2018 IEEE Confs on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics

IPFS for reduction of chain size in Ethereum

Robert Norvill
Sedan Group, SnT
University of Luxembourg
robert.norvill@uni.lu

Beltran Borja Fiz Pontiveros
Sedan Group, SnT
University of Luxembourg
beltran.fiz@uni.lu

Radu State
Sedan Group, SnT
University of Luxembourg
radu.state@uni.lu

Andrea Cullen
EECS, Engineering & Informatics
University of Bradford
a.j.cullen@bradford.ac.uk

***Abstract*—In this paper we propose a system that moves the bytecode of an Ethereum contract creation transaction off-chain. As blockchains are append-only we present a way to help reduce the chain size and growth for Ethereum. Contract creation transaction data is replaced with hashes which identify a file in InterPlanetary File System (IPFS). Doing so reduces the size of data stored in such transactions by 93.86% in our dataset. The proposed system retains the assurance provided by blockchain and reduces network traffic under certain conditions.**

***Index Terms*—Ethereum, Smart Contracts, compression, off-chain, IPFS**

1. Introduction

Ethereum is one of the world's foremost blockchain systems; it provides a global, public blockchain similar to Bitcoin. It is set apart by its heavy use of smart contracts, which are programs stored on the blockchain that allow for more complex behaviour than the simple recording of transactions. Thanks to the Turing completeness of the Ethereum Virtual Machine, smart contracts are capable of modelling legal agreements like domain name ownership or creating a secure system for voting. In Ethereum, they are compiled to bytecode from a source code language, most commonly Solidity, and stored on the blockchain as part of the transaction which adds them to the chain. This is known as a contract creation transaction (CCTX). This means that all contracts are stored permanently on the chain including old and unused contracts.

Blockchains are append-only by nature, data cannot be removed or changed without majority consensus. This is called a hardfork and is generally considered to be highly undesirable. As a consequence, full chains can become too large for average users to verify or store easily. Therefore ways to reduce the amount of data stored on the chain while retaining the assurances of a blockchain system are highly desirable.

IPFS stands for InterPlanetary File System, it is a program for distributed storage of files across a peer-to-peer network. IPFS stores files by splitting them into blocks and recording which blocks are belong to which file. Each file is identified by its hash, the user can check they have received the correct blocks by generating the hash of the file formed by the blocks and checking it against the hash they requested. Unrequested or unused files can be automatically removed from each participant's storage over time by IPFS's built-in garbage collector. Garbage collection handles local IPFS storage, however IPFS is designed such that files cannot be explicitly removed from the system as a whole.

In this paper we propose a system that leverages the inherent features of IPFS to reduce the size of the Ethereum blockchain. We propose that code in Ethereum CCTX's is stored in IPFS, instead of on the blockchain directly. Currently, the bytecode required to create and run a contract is stored in a CCTX. By moving this code off-chain and storing a hash in the transaction instead, the size of the chain and chain growth can be reduced. As IPFS identifies files by their hash, the inclusion of the hash of the code in the CCTX would allow for efficient and assured retrieval of the code from IPFS with the hash providing assurance that the retrieved code is correct and unchanged.

Ethereum contracts are stored in the state trie, meaning it is rare that contract code is retrieved from the transaction that created it. Nodes can download previous states from archive nodes. Therefore the code stored in CCTXs is bloating the chain. The data can be moved off-chain with little impact on performance and a significant impact on the size of chain data. IPFS is capable of removing unwanted files using garbage collection commands. `ipfs daemon --enable-gc` for automatic collection and `ipfs repo gc` to carry out removal manually [1].

Users can prevent the local removal of a file under IPFS by pinning it. Doing so tells IPFS never to remove it. If a user wishes to retain a file they can pin it, and in doing so avoid its permanent removal, while allowing other nodes to save space by not storing it. Under our proposed system the files stored on IPFS would be the contract data. If any node requires the data contained in a given file they simply have to request the hash stored in the chain.

Ethereum has a number of full archive nodes that hold the data of all the previous states of the Ethereum blockchain. The functionality of such a node is not affected by our proposal as it simply needs to pin all contracts to retain all the data that has been moved off-chain to continue acting as a full archive node.

Standard nodes, when fast syncing, do not verify all transactions. Therefore replacing the code sent in transactions with hashes allows the nodes to request and store less data. Ethereum makes heavy use of Merkle trees, originally conceived in [2]. As each header contains the hash of the previous block and various root hashes of Merkle trees for a variety of relevant information, including transactions, the integrity of the code is assured. The hash in the CCTX cannot be changed without changing the hashes in the tree, and therefore in the next block as well. The code associated with the hash cannot be changed without changing the hash. Therefore the code retrieved from IPFS is as secure as when the code is stored directly on the chain. This not only reduces the storage space required for Ethereum end users but also lessens network traffic as nodes using fast sync to catch up need only be sent the hashes, and not the full code, in order to acquire the transaction root hash for each block.

