Merkelized Abstract Syntax Trees

Jeremy Rubin, Manali Naik, Nitya Subramanian

{jlrubin, mnaik, nityas}@mit.edu
https://github.com/JeremyRubin/MAST

1. INTRODUCTION

In the context of modern cryptosytmes, a common theme is the creation of distributed turst networks. In most of these designs, permenant storeage of a contract is required. However, permenant storage can become a major performance and cost bottleneck. As a result, good code compression schemes are a key factor in scaling these contract based cryptosystems. For this project, we created a new data structure called the Merkelized Abstract Syntax Tree (MAST) to address both data integrity and compression. MASTs can be used to compactly represent contractual programs that will be executed remotely, and by using some of the properties of Merkle trees, they can also be used to verify the integrity of the code being executed. The project idea was developed with Bitcoin applications in mind, and the experiment we set up uses MASTs in a crypto currency network simulator. Using MASTs in the Bitcoin protocol [**?**] would increase the complexity (length) of contracts permitted on the network, while simultaneously maintaining the security of broadcasted data. Additionally, contracts may contain priviledged, secret branches of execution.

2. MAST DATA STRUCTURE

MASTs combine the traits of Merkle Trees [**?**] and Abstract Syntax Trees (ASTs) to compactly and securely represent programs. Merkle trees are data structures that can be used to efficiently verify the integrity of ther data they store. Data blocks are stored in the leaf nodes, and every non-leaf node is the hash of the labels of its children nodes (see Figure 1). In the Bitcoin Blockchain, Merkle trees are currently used to efficiently store transaction history. ASTs, on the other hand, represent the syntactic structure of programs. Primitives are located at the leaf nodes of ASTs, and non-leaf nodes represent programmatic operations and control flow mechanisms.

In a MAST, the root of the tree represents the entirety of the program, while all other nodes represent subprograms. Each path in the tree is a different execution branch that the program can take. The structure is Merkelized in that leaf nodes are hashes of the subprogram code that they represent, and non-leaf nodes are hashes of the children labels. MASTs can therefore compactly represent the execution flow of a program with just a sequence of hashes that specify which child edge to follow at each node. Overall, this means that for a program of length n, a compression to $O(\log n)$ could be expected.

3. IMPLEMENTATION

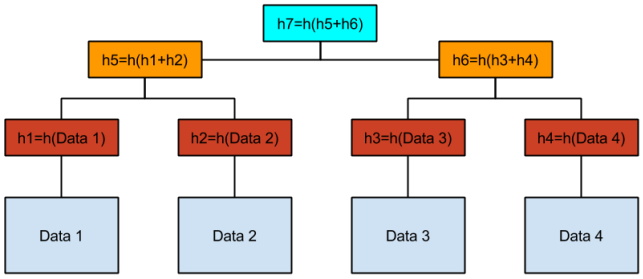

Figure 1: An example of the construction of a Merkle Tree

3.1 MAST Nodes

A MAST node is constructed with string content and parent pointers. The string content is code that can be executed. A MAST node can have any number of children, each one representing a different branch of program execution, and new branches can be added via the `addBr` method. Joining all the string content from each node along a path in a MAST therefore yields the code for one possible path of execution for a program.

Since branches in a MAST dont have to be added in any particular order, maintaining consistency during tree construction would be cumbersome. Instead, that hash function of a MAST node computes the correct Merkle hash using the current state of the tree. It does so by first forming a binary tree of all direct children nodes. As this tree is constructed, hashes are concatenated and hashed to compute a new Merkle hash at each level of the binary tree. Then, the root hash of the children tree is summed with the hash of the nodes own content, producing a Merkle hash representing the node's code and children. By not including the node's content in the binary tree we establish a syntax-tree style construction where the parent executes before the child. Figure 2a shows the structure used to calculate the Merkle hash of a given MAST node. The four children branches are placed in a binary tree whose root is the yellow "Branch Merkle Root." This Branch Merkle Root hash is summed with the content hash on the left to give the Merkle Root.

3.2 Proof Lists

In order to verify the integrity of code, the MAST function `generateFullProofUpward` generates a proof list to a given Merkle root hash from the current node. It does so by

traversing the tree upwards, generating a list of the Merkle hashes and code content it passes along the way. The logic gets more complicated due to the fact that Merkle hashes are calculated using a binary tree (as described in Section 3.1). As a result, the proof list generator crawls up this binary tree until it hits the branch Merkle root (the parent MAST node), and then repeats the process until the destination node is reached. The content and hashes included in the proof list for a piece of content is shown in Figure 2b.

For the verification process, we assume that another machine (without the entire program code) has the Merkle hash of the MAST root node. With a proof list whose destination node is the MAST root hash, this other machine could verify the code in the proof list by iterating over it, summing up the hash values to make sure that they add up to the next hash value in the proof list. If the final summation yields the root hash, then the sequence of hashes provided is correct. We check the integrity of the code against hashes in the proof list as content hashes are represented.

Additionally, scripts can be compiled to a proof format that is compatible with opcodes used on the Bitcoin Blockchain. We did not put any MAST's onto the blockchain, but tested it via a bitcoin script interpretor we wrote. This further augments compatibility between our MAST implementation and the Bitcoin protocol and allows for potential future extensibility of this project to be integrated within the Bitcoin Blockchain.

