A Private Verifiable Sparse Merkle Tree

Daniel Kales[1], and Roman Walch[1]
[1] TACEO
papers@taceo.io

Abstract. In this writeup we propose an oblivious and verifiable sparse Merkle tree construction that can be used to implement a private and provable hashmap which in turn can be used to build a privacy-preserving version of Solidity's mapping primitive where data from multiple data owners is stored privately. Our construction is based on MPC, coSNARKs and MPC-ified Cuckoo tables built on ORAMs. Using DuORAM, the communication complexity and number of rounds are only logarithmic in the actually present elements in the sparse Merkle tree. Thus, our current construction is mainly bottlenecked by the CPU load caused by generating distributed point functions used in DuORAM read and write operations. While we show with benchmarks that the performance of our single-threaded prototype is already reasonable for smaller maps with 100k entries, multithreading and potentially GPU acceleration have potential to drastically improve performance.

Keywords: Oblivious Map · Oblivious Sparse Merkle Tree · MPC · CoSNARKs

Contents

1 Introduction

Privacy-preserving blockchains which allow to execute smart contracts on private state, i.e., state only known to a user which is only represented as a commitment on chain, have become more prevalent and viable in the last years. However, what is still missing from these chains is a way for multiple data owners to collaborate on their joint dataset without leaking their private data to each other. Similarly, a privacy preserving hashmap, i.e., a private counterpart to Solidity's mapping primitive `mapping(_KeyType => _ValueType)`, which can be filled with data from multiple data owners without the need of revealing anything, has not yet been proposed in the literature. For such a primitive to be realized, we need to make sure that both the key and value stay private from any single party, it can be represented as a succinct commitment on chain, and it should still be possible to prove (with zero knowledge) membership and non-membership for specific `key-value` pairs, and that updating the map was done correctly.

In this document we design such a private and verifiable mapping primitive using secure multiparty computation (MPC) and collaborative SNARKs (coSNARKs) [OB22]. Our construction is based on a sparse Merkle tree (SMT) which, in contrast to traditional Merkle trees, allows for leaves in the tree to be missing. These missing elements are conceptually replaced by a special dummy value and instead of keeping the whole tree in memory, only present leaves and their indices are stored. Consequently, even for trees with depth 256 we do not need to store 2^{256} items, but only as many as are actually present in the tree. This data structure, in addition to allowing efficient membership proofs like a standard Merkle tree, also allows efficient non-membership proofs by proving that a leaf is the default value.

The underlying storage of a SMT is essentially a mapping of an index to a value for the present leaves, from which the full Merkle tree can be computed, substituting in the default value whenever required. A common optimization is to also have a mapping for the inner layers of the tree, such that those elements do not need to be recomputed on the fly. These inner mappings also map an index to a present node in the tree, however, the index datatype reduces by 1 bit for each layer when moving up the tree. Again, for the inner layers, if both children are not present, it is replaced by a hash of two default values, which again can be seen as a separate default value on its own, which does not need to be stored.

The sparseness of the data structure brings up some issues when translating it to a privacy-preserving version using MPC. A plain operation on a SMT would take an index (representing the path through the tree, essentially of type $\mathbb{Z}_{2^k}$ for a SMT of depth k) and a value (which can be of arbitrary type). For privacy, we want to enable operations that take a secret shared key and a secret-shared value instead (denoted by $[x]$). However, to support the sparseness, we need an efficient oblivious data structure to handle the mapping. While the mapping could be replaced by a simple list of ([key], [value]) pairs that are queried with a simple linear scan, this approach introduces a lot of communication and computational overhead, as performing the comparison of the query key with each element in the list is quite expensive in MPC.

Therefore, to store the key-value mappings for each layer in the SMT, we design an oblivious map primitive which is internally backed by a Cuckoo hash table, where each of the individual storage tables are again backed by a distributed ORAM. The design of the overall primitive is shown in Section 4.

The final aspect of the private SMT primitive is its verifiability. In a traditional SMT, one can easily generate traditional inclusion proofs against a Merkle tree state root. However, while we can retrieve the relevant nodes using our oblivious maps, revealing the final inclusion proof also reveals the index of the element, which violates the privacy. This is traditionally fixed using zero-knowledge proofs, however, in our MPC system there should also not be a single party that has access to the Merkle tree path, as it would leak the index to that party. Therefore, we also compute the zero-knowledge proof in MPC using collaborative SNARKs.