The main contribution of this work is reducing the size of stored Ethereum chain data by moving the bulk of contract creation data off-chain. To do so we make use of some of the inherent properties of IPFS. This reduces the amount of data sent over the Ethereum network as contract creation transactions can be stored without the full contract code.

The rest of this paper is structured as follows: in section 2 related work done to reduce the size of blockchains is covered. Section 3 details the key features of IPFS and Ethereum for our proposal, and how we utilise them. Section 4 covers how we constructed our dataset and how code replacement can be done. Lastly section 5 gives a detailed analysis of our experimentation and section 6 is the conclusion.

2. Related Work

Size and growth of public blockchains has been generally acknowledged as a problem. As append-only ledgers the size will only ever increase. In recognition of this fact, various different methods have been used to reduce growth.

Nodes & syncing. Three different types of node exist in the Ethereum network:

- Full Archive Node: Stores every block and state in the entire chain. Fully validates all transactions. These nodes are generally only run by service providers such as `etherscan.io`.
- Standard Node (using fast sync): Downloads all blocks but checks only block hashes and receipts, not the details of each transaction. This allows it to skip downloading and/or storing each state in turn. Stores only the current state and discards old ones. Fast sync is well described in [3].
- Light Client Node: Stores only block headers and not full blocks. A light client is incapable of full independent verification and must request transactions it is interested in from other nodes in order to perform a minimal form of verification.

Of the above implementations, the latter two both aim to reduce the space required to store the chain by trading security for speed and size. Both rely on other nodes in the network to store extra data should they need to refer to it at any point. EIP 170 sets a hard limit on the number of bytes that a contract can be [4]. Although primarily done to protect against attacks that could slow down nodes, it also restricts the amount of data that can be added to the blockchain in one contract. In doing so acts as a space saving mechanism.

Moving data off-chain. Work has been done to reduce the size of Bitcoin's blockchain by moving a lot of transaction data off-chain. This idea has been realised in the form of the Bitcoin Lightning Network [5], wherein participants can conduct transactions off-chain and only submit the start and conclusion of their transactions to the main-chain, thus reducing its size by requiring it to permanently record less transaction data.

Plasma [6] is being developed for Ethereum. It also moves data off-chain. It aims to store transactions and smart contracts, or parts thereof, off the main-chain in order to reduce the rate of its growth.

Block data. Ethereum allows users to create and store smart contracts. They are created by announcing a transaction to the network with the code for the contract in the data field. According to the Ethereum yellow paper the initialisation code is used only once to create the contract and then discarded [7]. However, as the root hash of the Merkle tree of transactions in a block forms part of the data stored in the block header, it is necessary to retain the contract creation data as part of the transaction. The transaction data is also required when any node wishes to fully verify historical transactions. Contract creation transactions are the target of our proposal to reduce the size of the Ethereum chain.

State tree pruning. Ethereum account balances, runtime contract code and contract storage data are stored as part of the state, which is stored as a tree. Storing each full state can consume a large amount of space, as such regular nodes store only the current state. The state is continually pruned to remove data which is no longer relevant. For example, old account balances can be removed from it. However, only states are removed, the contents of the other trees whose root hashes form part of the block header are retained. Notably for our purposes this includes the transactions in a block. A good description of pruning can be found in Vitalik Buterin's blog post on the subject [8].

Decentralised storage. We leverage the properties of the Ethereum blockchain to provide assurances about the integrity of data stored off-chain. This has been done for a different, but related purpose in the Filecoin system [9]. Under Filecoin one can pay to have files stored in a distributed way, using the hash of the file stored on the blockchain to provide assurance for the integrity of file when retrieving it.

IPFS. IPFS uses hashes to identify files [10]. This provides the highly useful property of being able to verify a file by checking its hash against the name (hash) of the file you were looking for. The equality of the two acts as proof the right file was received. Filecoin is build on top of IPFS.

Swarm. Ethereum's developers are already working on a decentralised system similar to IPFS for off-chain storage. The main benefit over IPFS is the incentive system provided to users. Although financial incentives can be brought to IPFS, as done by Filecoin. Swarm is still under heavy development [11]. The Solidity compiler appends a Swarm hash to a contract's metadata. In the event of an eventual release, our proposed solution would be seamlessly translatable from IPFS to Swarm.

3. System Architecture

In this section we detail the key features of IPFS and Ethereum that are relevant to our proposal. We describe how each feature is leveraged or changed for the purpose of reducing the size of chain data.