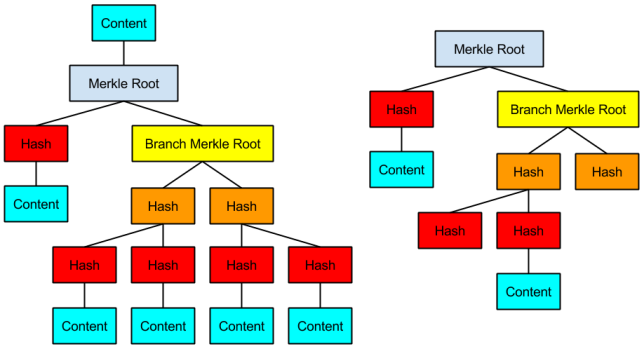

Figure 2: Structure of a simple MAST and accompanying proof. The recursive data structure (left) and a proof for a particular branch of execution for a MAST (right)

3.3 Consensus Protocol Simulation

We implemented a crypto currency network simulator to use MASTs to verify and execute code transferred over the network. To simulate the transaction verification process used in Bitcoin, we created `ConsensusNode` objects to execute and validate the code in a transaction. We implemented types of `ConsensusNode`: `GoodNode`, `EvilNode`, and `InconsistentNode`. `GoodNode`s execute code properly and include transactions in their local ledgers if they are valid. `EvilNode`s can add invalid transactions to their ledgers and exclude valid ones. `InconsistentNode`s behave correctly with some probability. With the three node types, we can simulate more realistic network conditions and the presence of adversaries.`ConsensusNode` can be further subtyped to simulate other types of adversary.

Code for a transaction is stored in a MAST, and the transaction saves the corresponding root Merkle hash. The compression MASTs offer reduces the amount of data that has to be transferred between nodes during validation, as well as the amount of data that is stored in the ledgers. Rather than sending and storing the entire transaction code, we only transmit and store the desired path through the MAST (as a sequence of Merkle hashes). Using an args array, we support passing arguments to transaction code and specifying the subsequent transactions to be run. Transactions also have associated amounts that are used to check whether the code is valid. A valid transaction is one whose amount is at least as large as the sum of amounts of its subsequent transactions.

After individual nodes validate/invalidate a set of transactions, a special `GlobalConsensus` node determines the final outcome of a transaction; a transaction is valid if a majority of the `ConsensusNode`s validate it. The `GlobalConsensus` node updates a global ledger representing the correct state of the system, and `ConsensusNode`s sync with this ledger at each simulation tick to maintain accurate state.

4. APPLICATIONS AND EXPERIMENTS

4.1 Contractual Agreements

Using the previously discussed consensus protocol, we implemented a contract modeling a will that utilized the shared contract creation, verification, and execution functionalities of the MAST to create a multiparty execution environment. The will-based contract we implemented consisted of an agreement between three parties: Alice, Bob, and Carol. The branches of the tree represent approved expenditures both before and after Alices death, and present clauses of the contract as a series of branches that are unlocked upon the fulfillments of certain conditions in conjunction with the signatures of relevant signatories. This structure allows for sub-contracts to exist within the subset of the primary signatories and enables full transparency of the content of the contract while restricting execution of certain clauses to when necessary preconditions are satisfied.

The consensus protocol to verify valid construction and execution of the will was implemented using a group of `ConsensusNode`s (comprised of the three types discussed above) which each verified the validity of a transaction.

4.2 Code Compression

The primary benchmarks used to quantify code compression were a series of python scripts, which can be found in our repo at the location MAST/bin. The file longcode.py (Figure 3) generates a Merkle tree with tens of thousands of branches, yielding a total code length of 2M characters, The post-compression result after applying our algorithms was under 23,500 characters, indicating a compression rate of over 98directly to compressing with another algorithm such as LZW because such algorithms employ frequency based compression and longcode uses repeated segments again and again This is OK to do because we perform structural compression. Instead, we compared it to using zip to compress code from the Linux kernel. This took it from roughly 6115332 characters in length to 1754665 characters, a compression rate of 70could also be used in several places in MASTS as well: per code block, and on the complete proof list. Additionally, we use an unoptimized message format to

```
ex = "some code"
M = Mast("compile", "")
n = M
for i, c in enumerate(X*[code]):
    [n.addBr(c) for i in xrange(Y)]
    n = n.addBr(c)
proof = n.generateFullProofUpward(M.hash)
# Run in simulator
merkleVerifyExec(M.hash(), proof, 10)
# Generate and run as Bitcoin script
scriptSig, script = toScript(pr, M.hash)
full = scriptSig + script
run(full)
print "compression rate:", 1-len(together)/(len
    (code)*(Y+1))
```

Figure 3: An example demonstrating code compression on a Merkle tree with millions of characters. Conversion to a MAST representation yielded a compression score of over 98.9%

send MASTS which is essentially a list of $[([subproof], data, mroot)]$. This could be further optimized to not use punctuation and whitespace and use more efficient character encodings. Surprisingly, encoding the proof in a bitcoin script (ie, self proving) was more efficient than having an external validation script.

5. SIGNIFICANCE AND FUTURE WORK

The primary impact of this projects is in its applications to established environments utilizing contracts. The introduction of MASTs has potential to greatly impact existing problems ranging from bitcoin contracts to code transfer between distributed nodes on a network. Potential improvements to this implementation of the MAST could include greater support for distributed construction and execution or the addition of a framework allowing greater extensibility by users of this data structure. Complete integration with Bitcoin would reduce the amount of data that is stored in the Blockchain, and will make it possible to perform more complex transactions like the will we modeled.