1.1 Goals

In the following, we describe the goals we want to achieve with this design:

- The map should allow for efficient inclusion and update proofs.
- For performance reasons, we target a depth of 40 layers in the SMT for now, allowing keys to be represented by 40 bits.
- In all but the last level, we store hash values produced by a ZK-friendly hash function (which are field elements $\in \mathbb{F}_p$), so the oblivious map should be tailored to that.
- In many MPC applications, network communication and round complexity becomes the bottleneck. Thus, we aim to make network overhead reasonable, such that the system can theoretically run in a cross-cloud setup.

To keep performance reasonable we specifically allow the leakage of some metadata: 1) We allow to leak the type of request (read, update, insert). 2) Internal cuckoo building blocks leak their actual stash size. Traditionally this would leak some information about the set of items in the cuckoo hash tables, however, since we evaluate secret-shared hash functions in MPC and never

reveal their outputs, leakage is harder to exploit. One could observe that a given insert in the cuckoo table produced an internal hash function collision, but no knowledge about the hash function inputs, the hash functions themselves or the hash function outputs is ever revealed.

2 Background

2.1 MPC

In this work we will use secure multiparty computation (MPC) and collaborative SNARKs to build a private verifiable sparse Merkle tree. MPC in general allows multiple mutually untrusted parties to compute functions on their private inputs without revealing any information about them to each other. In this work we will use a flavour of secret-sharing based MPC [Dam+12, Sha79], namely 3-party replicated secret sharing [MR18]. In this protocol, secret data is split amongst 3 parties such that the data x can be reconstructed as $x = x_0 + x_1 + x_2$ where all x_i are random. Then, each party gets two from this 3 so-called additive shares and have a replicated share (x_i, x_j) where $j = i + 2 \bmod 3$. Having this replicated share does not leak any information about x, but two parties can combine their shares to reconstruct x. Such a scheme is called 2-out-of-3 secret sharing and requires an `honest majority` among the MPC parties for x to remain private.

The parties can jointly compute functions on x by computing on their shares. Linear operations, such as adding shares, thereby, can be computed without the need of communication among the parties, whereas non-linear operations such as multiplications require them to exchange some form of randomized information . Consequently, the number of non-linear operations directly impacts the performance of MPC protocols and should ideally be minimized.

In this work we focus on the `semi-honest` security model, which is only considered secure as long as the parties follow the protocol honestly. However, since we build a verifiable protocol by using coSNARKs (see next section), parties can not produce faulty results by deviating from the protocol – only privacy will be lost when enough parties collude with their shares.

2.2 Collaborative SNARKs

Zero knowledge (ZK) proofs allow a party (the prover) to prove the validity of statements to another party (the verifier) without leaking anything beyond what can be implied from the statement itself. While the statement, in general, can be any NP statement, in our case it will be among the lines of "The data returned is part of the data structure at the specified index for which I know commitments".

In our case, the data structure will only be held in secret-shared form by a set of MPC-parties. Thus, alongside the actual access to the data structure, the ZK proof also needs to be computed in MPC. This is generally referred to as computing a collaborative SNARK [OB22] (coSNARK). In this work we will consider a coSNARK version of the well known Groth16 [Gro16] proof system.

2.3 Notation

Throughout this paper we denote by $[x]$ that the value x is secret-shared among the MPC parties.

3 Related Work

3.1 Distributed ORAM

Roughly speaking, distributed ORAMs (DORAM) are built using two approaches: The first approach is based on distributed point functions (DPFs) or function secret sharing (FSS) and is very communication efficient (especially in terms of communication rounds), but requires linear computation (essentially a linear scan over the full secret-shared RAM). The first paper championing this approach is `Floram` [DS17], with `DuORAM` [VHG23] being a more recent work in that line.