3.1. Key features of IPFS

In this section the key features of IPFS that this project makes use of are discussed in more detail. Full details of the IPFS system can be found in [10].

Garbage collection. Garbage collection removes the blocks of files which are unpinned. If there is a file that the user wishes to retain they can pin it. Pinning marks files that are not to be removed by IPFS. The user can pin/unpin a file using `ifps pin`. Files can be pinned when they are added to the IPFS using using `ipfs add pin` [1].

In our proposal the garbage collection feature is used to help reduce the storage space required for the Ethereum chain. In the case of regular or fast syncing clients the default will be for IPFS to store nothing. This fits well with IPFS's paradigm of storing only what is specifically requested by the user, as well as with Ethereum's state pruning whereby information not immediately required can be removed.

In our proposal the code moved off-chain is identified by its 32 byte SHA256 hash. Hashes would be stored for blocks of code $>$ 33 bytes in size, while code that is <= 33 bytes could be stored directly on chain, as no space would be saved by moving them to IPFS.

Hashing. IPFS is capable of using different kinds of hashes to identify files with its multi-hash format. It consists of three fields which are as follows: The first is one byte in length and denotes the kind of hash (SHA2, SHA3, etc), the second field is also one byte and denotes the size of the hash. The third field contains the hash itself, its size defined by the value of the previous field.

Block reuse. In our previous work we demonstrated that contracts can have similar bytecode and similar functionality [12]. IPFS stores data in blocks which are up to 256 kb in size. Given the size of contracts, each will be stored in a single block. IPFS has the ability to assign the same block to multiple files if they share one or more identical blocks. In our scenario, if two smart contracts are identical then anyone storing both contracts would only need to store the block once. The chance of this is further increased as each contract is split into init code and runtime code. Identical contracts will result in an identical hash stored on the chain. As this indicates an identical instance of the code it should not be considered a problematic collision. It further increases the space saving when using IPFS.

Pinning. Regular and light Ethereum nodes request information they do not store locally from archive nodes when they need it. Under our proposal the data from CCTX's is held by archive nodes and can be requested when required. An archive node pins every file associated with a hash in a CCTX's data field. Any node can retrieve the bytecode associated with a hash at will, without the current necessity of storing it permanently as part of a block. The role of archive nodes remains the same under our proposal and the data we move off-chain can be retrieved in the same way. Pinning also opens up an interesting new ability for a user to ensure the existence and availability of a contract they deem to be important. For example, if one has a private wallet contract one could pin it to ensure its continued existence without relying on archive nodes. As long as the pinned local file(s) exist it can be requested by other nodes.

IPFS file propagation. By default IPFS does not store anything the user does not explicitly request it to. This will change the way data for contracts propagates through the Ethereum network. Under the current system, upon hearing about a transaction, a node will forward the transaction to its peer. In this way new transactions are flooded through the network. Under our proposal flooding would still take place for transactions. However in the case of CCTXs, after becoming aware of such a transaction nodes would have to carry out the additional step of requesting the contract bytecode from IPFS using the hash(es) in the CCTX. This involves adding the hash(es) identifying the code to their IPFS 'want list'. The node(s) from which the files can be obtained are discovered by searching the Distributed Hash Table (DHT). In the DHT file locations are stored based on the closeness of hashes at bit level. The significance of the DHT is discussed in our results in section 5.2. This is a shift from only listening for data to listening followed by requesting data. The model therefore shifts from push to push followed by pull. IPFS's default behaviour of storing only explicitly requested data enables us to minimise the local data storage per node. For code with a size $<$ 33 bytes IPFS is not used and the original transaction propagation would take place.

3.2. Key Features of Ethereum Transactions

In this section we describe the key features of Ethereum transactions and how they are adapted for space saving under our proposal.

CCTXs. Currently, Ethereum CCTX's consist of the fields shown in 1.

TABLE 1: Field of an Ethereum CCTX

CCTX					
nonce	gasPrice	gasLimit	ether	to	data

The data field is the relevant one for our proposal, it is an unlimited size bit array, in a CCTX it holds the code that a contract will have. Formally, it contains code that returns the contract's runtime code. This takes the form of init code followed by runtime code. Init code is discarded after contract creation, and is not used as part of the new contract. In our system the hash of the file containing init or runtime code, as stored in IPFS, would be placed in the data field. Verifying nodes would then get the blocks that comprise the file through IPFS, check that the hash of the combined blocks matches the one in the transaction, and then verify the associated contract code.

