ProjectEuanthe

Streamlined Circuit Operations Inside Elara

Working Draft v1.0

draft@projecteuanthe.org

SEPT 26, 2021

1 Introduction

Projecteuanthe is a set of Decentralized Invisible Transaction (DIT) instances built on Ethereum. It deals with two modules, namely ebb and elara, with different schemes leading to private transactions on Ethereum through the use of group signatures and shielded contracts, respectively. This paper deals with the problems of intensive computational complexity inside a snark and explores a balanced collision-resistant hash function to work with Elara.

2 Background

2.1 Construct of Elara

Elara samples Groth and Maller's SE-SNARK as an instance for its zk-SNARK, due to its efficiency in terms of proof's size and verifier's computation. We replace our Collision resistant hash function sha256 with Poseidon, as our cryptographic hash function. We also use a hybrid construction of Merkle tree and an incremental Merkle tree, as a substitute for our traditional Merkle trees. In our system most of the constraints originates from doing sha256 hashing from the leaf of a Merkle Tree to its root for its circuit operations of reddemrewards, transfer and shield/unshield, which is now reduced by replacing it with Poseidon hash function. In general, there are no candidate hash functions, such as MiMC e7r91 or Poseidon t6f8p57, that are low in gas costs and small in circuit sizes at the same time. Arithmetic circuits built with MiMC and Poseidon hash bear relatively smaller constraints compared to systems relying on other hash functions like sha256 or keccak, but it suffers from high gas costs. We select Poseidon as our function to build our system due to its compatibility with GF(p) objects and fewer constraints per message bit. We propose a new type of data structures for storing all our hashes and retrieving them efficiently for building our proofs and keeping our constraints low through a snark specific hash function to increase usability efficiency.

2.2 Related Privacy Work

Shashank Agrawal & Benedict Bunz came up with a Zether in the winter of 2019, where they proposed, confidential payment mechanism that takes an account-based approach similar to where they propose Σ-Bullets, which makes Bulletproofs more inter-operable with Sigma protocols. In their implementations, the cost of a Zether transaction is around 7.8m gas which is expensive and it does not even hide the receiver the recipient of a transaction, where even in the anonymous Zether with a complete anonymity set of 64, gas cost for a single transfer reaches almost 48.7m.

Michal Zajac and Antoine Rondelet proposed a zerocash construct on Ethereum, ZETH in the spring of 2019, allowing expressive functionalities like transferring notes. It also uses sha256 and is built with approximately 59,281 constraints for its arithmetic circuit, where the cost for verifying a zkSNARK proof of 2m gas.

We can observe that existing privacy protocols on smart contracts are not practical enough where users need to wait around over 50 seconds to generate a transaction and is inefficient due to the gas cost involved. We aim to bring the proof generation and less as possible and limit gas costs on a NIZK system to make it efficient on Ethereum.

3 Moderations

3.1 Efficient Data Structures

We use an authenticated data structure (ADS) to help compile proof of a set $Z = \{z_1, \ldots, z_n\}$, where with its help, we are allowed to build deterministic data structure to represent a set membership where the uniqueness of the set depends upon the order u in which updates are made.

To formally define a relation a verifiable data structure for set would be, Ø = (Crit, Prove, Update, Verify) is a tuple of four efficient algorithms:

- $L \leftarrow \text{Crit}(1^{\lambda}, Z)$ the initialization algorithm takes as input the security parameter and the set $Z = \{z_1, \ldots, Z_n\}$ where $Z_i \in \{0, 1\}*$, output $L \in \{0, 1\}\lambda$.
- $\pi \leftarrow \text{Prove}(i, z, Z)$ takes as input an element $z \in \{0, 1\}*$, $1 \le i \le n$, and set Z, outputs a proof that $z = z_i \in Z$.
- $0/1 \leftarrow \text{Verify}(i, z, L, \pi)$ takes as input $1 \le i \le n$, $z \in \{0, 1\}*$, $L \in \{0, 1\}\lambda$, and proof π, output 1 if $z = z_i \in Z$ and $y = \text{Init}(1\lambda, X)$. Otherwise, return 0.
- $L' \leftarrow \text{Update}(i, z, Z)$ takes as input $1 \le i \le n$, $z \in \{0, 1\}*$ and set Z, output $L' = \text{Crit}(1^{\lambda}, Z')$ where Z' is obtained by replace $z_i \in Z$ with z.