The second approach is building distributed ORAM using more classical hierarchical [GO96] or tree-based [Ste+13] ORAM techniques, which require only sublinear computation, but usually require more total communication in terms of both total data sent and total communication rounds. Jarecki and Wei [JW18] presented a 3-party version of Circuit ORAM [WCS15], and a more recent variant of such a design is presented in `GigaDORAM` [Fal+23], which tries to optimize the round complexity of hierarchical ORAMs further and is targeting the high-performance datacenter setting. As an alternative to ORAMs with (poly-)logarithmic complexity, `Ramen` [Bra+23] is a work that builds distributed ORAM with square-root complexity from traditional square-root ORAM approaches [GO96].

3.2 Oblivious Data Structures

A natural application of ORAM and distributed ORAM protocols is to serve as the underlying storage for various traditional data structures, such as stacks, heaps, trees and maps. For traditional ORAM protocols, Wang et al. [WCS15] show that specialized protocols can outperform a generic ORAM instantiation for some specific data structures and give recipes for many oblivious data structures, including an oblivious hash table (Sec C.7) which we base our cuckoo hash table on. For the distributed ORAM setting PRAC [SVG24] gives a comprehensive study of various different oblivious data structures backed by DPF-powered DORAM. In particular, they focus on efficient binary search, oblivious heaps and AVL trees. Cao et al. [Cao+24] explore the efficiency of oblivious maps that are internally build by combining tree and hash-table based approaches.

3.3 Distributed Point Functions (DPFs)

One of the main building blocks for the communication efficient flavours of ORAMs are distributed point functions (DPFs), which is a flavour of function secret sharing (FSS). The general idea of function secret sharing was introduced by Boyle, Gilboa and Ishai [BGI15], and the same authors presented an efficient tree-based 2-party DPF with logarithmic complexity in [BGI16]. Recently, the Half-tree paper [Guo+23] showed how to reduce the work to generate DPFs, which is mostly concentrated in the expansion of a GGM tree, by utilizing pseudorandom correlated GGM trees instead.

Of increased relevancy for distributed ORAM protocols are protocols to generate DPF keys collaboratively in MPC protocols. While a trivial execution of the DPF.KeyGen functionality in MPC would achieve this, Doerner and shelat [DS17] show how to perform DPF key generation in MPC without having to evaluate the PRG in MPC and the authors of the Half-tree paper also apply their optimizations to this variant. Xing et al. [Xin+25] show how to more efficiently perform DPF key generation in MPC if the output needs to be an arithmetic secret share.

Most applications of DPFs focus on the (2,2)-DPF setting, where a DPF is split into two key shares with security against 1 corruption. For distributed ORAM use-cases (2,3)-DPFs (3 key shares, with security against 1 corruption) are especially useful, since they can enable both private reads and writes at the same time because the resulting secret sharing allows one multiplicative homomorphism, while (2,2)-DPFs need to work around these issues (as seen in [VHG23]). Various constructions for (2,3)-DPFs are given in [Bun+20],[BKO22] and [ZYP24].

A different line of work is related to the verifiability of DPFs. Boyle et al [BGI16] also give a verifiable DPF construction and [Bon+21] show how to make it secure against malicious adversaries for DPFs with binary outputs. de Castro and Polychroniadou [CP22] generalize this malicious verifiability check and further improve its efficiency.

4 Verifiable Private Sparse Merkle Tree

In this section, we describe our design on different levels of abstraction.

4.1 Level 0: Sparse Merkle Tree

At the highest level of the design, we have a sparse Merkle tree (SMT), which is a Merkle tree where non-existent leaves are represented by a default value (e.g., zero). The SMT allows us to efficiently verify the inclusion and non-inclusion of a key-value pair in the tree. Since we need the SMT to be ZK-friendly, we use the `Poseidon2` [GKS23] hash function to compute the nodes in the upper layers.