Block headers & trees. Ethereum stores a number of root hashes of Merkle trees in the block header. Merkle trees have transactions as their leaves, each transaction is hashed and trees are formed by combining and hashing inputs until the root is reached. One such tree is comprised of transactions. This means that block headers rely on the data of all transactions and full verification requires the full data of all transactions in block. Figure 1 shows an example of a transaction hash tree in Ethereum with the data field being populated by the data to be retrieved from IPFS.

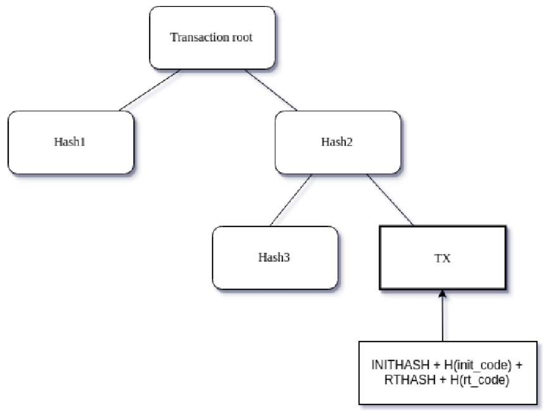

Figure 1: Ethereum transaction tree including proposed opcodes and hashes

Storing hashes. The most commonly used hash in IPFS is SHA-256. This hash is 32 bytes in length, most contracts are longer than this: between 514 and 1027 bytes according to our dataset. According the Ethereum's yellow paper init code is run once, then discarded [7]. Init code frequently exceeds a length of 32 bytes. It is therefore possible to retard the rate at which the blockchain increases in size if contract code is replaced with a hash that describes a file stored in IPFS. Contracts below 33 bytes continue to be sorted on the chain as there is no space to be saved by storing them off-chain. This necessitates that hashes representing init and runtime code must be prepended with an extra byte indicating whether they are code to be read directly or a hash of a file to be retrieved from IPFS.

Contract retrieval. In blockchain systems like Ethereum, weight increases trust in a transaction. Weight is the amount of valid blocks appended after the block containing the transaction in question. The more valid blocks, and therefore valid proofs of work, on top of the block containing a giving transaction the less likely it becomes that the chain will fork below the given block. As a result weight equates to trust. Therefore contracts that are less than n blocks old could be pinned by default, where n is defined as the number of blocks required to provide sufficient weight. Meaning that no contract without sufficient weight can be removed. The contents of an IPFS file are assured in the same way regular transaction data is assured; by proof of work, or the weight of the blocks on top of a given block.

Optionally, blocks with sufficient weight could be trusted without ever retrieving the code and checking the hash. Fast syncing nodes do not fully check old transaction up until recent blocks at which point they begin to fully verify all transactions in a block. With our solution they would not need to request the data for CCTX's that are not being fully checked. However, in the unlikely event that no achieve nodes are able to provide the code associated with a given hash and assuming no user has the file pinned in IPFS a user could decide to trust the block based on its weight.

In section 3.1 we describe how the interoperation of IPFS and Ethereum will change the way data propagates through the network in the case of CCTX's. This change also affects the way a user launching a contract on Ethereum is to behave. Currently a user announces a CCTX to the peers they are connected to and can then disconnect from the network with relative confidence that the transaction will propagate through the rest of the network and be included in a block. However under our proposal the user must also remain online long enough for their immediate peers to request and receive the data pertaining to the IPFS hash(es) in order for it to be propagated through the network as well. As our experimental results in section 5 show, this is likely to be no more than 541ms. As retrieval from IPFS will be an infrequent event, and block confirmation time is around 14 seconds, our proposal will have little impact on the user sending the CCTX.

3.3. Technical summary

Our proposal involves adding contract data to IPFS unpinned, with a limited IPFS storage space so that contract creation data will be quickly removed from nodes' local storage after the transaction has been verified and the contract account created. IPFS acts as a cache for CCTX data which is not required for the general operation of the chain due to Ethereum holding the data it needs quick access to in the current state. The off-chain data can be retrieved through IPFS, from archive nodes as and when necessary. The data field of a CCTX would, in the event of code being stored off-chain, hold two opcodes which instruct the EVM to retrieve the code from IPFS by reading the 32 bytes following the opcode as the hash. Table 2 shows an Ethereum transaction with the data field holding the two opcodes and the corresponding hashes. With `INITHASH` and `RTHASH` being the opcodes and $H(x)$ being the hash of x.