Through with our definition of authenticated data structures, we expect our ADS to be correct and secure and defer the formal definition to Boneh and Shoup.

A per terminology Merkle tree is a ADS, where each leaf is responsible for recording the hash value of a data blob, and the non-leaf node is tasked with recording the cryptographic hash of its child nodes. We use a hybrid construction, consisting of a variant Merkle tree and an alternation of the Merkle tree known as the Incremental Merkle Tree that only supports data insertion for our system.

We formally name our data structure to be *Sequential trees* which is made up of two individual trees adhering to Ø, namely Alphabranch and subMtree respectively. Sequential trees are used for formulating the Merkle Proofs in our system, where a leaf is not represented by the tree's root, but is signified by a stack of pointers, one from each level of the Alphabranch and for verifying a proof all the pointers of the Alphabranch are saved in the Merkle tree variant, subMtree, where the tree is divided into several subtrees and only the roots of these subtrees are saved. With this data structure, the complexity of the circuit proof only increases by a small factor, but it gives us the advantage of commitments to be inserted with only *O(1)* updates and get verified efficiently.

For a formal definition of the Alphabranch ý,
• Let structure height of ý be defined to be K,
• Representing pointers or Nodes in the Alphabranch are represented by K+1, one from each layer of ý. At each layer of the tree, the node chosen is either the only leaf or the rightmost leaf where it is the root of a perfect left tree.

Update: Leaf updates are made in the Alphabranch by inserting changes to a single leaf from a single level and only one node is modified that is closest to the leaf. It is achieved through switching to a new node or recalculating the hash value. Alongside while updating the Alphabranch, whenever a leaf is inserted, a node in the Alphabranch is computed and its corresponding subtree in the subMtree structure is recalculated along with its Merkle root. The subMtree is designed so that it can make the whole data structure bandwidth efficient and can help constrain public inputs in the zksnark circuit to 1. The subMtree uses a new form of Merkle tree where the tree is divided into several subtrees and the roots of these structures are saved. Through this global construction of sequential trees, we can observe an efficiency in hashing to the node and the system becomes more efficient as the gas cost for inserting data reduces drastically compared to the original Merkle tree, where complexity goes down from $0(\log_2(n))$ to 0(1). This hybrid construction works well because all the nodes are publicly recorded and their Merkle roots are fixed in the subMtree. Therefore, in Alphabranch instead of updating the root each time, we only update only one node every time a leaf is added along with only one change that we make to its corresponding subtree after calculating the node, in the subMtree. Therefore at an average, it would take

only one hash calculation in the Alphabranch for inserting each data into a leaf, instead of repeated hashing from the leaf of a Merkle tree to its root. This helps us avoid reading or modifying every node on the path from the leaf to the root node, so during the execution of the smart contract, every state read and write doesn't correspond to a large number of reads and writes of the data structure, which can have a positive impact on efficiency. For each non-leaf node in ý, the hash would be summed for a maximum of $2^k - 1$ times for the entire tree after all leaf nodes are inserted. When inserting to a left node in the Alphabranch, there are no has calculations, but while adding data to the right node, the depth of the computation equals to the selected subtree. Computational intensity for hashing depends on the position of the node, which increases when the node is closer to the right and the higher is its subtree selection. With a hight K, adding all the leaves in a traditional Merkle tree would require $2K*K$ calculations compared to $2^k - 1$ in an Alphabranch.