The main high level operations we want to support are:

Operation	Comment	Output
init(default_value, max_depth)	Initialize the SMT with a given default value.	Self
get_root_hash()	Retrieve the root hash of the SMT, which is a commitment to the entire tree.	root_hash
get([key])	Retrieve the value associated with a secret-shared key, or a secret-shared default value. Optionally, also retrieve an inclusion proof for the key.	[value], inclusion_proof
insert([key], [value])	Insert a secret-shared key-value pair into the SMT.	insert_proof, new_root
remove([key])	Remove a secret-shared key-value pair from the SMT, inserting a secret-shared default value.	remove_proof, new_root
update([key], f)	Update the value associated with a secret-shared key using a function f, which takes the current value and returns the new value. Both insert and remove can be seen as special cases of update, where the function is either returning the new value or a function that returns the default value.	update_proof, new_root

For read operations, we want to support efficiently retrieving the associated Merkle tree path from our mapping primitive. To do this, we want a sparse data structure to store the non-default leaves of the Merkle tree at each level. This allows us to efficiently retrieve the path from the root to the leaf, as well as the leaf itself. For write operations, we need to retrieve the path to the leaf, while also updating the path up to the new root. This means that our data structures needs to support efficient updates as well.

The following describes the basic approach to implementing our SMT primitive:

SMT Operations:

- `init(default_value, max_depth)`: For each level of the Merkle tree i, call $m_i \leftarrow \text{OMap.init}(2^i, \text{default\_value})$. The root m_0 is kept public, and initialize to the root corresponding to the default value.
- `get_root_hash()`: Return m_0.
- `get([key])`: The parties together:
  1. Bit decompose the key into its binary representation $k_\ell, ..., k_0$.
  2. For each level i from 0 to `max_depth`, retrieve the value v_i from the oblivious map m_i using the key truncated to the first i bits, while negating the last bit: $v_i = m_i.\text{get}((k_\ell, ..., k(\ell - i) \oplus 1))$.
  3. Return the value $v_{\text{max\_depth}}$, as well as the inclusion proof, which is calculated as a coSNARK of the Merkle tree inclusion proof for the key, using the values $m_0, v_0, ..., v_{\text{max\_depth}}$.
- `insert([key], [value])`: The parties together:
  1. Bit decompose the key into its binary representation $k_\ell, ..., k_0$.
  2. Perform Step 2 of `get` to retrieve the values v_i from the oblivious maps m_i.
  3. Based on the old Merkle tree inclusion proof, compute the new values $v'_i \leftarrow H(v_{i+1}, v'_{i+1})$ for each level i from `max_depth - 1` down to 0, where H is the hash function used in the Merkle tree. The value of $v'_{\text{max\_depth}}$ is set to the new value.
  4. For each level i from 0 to `max_depth`, insert the value v'_i into the oblivious map m_i using the key truncated to the first i bits: $m_i.\text{insert}((k_\ell, ..., k(\ell - i)), v_i)$.
  5. Update the root m_0 to reflect the new value v'_0.
  6. Compute a coSNARK of the following statement: The old root m_0 is the root of the Merkle tree where an element with index key has the Merkle tree path $v_0, v_1, ..., v_{\text{max\_depth}}$, and the new root is the root of the Merkle tree where an element with index key has the same Merkle tree path. This ensures that only the element at position key could have been updated.

4.2 Level 1: Oblivious Map for Sparse Storage of Merkle Tree Nodes

We use an oblivious map to store the non-default nodes of the Merkle tree. A separate map is used for each layer in the Merkle tree, where the keys are the path in the Merkle tree truncated to the current level. For the first few levels, a simple linear scan variant of this primitive is sufficient since there will only be a small number of elements present, but for the higher levels, we want a more efficient data structure.

We want to build an oblivious map that supports the following operations:

Operation	Comment	Output
init(capacity, default_value)	Initialize the oblivious map with a given capacity and default value.	Self
insert([key], [value])	Insert a secret-shared key-value pair into the map. W.l.o.g, we assume key types for this map to be of the form $\mathbb{F}_{2^k}$ and value types of the form $\mathbb{F}_{2^{256}}$	
get([key])	Retrieve the value associated with a secret-shared key, or a secret-shared default value.	[value]
remove([key])	Remove a secret-shared key-value pair from the map, inserting a secret-shared default value.	

While we want to keep both the keys and values private (i.e., secret shared) including no leakage on their values by observing access patterns, we explicitly allow to leak the type of operation (insert, get, remove) and the size of the map (i.e., the number of elements in the map).