TABLE 2: CCTX with init field holding proposed opcodes and hashes

CCTX				
nonce	gasPrice	gasLimit	to	`INITHASH` + $H(init\_code)$ + `RTHASH` $H(rt\_code)$

Figure 2 shows the interactions between system components. When both sections of code have been replaced with hashes, the steps are as follows:

1) The EVM reads one of the proposed opcodes and takes the 32 bytes after it as the file hash to collect from IPFS
2) The EVM requests the file through IPFS
3) IPFS returns the file matching the requested hash
4) The EVM verifies the hash of the file and creates a new contract in the standard way

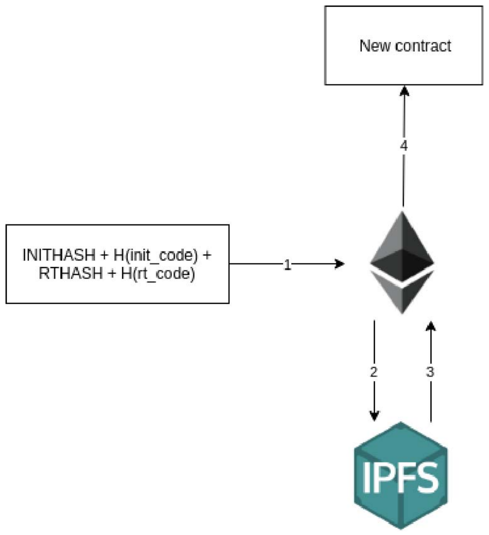

Figure 2: Opcode, EVM and IPFS interactions

IPFS records the hashes of files in its own multihash format. As our proposal stores 32 byte SHA256 hashes on-chain, the EVM must build the multihash format in step 1 after getting the hash(es) from the CCTX. The format of the IPFS multihash can be seen in table 3. The EVM must build the multihash by appending two byes to the SHA256 hash. The first field (fn code) denotes the type of hash, the second (length) dictates the number of subsequent bytes to be taken as the hash. IPFS deals with base58 encoded hashes. The EVM must encode the constructed hash in base58. Once the multihash is constructed and encoded it can be requested from IPFS.

TABLE 3: IPFS multihash format

IPFS Multihash		
fn code *(1 byte)*	length *(1 byte)*	hash digest *(variable len.)*

4. Methodology

In order to have a sample of recent CCTXs for our dataset we use a scraper to collect all contract creation transactions for a given number of blocks in Ethereum's main-chain from `etherscan.io`. The raw bytecode is translated into its equivalent opcode format. Each contract is split into init code and runtime code. This is done by finding the first instance of the 'STOP' opcode, with everything up to and including the 'STOP' being init code and everything after it being runtime code. The length of the two parts of each contract is calculated. As each opcode in Ethereum is one byte long the number of opcodes is summed. 'PUSH' instructions take a subsequent number of bytes between 1 and 32 as a value to be pushed to the stack. As these values are a hard-coded part of the contract code, the sum of opcodes has the cumulative number of bytes denoted by 'PUSH' operations added to it. This forms the total size in bytes of each part of each contract. A contract's total size is the totals from the init and runtime parts summed together.

The savings made per contract are calculated by subtracting, from each of the two parts of a contract's creation code, one byte for the opcode that signifies the data is to be read as a hash, and the length of the hash. In simple terms: $codelength - 33bytes$. If the resulting value is > 0, and therefore a positive saving, it is added to the total savings. If it is < 1 then the saving is 0 as we leave parts of a contract smaller than 33 bytes on the chain.

We propose the two opcodes: `INITHASH` & `RTHASH` to act as instructions for when the EVM is to request files from IPFS. Under our proposal the data field for a CCTX would have the format seen in figure 4 for a CCTX where both init code and runtime code have a length > 33 bytes. The format can be adjusted for either code section being < 33 bytes by placing that section of code directly on the chain. By defining two new opcodes we are able to maintain backwards compatibility with older contracts in Ethereum. If one of the new opcodes is encountered the EVM would acquire the code via IPFS, check the hash matches, and then continue execution on the retrieved bytecode. If the opcode is not encountered the EVM will continue execution as normal. Therefore both new CCTXs with hashes, and those without will be able to function.

Formally, the construction of the data field of a CCTX is done according to the following: where *H(x)* is a function returning the SHA256 hash of *x* and *len(y)* is a function returning the number of bytes making up the bytecode *y*. *ic* and *rtc* are init code and runtime code respectively. + denotes the concatenation operation.

$$data = ic + rtc \tag{1}$$

Where:

$$ic = \begin{cases} INITHASH + H(ic), & \text{if } len(ic) > 33 \\ ic, & \text{otherwise} \end{cases} \tag{2}$$

and:

$$rtc = \begin{cases} RTHASH + H(rtc), & \text{if } len(rtc) > 33 \\ rtc, & \text{otherwise} \end{cases} \tag{3}$$

In section 5 we include results for replacement of both code sections with hashes (two-hash solution) and for replacing only runtime code (single hash solution) that replaces only runtime code and not init code. The comparison of the two highlights that the lesser used and usually smaller init code section has a significant impact on chain size. By leaving init code on chain the number of IPFS retrievals the system needs to perform are halved, assuming both parts of code are $>$ 33 bytes.