ALGORITHM : SEQUENTIAL ALGORITHM	
1	**initialize** alphabranch
2	index ← nxt index;
3	active index ← nxt index;
4	nxt index++;
5	**procedure** insert = new insert();
6	level hash ← insert();
7	**while** all leaf falls right() **do**
8	**for** i in levels **do**
9	**if** active index % 2 == 0 **then**
10	leaf left ← level hash;
11	leaf right ← zeros(i);
12	Node(i) ← level hash;
13	**break;**
14	**else**
15	leaf left ← node(i);
16	leaf right ← level hash;
17	**end if**
17	level hash ← hash(leaf left, leaf right);
18	active index /= 2;
19	**end for**
20	push();
21	**end while**
22	**end procedure**
23	**procedure** verify = new verify();
24	active index ← verify();
25	level hash ← verify();
26	**for** i in levels **do**
27	**if node** == 0 && node (i) == level hash **then**
28	verified();
29	**end if**
30	**if** active index % 2 ==0 **then**
31	leaf left ← level hash;
32	leaf right ← path(i);
33	**else**
34	leaf left ← path(i);

35	leaf right ← level hash;
36	**end if**
37	level hash ← hash(leaf left, leaf right);
38	active index /= 2;
39	**end for**
40	**end procedure**

Figure 1: ***Sequential Trees.*** *node(i) ascertains the node closest to the leaf and the stack gets initialized with empty nodes, when a new node is inserted it find its lowest left node as the selected pointer, and update its root. We define the serial number of the node on each layer through active_index., and we select the left active_index %2 == 0.*

Verify: Sequential trees are composed of a series of nodes in the Alphabranch, which are saved in the subMtree and these pointers can cover all the leaves that have been added to the data set. One of our main goals is to design a system that's bandwidth efficient, and we achieve that by reducing membership proof size. Our formal construction of is capable of dramatically reducing the gas cost for inserting data into the set and proving it in an bandwidth-efficient manner. In the Alphabranch the prover would need access to not only the path from the leaf in question to its nearest node but also towards all the nodes in the structure to verify if the nearest node maintains structural integrity in the system with respect to the leaf for being its closest. These nodes represents the uniqueness of the whole data structure and through with its help, we can prove if the data in question belongs to the set or not. This alone would have become bandwidth heavy but coordinating these nodes saved in the SubMtree, where the tree is divided into fragmented sets and where only the roots of these sets are saved helps minimise multiple data input while curating the proof and restricts 1. We utilise the path obtained from the subMtree instead of Alphabranch to prove data validity for any particular leaf, rather than calculating all the paths to the nodes from the leaf of Alphabranch.

We also modify our function in the system defined as follows, F(x,y) where x is the public input and y is the private input in the snark to F(H, f(x,y)$^{H(x)}$) to limit the size of all the public inputs for in the circuit to 1. The Poseidon hash is used as the hash function for all the input concatenation in the system. As a result, the construction allows for efficient data insertion and verification.

Elara consists of a handful of data structures such as the nullifier list, the deposit commitment tree and the reward tree. The ADS built for the nullifier lists are structures where the destination of a data is determined by the hash of the nullifier note data. This construction utilised resembles with that of a sparse Merkle tree and helps the system provide un-linkability between say the creation of a deposit commitment and a spent nullifier. The tree comes with 2^{256} theoretically-available leaves where each leaf initialises with the number 0. We prove if the location holds a zero or otherwise for efficiently proving if a withdraw or transfer operation in the system is valid or not.

3.2 Hashing Algorithm

Algorithm logic implementation can touch up to 1000s of gates depending on different circuits. That leaves hashing algorithm controlling most of the computation in a snark operated system. Elara uses a form of hybrid tree made up in the form of an incremental Merkle tree and Merkle tree, to accumulate identity commitments, where some in-system operation requires inserting and verifying data into the tree in a single update which put emphasis on selecting an efficient hashing algorithm with an updated data structure.