4.3 Level 2: Cuckoo Hash Table for Key-Value Pairs

The oblivious map primitive performs the actual `key => value` mapping which we implemented using an MPC-adapted version of a Cuckoo hash table. Informally speaking, the Cuckoo table translates the access type to `index => value` by hashing the `key`. Since hashing keys can lead to colliding indices, especially for small table sizes, the Cuckoo hash table is usually build from multiple RAM primitives, with a random hash function each. Elements are then inserted in the first RAM for which the hashed index is empty. Furthermore, a stash is used to accommodate elements which do not fit in any RAM. While the stash, which is read and written to via linear scan, is allowed to grow indefinitely, it is crucial for performance to parameterize the Cuckoo table correctly (e.g., see [Yeo22] for a detailed discussion) such that the stash size remains reasonably small (e.g., logarithmic in the input table size).

Furthermore, when too many elements are inserted in the Cuckoo table a rebalance step is required during which the table size is adapted (e.g., doubled in size). Thereby, new hash functions are sampled and all elements are reinserted into the tables.

To achieve our goals we implement an MPC-adapted version of a Cuckoo table. Thus, the RAM blocks will be implemented as Oblivious RAM primitives (see next section) and we use `LowMC` [Alb+15], an MPC-friendly block cipher, as hash function using a fixed randomly sampled secret-shared key. Since the key is secret-shared, no party can compute hashes on their own.

In the following subsections, we give more details on the employed algorithms in our MPC-version of the Cuckoo hash table.

4.3.1 Data Structure

We will construct a Cuckoo table from two RAM blocks to keep the number of ORAMs accesses at a minimum. Furthermore, for a data structure of size n the ORAMs will be initialized with a size of $\kappa \cdot n$ where $\kappa = 2$ to reduce the likelihood of collisions. The ORAMs themselves store both, the secret-shared key and the secret-shared value, alongside a secret-shared occupied bit indicating whether the slot has been written to at least once.

4.3.2 Read

To read a value, we read from both ORAMs to get the stored key-value pairs. Then, we perform a linear scan over the stash alongside the results of the two ORAM reads to create a vector of secret-shared bits, indicating if and where the element has been found. The bits corresponding to the ORAM values have to be combined with the ORAMs occupied bits to prevent false reads on default key values. The result is a vector $\vec{x}$ where at most one bit is set. Calculating a dot product over the stash (including the read ORAM elements) with $\vec{x}$ gives the correct read result or zero. Adding the product of the default value with $\left(1 - \sum_i x_i\right)$ sets the result to be the default value if the element was not found.

4.3.3 Write

Since many ORAM implementation only allow to update values instead of overwriting them, we first have to read the element from the Cuckoo table. Thereby, we also get the secret-shared bitvector $\vec{x}$ which indicates the position of the read element if it has been found. Thus, $\sum_i x_i$ is a bit which indicates whether the element was already present in the Cuckoo table or not. In this implementation we allow leaking this bit (which we currently think is ok because the MPC-parties only learn whether the unknown position produced by a secret-shared hash function was written to before or not) to decide whether we have to update the value or insert a new one.

4.3.3.1 Update

Since we already have the secret-shared bitvector $\vec{x}$ indicating where the element has been found, we can update the stash using a CMUX over the whole stash data structure, requiring communication linear in the stash size. For the ORAMs, we CMUX the update value to be 0 if the corresponding bit is 0, else to the new_value xor the read_value and call `ORAM.Update`.

4.3.3.2 Insert

To insert the pair (k, v), we perform the following algorithm from [ANS09]. First, the stash is organized as a queue and we insert (k, v) at the back of the queue, alongside a indicator value which memorizes in which ORAM we will try to insert this pair later. In our case of two ORAMs, a bit is enough which we set to 0 for new key-value pairs. Thus, the stash is contained of the key-value-bit tuples $[(k_0, v_0, b_0), (k_1, v_1, b_1), ..., (k_n, v_n, b_n)]$, where we define (k_0, v_0, b_0) as the head and (k_n, v_n, b_n) as the back. Then we iterate the following algorithm L times:

- We take the head of the queue (k, v, b) and swap it with the element at index $h_b(k)$ in the ORAM indicated by b to get (k', v', o'), where o' is the occupied bit. Thereby, $h_{b(k)}$ is the hash function used in the ORAM indicated by b.