TABLE 4: Structure of CCTX data field for using hashes

INITHASH (1 byte)	init code (variable length)	RUNHASH (1 byte)	runtime code (variable length)

All the code used for this work, including the scraper used to create the dataset and the code to carry out all tests, calculations and graphs is available at [13].

5. Experimental Results

Our experimental results are split into two parts. Section 5.1 looks at the results and impact on size of the one and two-hash solutions as applied to our dataset. Section 5.2 looks at the impact of IPFS retrieval times and provides some analysis of file size based differences.

5.1. Code size

Our dataset contains all 1403 CCTXs in the 1048 blocks from the 5,000,000th block onwards. By counting the number of opcodes and the byte length of push values for both init and runtime code in each CCTX in our dataset we calculated the total size taken up by the code. We found that runtime code was larger than init code for 1345 of the CCTX's in our dataset. 1027 contracts, which is 73.2% of the dataset, fall within the range of 514 to 1236 bytes. Although far larger contracts exist, their occurrence is extremely rare. The single largest contract is 19,014 bytes in

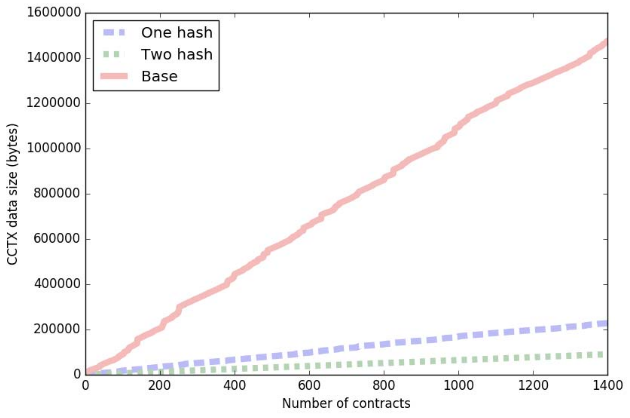

Figure 3: Cumulative size of contract data in the transactions of our dataset

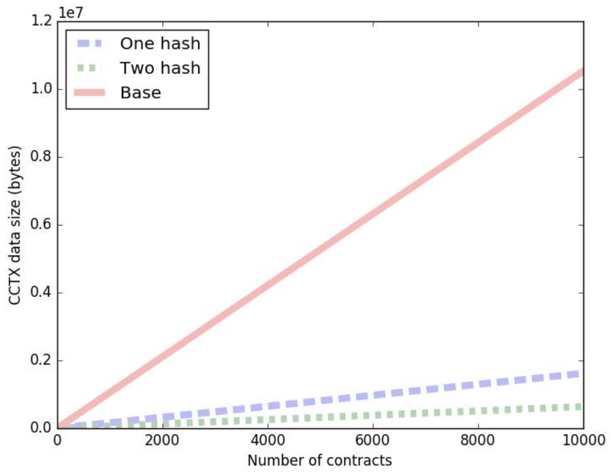

Figure 4: Average size growth per CCTX

size. We calculate the mean average size of contracts to be 1053 bytes, this contract size is used in our experimentation as one of the file sizes to test retrieval times. The largest init code is 2150 bytes and the largest runtime code is 18745 bytes. Mean init code size is 131 bytes and mean runtime code size is 922 bytes.

TABLE 5: Comparison of total and average size of CCTX code in bytes

		Base	**Single Hash**	**Double Hash**
Size	init	183871	183871	46106
	rt	1293963	44649	44649
	Total	1477834	228520	90755
Reduction (%)		$\sim$	84.54	93.86
Avg. size per block		1410	218	87

Table 5 shows the total size of each code segment for the base code, one and two-hash solutions. The runtime code makes up the majority of the base data, making the hashing solutions highly effective. However, the two-hash solution offers a larger reduction due to init code often being large enough to benefit from being replaced by hashes.

Figure 3 shows the cumulative size of CCTX data per CCTX in our dataset for the base data as well as our one and two-hash solutions. The two-hash solution shows a dramatically smaller rate of growth both the one-hash solution and the base size.

Figure 4 shows the estimated, cumulative size of CCTX data based on the mean average sizes of the CCTX data as it is in the dataset and for the one and two-hash replacements. These averages are summed cumulatively, up to 10,000 CC-TXs. The two-hash solution provides the greater reduction due to it being common for the init code to be > 33 bytes.

