for the messaging client are:

- Code blockproofs currently exists to attach proofs to transactions that are included in blocks[4]
- What is not done is attaching proofs to transactions as they are propogated across the network and into the mempool. To do this would require a similar code to blockproof (i.e. list of target, list of hashes) but needs to be constructed differently. This can be done by sending all proofs for all new transactions (easier way but requires more space)

Or the more efficient way is for UTreeXO nodes to communication as follows:

- Sender sends transaction information and the location of the UTXO in the network
- Other node on the network responds with a request for only the proof information they are missing (e.g. I need a proof to height x). Because of the merkle tree structure of UTreeXO it is likely that the receiver of the information will have proof information to a certain height at which the new proof will merge.
- Sender sends proofs up to the requested height only (rather than all proofs to the merkle root).

In order to implement uspv into the wallet we used ‘go bind’ to generate .jar and .aar bindings for the library in order to import it to android studio. This allowed us to develop a basic UTreeXO wallet using BlueWallet, which can be seen in Figure 12.

[4] https://github.com/mit-dci/utreexo/blob/52d341315fd77a2fa97250660db32541a6d412d7/utreexo/blockproof.go

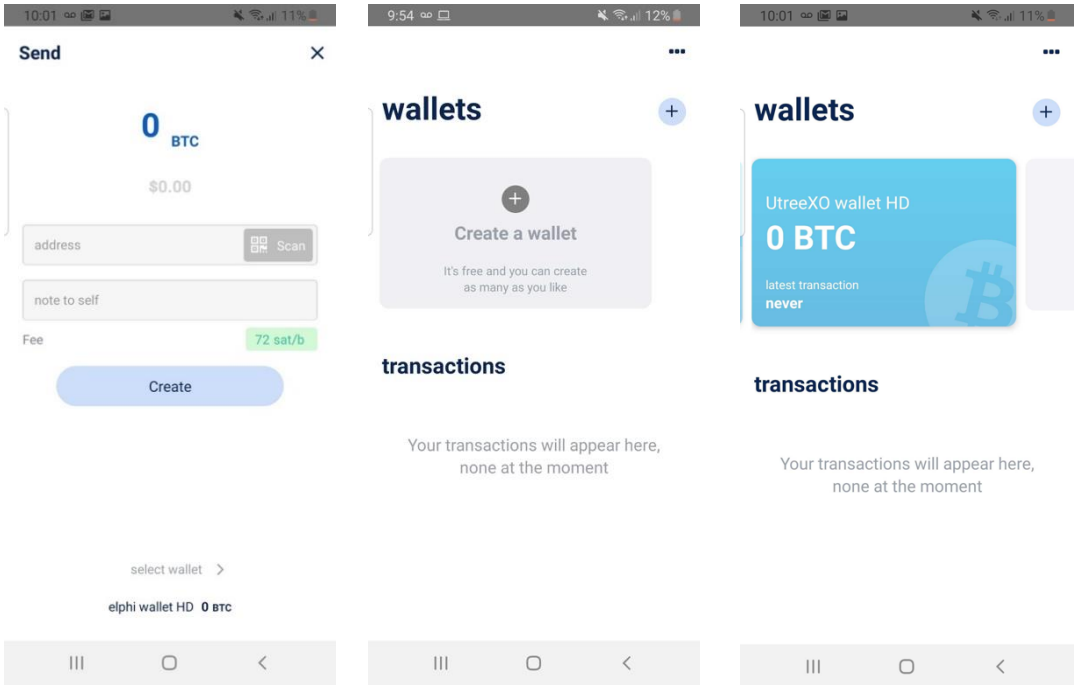


Figure 12 - UTreeXO wallet skins

Conclusion

UTreeXO applies a novel cryptographic solution to deleting and swapping Merkle tree leaves that for the first time allow the Merkle tree structure to be used as a dynamic accumulator. This can be implemented across UTXO public blockchains to reduce the current state into a representation consisting of series of hashes where membership to the accumulator can be identified through proofs of inclusion.

By doing so, the requirements to run a full node are significantly reduced since device data requirements are reduced from 3GB to under 1KB. This allows even the smallest of mobile devices and computers to feasibly run a full node and benefit from the security and privacy benefits of the Bitcoin network.

UTreeXO can be launched without any required change to the Bitcoin Core code which significantly increases the feasibility of the network. However, it considered that a better way to implement it, without a Bitcoin soft fork, would be with a peer-to-peer update, making UTreeXO a custom configuration setting when setting up a Bitcoin node.

The key concerns that UTreeXO needs to overcome to ultimately provide an efficient network for participants to benefit form are:

- The existence of a large number of distributed bridge nodes which have an increased data storage overhead requirement of ~15-20% above a Bitcoin full node due to the existence of proof data.
- Achieving a mass of users on UTreeXO so that the network can benefit from the efficiencies of sending and receiving proof data and verifying Merkle roots to build consensus on the current state.

UTreeXO is an exciting proposition for the UTXO public blockchains that have faced the challenge of centralization of nodes due to the increasing data storage requirements of running a full node and the impracticality of running these on mobile devices and computers as it grows.