- We open the occupied bit o'. If it was true, we put $(k', v', b' = 1 - b)$ at the back of the stash. This only leaks that some two unknown elements have produced a hash collision for an unknown hash functions.

We want to note that [ANS09] puts the read element at the head of the queue and performs a periodic cycle detection algorithm to check whether the head can actually be inserted in any ORAM or not. In case a cycle is detected, the head is deferred to the back of the queue. To avoid this cycle detection algorithm we directly insert the read element at the back of the queue.

4.3.4 Reducing Stash size

Once the stash becomes too large, we rebuild the Cuckoo hash table with larger ORAMs and start over. Note, new hash functions (i.e., new `LowMC` keys) need to be sampled. Our approach is simply opening the occupied bits in both ORAMs and take all secret-shared values corresponding to set bits. For security, we shuffle all elements of both ORAMs before opening the occupied bit. Based on the opened bits, we can hash the stash and ORAM elements and try to insert them into the new Cuckoo table. Since we use an unknown cryptographic hash functions (remember the encryption keys are secret-shared), we might perform the ORAM insertions based on opened hash values. This would improve performance significantly at the cost of leaking the hashes of the keys in the map. Nonetheless, this rebalancing step is only called rarely, thus its performance overhead does not severely affect overall performance.

4.4 Level 3: Distributed ORAM

The underlying ORAM construction is DuORAM [VHG23], a three party ORAM construction that uses distributed point functions (DPFs) to achieve logarithmic communication complexity. The main operation is the generation of DPF keys in MPC, which are then evaluated locally by the parties to get a secret-shared one-hot encoded vector. While the evaluation of DPFs requires $O(n)$ operations, these are cheap, local operations (evaluations of a pseudo random number generator (PRNG)), and happen outside of the MPC protocol. This vector is then used to read from the ORAM, by doing a full dot product with the secret-shared data in the ORAM. Writing to the ORAM is done by first reading the value, and then updating the specific location with the difference to the new value. DPF evaluations are also used for the write part to keep communication complexity logarithmic.

The interface of the ORAM is as follows:

Operation	Comment	Output
init(size, default_value)	Initialize the ORAM with a given size, and the given default value. The size is fixed to be a power of 2^k, and implicitly defines the secret-shared index type $\mathbb{F}_{2^k}$	Self
read([idx])	Read the value at the given secret-shared index.	[value]
insert([idx], [value])	Insert a secret-shared value at the given secret-shared index.	

In its basic form, DuORAM leaks the access type (read, insert) and the size of the ORAM but nothing else, which already fits the privacy goals outlined in Section 1.1.

Due to using DPFs, DuORAM hash a logarithmic (in the set size) communication complexity, a logarithmic number of communication rounds, and linear computational complexity (i.e., expanding the DPF to the size of the ORAM plus the dot product with the database). Furthermore, it comes with the nice feature that the DPFs can be computed in an input-independent offline phase, thus online performance is reduced to communication complexity and rounds which are $O(1)$.

4.5 Level 4: DPF

All operations in the DuORAM constructions require various DPF keys. In general, there are two strategies to produce the DPFs required: (i) having a dealer party distribute them to the evaluating parties or (ii) collaboratively generate them in MPC. The standard method to do (ii) was introduced in [DS17], and manages to perform all of the evaluations of the PRGs outside of the MPC protocol, which greatly speeds up the DPF key generation. However, a slight drawback is that during the DPF key generation, the participating parties are already required to perform a full-domain DPF expansion as well. In many protocols this is not a real limitation since the expanded DPFs are needed anyways, however, if one wants to generate these DPFs in a preprocessing phase, either the fully expanded output needs to be kept in memory, or it needs to be discarded and later on regenerated from the DPF keys, duplicating some work.

A recent optimization for DPFs is presented in [Guo+23], where the authors show that for all but the last level in the DPF tree construction, one can use a so-called correlated GGM tree, where the two leaves of a parent s are computed as $H(s)$ and $s - H(s)$, respectively. This reduces the number of PRG evaluations and also the round complexity of the interactive MPC protocol used to generate the DPFs.