5.2. Retrieval times

In order to test the impact of IPFS retrieval on Ethereum test files were generated with specific sizes: 66, 1053, 19,014 and 1000000 bytes. The first is the threshold at which code replacement takes place, the second is the mean size of contract bytecode from our dataset, the third is the size of the largest contract in our dataset. Finally, 1000000 is used as a stress test, it is larger than any smart contract is likely to be in the foreseeable future.

We generated 5 batches of 100 files of random data for each of the file sizes specified above and added them to IPFS. The list of resulting IPFS hashes for each batch was passed to a server running a separate IPFS node. It attempted to retrieve each file and recorded the time taken to do so. The node serving the files was run in server mode to enforce external connections only, as is likely to the be the case in a real world scenario. This procedure was carried out for each of the 5 batches of each file size.

We take the mean retrieval time of each batch of test files. All file sizes had a similar mean average retrieval time. The total spread is 62ms which suggests that IPFS is well suited to transferring the upper limit of present day contracts. The mean retrieval time for 1000000 byte files is only 42ms higher than that of 19,014 byte files, suggesting that IPFS will be suitable even in the face of large increases in contract size. The mean average retrieval time for all files across all batches is 541ms.

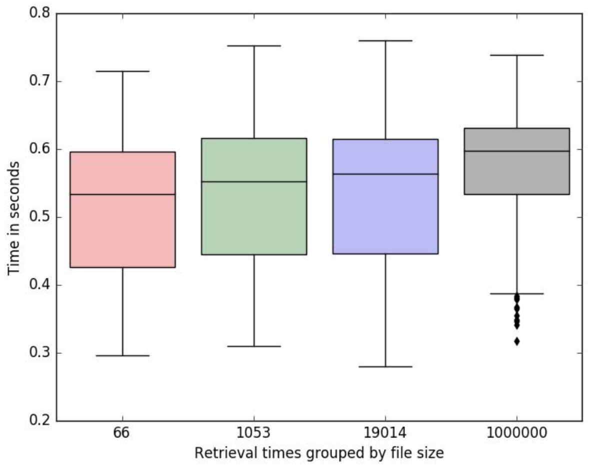

Figure 5: Boxplot for retrieval times by file size

The boxplot in figure 5 shows very similar medians and distributions between the first 3 file sizes. The rightmost plot, for 1000000 byte files, has a slightly higher median and a much smaller spread in the data. It is the only plot with any outliers.

We calculate the intersection for each pair of retrieval time histograms using the equation presented in [14] and shown here in formula 4.

$$int(T_x, T_y) = \frac{\sum_{i=1}^{n} min(T_x[i], T_y[i])}{\sum_{i=1}^{n} T_x[i]} \tag{4}$$

Where T_x and T_y are histograms of file retrieval times with x and y defining the file size. For example T_{66} is the histogram of retrieval times for 66 byte files. i denotes a particular bin in a histogram, with $T_x[i]$ being equal to the number of retrieval times in the i^{th} bin. min is a function that takes two bins $T_x[i]$ and $T_y[i]$ as arguments and returns the smallest of the two values. The result of $int(T_x, T_y)$ is the sum of the number of shared values in each bin as returned by the function min, divided by the number of values in the T_x to normalise the results. The function assumes both histograms have the same number of bins with the same ranges.

Figure 7 shows that the the plot for $T_{1000000}$ is left skewed. The plots for T_{66}, T_{1053} and $T_{19,014}$ all display a bimodal distribution. Table 6 shows the percentage of intersection of all pairs of histograms. T_{66}, T_{1053} and $T_{19,014}$ have a mean average intersection of 0.8809 and all 3 have intersections > 0.85. Each intersection with the $T_{1000000}$ is < 0.79, which is significantly lower than that the intersection between the other 3 histograms.

Figure 6 shows the intersection of the 3 bimodal distributions. Given the higher intersection of these 3 and the different, left skewed distribution of $T_{1000000}$, we conjecture that the two peaks in the smaller 3 are due to IPFS's DHT. IPFS stores files based on hash. For smaller files the primary determining factor for retrieval time is the number of nodes in the DHT that have to be traversed. However, 1,000,000 byte files are large enough for internet bandwidth to become the primary determining factor over DHT look-up time. This explains the left skewed distribution for the largest file size in our experiments.