In a zkSNARK system we need to select between two trade-off, first between the tree capacity and gas costs for tree initialisation and insertions; and second between the circuit size and tree capacity. We overcome the factor of gas costs through our sequential trees for data insertion and we deal with the our circuit size through having a bandwidth-efficient construct for proving set membership Poseidon in order to lower down the number of constraints and minimise the amount of field multiplications to increase efficiency of usability.

A hash function is selected based on, the gas costs and proof generation time. Both the selected variables are often inversely related to each other, where a hash function with a lower gas costs leads to a greater circuit constraints, which leads towards a longer proof generation time and vice versa. Even if a SHA256 compression function is cheaper in the EVM, it consists of mostly Boolean operations and has to be performed on one bit per field element which leads to about 27000 constraints for a 512 bit message, where a single proof creation takes around 1.5 sec. This means around 52.7 constraints per bit processed, so it becomes highly inefficient in an arithmetic circuit over a large prime field. But now, with our newly updated ADS, we can have faster ECC operations and have the algebraic primitives to work on large prime fields. The highly optimised Pedersen hash function, provides 1.7 constraint per bit and reduces constraints down from 27000 to 896 for a 512 bit message. Pedersen hashes are known to be provably secure against discreet logarithmic assumption but has the threat of length-extension attack, pre-image vulnerability and homomorphism. It can be staged against a hash function that's in-differentiable from random oracle like MiMC hashes which requires gas cost up to 59840 to hash 2 value using the MiMC in a classical sponge construction. But, both gas costs and circuit constraints required to append a value using MiMC for an incremental Merkle tree increases linearly with its depth of operation. Each round has one multiplication and has a key or a construct that needs $n\log_3(2)$ steps to reach a maximum of second degree. It's not trivial to generalise it for a wider state.

Poseidon is an optimised hash function in the lines of HADESMiMC built for a snark based system. It is built to work on large prime fields through which we can avoid the conversion of the data into bits, in where field elements aggregates overheads per element in the input. Poseidon works natively with field elements reducing dependency up to only a few extra constraint. It combines full rounds and partial rounds to increase efficiency and operations are defined to occur in a field with which it is able to provide

a low arithmetic circuit complexity. A round in Poseidon is made up of three stage layers namely, *ARK(.), S-box* and *M(.),* where the *ARK* layer is an add round case and the *M* layer is the Matrix multiplication round where we operate on a constant and are both free in a SNARK. It uses x^{α} as its S-box, in bijection where with the help of partial rounds is able to remove a lot of invocation caused by the S-box. Poseidon is recorded to give 228 constraints in MNT4/6, with only around 0.3 constraints per bit processed and are able to work on field elements without any conversion. Staging PoseidonT3 against MiMC hash to build an incremental Merkle tree we can observe an efficiency in the former for path verification circuit constraints.

4 Conclusion

This paper provides an outlook toward making circuit operations more economical and efficient for Elara. The paper lends the reader a high level overview of the dependencies and the factors governing a cryptosystem's efficiency. It run over algorithms that minimises hashing cost for the software and helps reduce public input for any proof generation by proposing a two tree based data commitment scheme. It also goes over different hashing algorithm and describes why it selects Poseidon for our system construct running on EVM.

References

1. Ce Zhang , Cheng Xu , Jianliang Xu , Yuzhe Tang , Byron Choi. "GEM-Tree: A Gas-Efficient Structure for Authenticated Range Queries in Blockchain".

2. Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. "MiMC".

3. Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge, pages 90–108

4. Crosby, S.A., Wallach, D.S.: Authenticated dictionaries: Real-world costs and trade-offs. ACM TISSEC 14(2), 17:1–17:30 (2011)

5. Rasmus Dehlberg, Tobias Pulls and Roel Peeters. "Efficient Sparse Merkle Trees: Caching Strategies and Secure Non-Membership Proofs"

6. John Kuszmaul. "Verkle Trees".