Another common optimization arises when the output datatype of the DPF is $\mathbb{F}_2$, as it is for ORAM reads where these bits are used to compute a dot-product against the database. Traditionally, last level seeds in the DPF tree are converted into the required output type by using the seed as an input to a PRG that produces an element of the required type. If the output type is smaller than the seed size κ, this can be skipped and the bits of the seed can be taken directly. However, for $\mathbb{F}_2$, this still wastes 127 bits of the output. To alleviate this, we

can pack 128 bits of output into a single $\mathbb{F}_{2^{128}}$ item, reducing the number of output element by a factor of 128, and therefore the tree depth by 7. In turn, this also reduces the number of internal PRG calls for expansion by a factor of 128. The only change that is required is that the chosen output value β of the DPF is now a 128-bit string with a 1 at the position specified by the index.

5 High-Level Use Cases

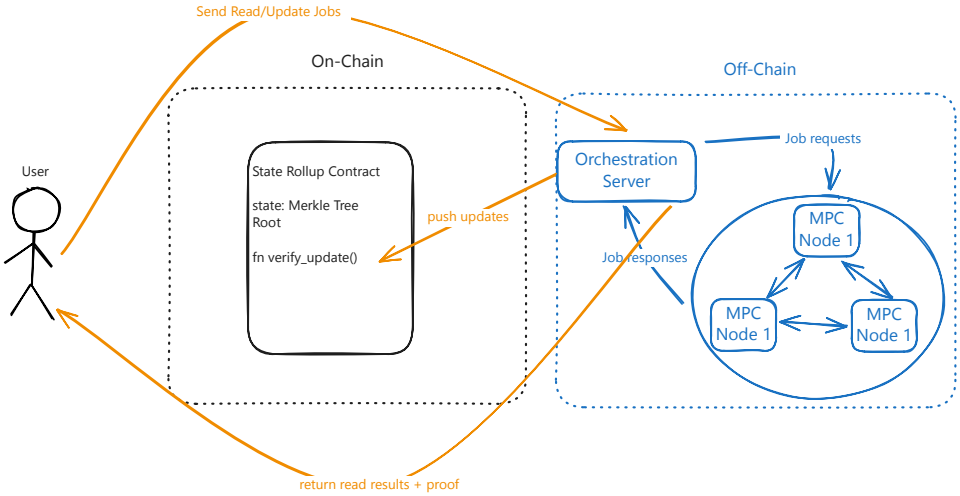

Fig. 2. A basic overview of the on- and off-chain interaction.

5.1 Sparse Merkle Tree Interaction with Smart Contracts

As discussed earlier, we use the sparse Merkle tree to implement a private and verifiable key-value storage for private data associated with multiple parties for smart contracts, i.e., a private counterpart to solidity's `mapping(_KeyType => _ValueType)` type. Whenever we use such a mapping in a smart contract, the contract itself will store the commitment (i.e., the root hash) of the sparse Merkle tree, while the actual data is stored (in secret-shared form) on the MPC network. A basic visual overview of the architecture is given in Fig. 2.

5.1.1 Request Inputs

Whenever a request (read, write, etc.) is made to the sparse Merkle tree, the MPC-parties need to get their secret shares of the private inputs (keys, values). Thus, there needs to be a way to communicate these to the MPC network. In general, there exist multiple ways to achieve this

data transfer: 1) The data holder directly communicates the shares to the MPC network; 2) The data holder stores the shares (encrypted with the recipients public key) on chain (leaking itself to be a user of a specific map). A second issue is telling the MPC network to process a request. This can again be implemented by a direct communication between the client and the MPC network, or the MPC network checks on-chain transaction whether new (encrypted) secret shares have been provided. Finally, the inputs themselves (key, value) should also be represented as commitment on-chain to bind the inputs to the actually generated ZK proof such that users get assurance that the correct inputs where actually used.

5.1.2 Request Outputs

As a result, the MPC network produces an output (e.g., the read value), and potentially an update to the sparse Merkle tree for write requests. The Merkle tree root stored in the smart contract should only be updatable if the provided ZK proof is valid. Furthermore, the smart contract can also specify that only a specific private key (e.g., the coordinator of the MPC network) can update the root. For user outputs, again multiple ways exist to communicate them with the user: 1) direct communication; 2) stored encrypted on chain (a proof of valid encryption should be part of the produced ZK proof). In the latter case, the output can be stored in a normal solidity mapping which maps an identifier (generated by the client creating an input) to the encrypted output. Furthermore, in the latter case the encrypted output also can serve as a commitment to the data, assuring the recipient that it will receive a correct value.