TABLE 6: Intersection of all pairs of bimodal latency distributions

Pairing	% intersection
$T_{66} \cap T_{1053}$	88.56
$T_{66} \cap T_{19,014}$	88.96
$T_{1053} \cap T_{19,014}$	86.75
$T_{66} \cap T_{1000000}$	71.89
$T_{1053} \cap T_{1000000}$	77.71
$T_{19,014} \cap T_{1000000}$	78.51

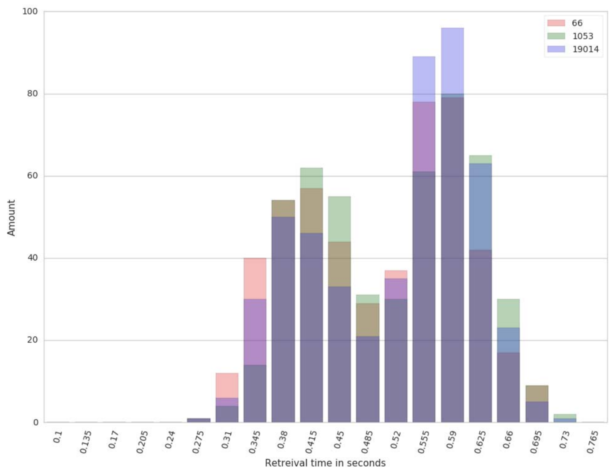

Figure 6: Overlay of plots for T_{66}, T_{1053} and $T_{19,014}$

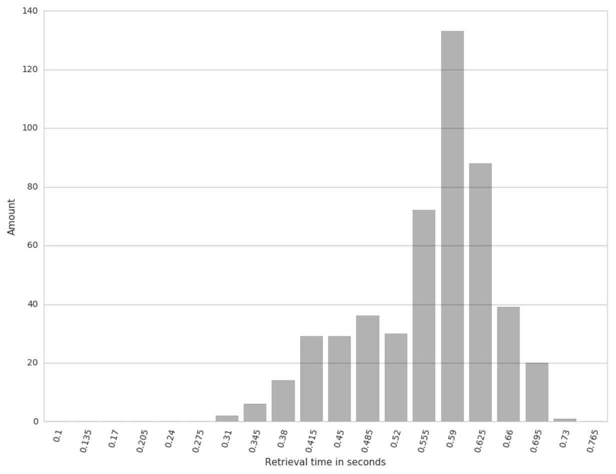

Figure 7: Plot for $T_{1000000}$

6. Conclusion

In this paper we present a method for reducing the size of Ethereum chain data and reducing the rate of growth for regular nodes by moving the code stored in contract creation transactions off-chain and replacing it with a smaller hash that allows for the retrieval of the code. This is achieved with minimum impact on system performance and ensures nodes retain the ability to fully verify transactions. As a positive side effect fast syncing nodes have to download less data.

7. Future Work

Future work includes building a working implementation of the proposed idea by modifying the EVM to incorporate the proposed bytecodes. Investigating other candidates in Ethereum's data and transactions to be moved off-chain. Lastly, we would like to incorporate the idea of code similarity as shown in [15] to further improve the size reduction when utilising IPFS.

References

[1] P. Labs, "Commands — IPFS Docs," https://ipfs.io/docs/commands/, 2017, [Online; accessed 5th April 2018].

[2] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1987, pp. 369–378.

[3] P. Szilgyi, "eth/63 fast synchronization algorithm," https://github.com/ethereum/go-ethereum/pull/1889, 2015, [Online; accessed 8th March 2018].

[4] V. Buterin, "Ethereum Improvement Proposal: Contract code size limit ," https://github.com/ethereum/EIPs/blob/master/EIPS/eip-170.md, 2017, [Online; accessed 3rd April 2018].

[5] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," *draft version 0.5*, vol. 9, p. 14, 2016.

[6] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," *White paper*, 2017.

[7] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[8] V. Buterin, "State Tree Pruning," https://blog.ethereum.org/2015/06/26/state-tree-pruning, 2015, [Online; accessed 8th March 2018].

[9] P. Labs, "Filecoin: A decentralized storage network," 2017.

[10] J. Benet, "Ipfs-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.

[11] *Swarm - ethersphere/swarm, author = Ethereum Foundation, note = Available at https://github.com/ethersphere/swarm , as of March 2018.*

[12] R. Norvill, B. B. F. Pontiveros, R. State, I. Awan, and A. Cullen, "Automated labeling of unknown contracts in ethereum," in *Computer Communication and Networks (ICCCN), 2017 26th International Conference on*. IEEE, 2017, pp. 1–6.

[13] R. Norvill, "ethereum-ipfs-paper-code Private," https://github.com/pisocrob/ethereum-ipfs-paper-code, 2018, [Online; accessed 14th April 2018].

[14] M. J. Swain and D. H. Ballard, "Color indexing," *International journal of computer vision*, vol. 7, no. 1, pp. 11–32, 1991.

[15] B. B. F. Pontiveros, R. Norvill, and R. State, "Recycling smart contracts: Compression of the ethereum blockchain," in *New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on*. IEEE, 2018, pp. 1–5.