5.1.3 Payments

Payments can be realized by users putting tokens in escrow alongside their (encrypted) secret shares of requests. Then, the smart contract – on valid output delivery on chain – releases the tokens in escrow to the MPC providers. For user safety, the tokens should be released after a pre-defined time to the client, such that a client has no loss in case of the MPC network not fulfilling the request.

5.1.4 Epochs

The system by nature allows multiple parallel reads, but the Merkle-root changes for each write request.

For processing requests, this should not really be an issue, since read requests can be processed until a write request is encountered and the corresponding proofs can be verified on or off chain without issue. However, it might still be beneficial to store the root hashes in a mapping `epoch_id => hash` to keep old root hashes in the smart contract for external verifiability. Then, each produced proof should also indicate the epoch_id.

On the same note, the MPC network should also store – on a write operation – the previous state at least until the update proof can be verified on chain to prevent faulty state transitions. Additionally the MPC nodes could/should verify the update proofs themselves.

6 Benchmarks

In this section we discuss some early benchmarks of the oblivious map primitive. We use 3-party replicated secret sharing [MR18] as our MPC protocol and use the MPC-variant of the Groth16 [Gro16] proof system from TACEO co-SNARKs[1].

6.1 Map Access Runtime and Communication

In this section we measure the runtime of the map access operations for different set sizes (i.e., number of non-dummy elements stored in the leaves of the SMT) and different Merkle tree depths. All benchmarks are performed in a co-located setup where each party is an AWS m7a.8xlarge instance. Benchmarks are single-threaded and are using the MPC version of DPF key generation (i.e., there is no dealer party generating the DPF keys).

Table 4 shows the runtime of a read operation, an insert operation (i.e., writing a new key to the map), and an update operation (writing an already inserted key to the map). Furthermore, we give the (maximum) communication per party for the same benchmarks in Table 5.

Table 4. Runtime in `ms` for a read, insert, and update operation.

	Set Size		
Depth	10 000	100 000	1 000 000
Read [ms]:			
20	52.3	260.3	555.6
40	126.1	913.1	5700.7
60	216.5	2334.6	10885.8
Insert [ms]:			
20	279.5	1279.1	3514.6
40	1067.5	6303.0	55330.5
60	1771.8	11330.4	112866.7
Update [ms]:			
20	261.0	880.2	1646.9
40	756.3	4217.9	33885.2
60	1216.7	7684.5	64881.6

[1] https://github.com/TaceoLabs/co-snarks

Table 5. Communication in `MB` for a read, insert, and update operation.

	Set Size		
Depth	10 000	100 000	1 000 000
Read [`MB`]:			
20	0.1	0.1	0.4
40	0.2	0.3	0.4
60	0.4	0.4	0.5
Insert [`MB`]:			
20	0.5	0,5	1.2
40	1.1	1.1	1.4
60	1.6	1.7	2.1
Update [`MB`]:			
20	0.4	0.5	1.2
40	1.0	1.0	1.3
60	1.5	1.6	1.9

Table 5 clearly shows the logarithmic communication complexity of the used ORAM implementation, which also indicates that the overall runtime is heavily bottlenecked by CPU. Further inspecting our code shows that most runtime is spent in the expanding of the DPFs and the final dot-products between the ORAM databases and the expanded DPFs. Multithreading and potentially using GPUs might thus drastically improve performance of our current oblivious SMT prototype.

6.2 CoSNARK Generation

We further benchmark adding a coSNARK zero knowledge proof to the Map operations. These benchmarks were performed in a co-located setup where each party is an AWS m7a.4xlarge instance. As is usually the case for coSNARKs, we split the runtime in `witness extension`, i.e., generating the trace (in this case the R1CS witness), and the actual `proof generation`, i.e, in our case the Groth16 proof generation from the trace. Benchmarks for a SMT depth of 40 layers are shown in Table 6.

The read proof consists of a Merkle tree inclusion proof (i.e., d consecutive `Poseidon2` hashes for a SMT of depth d, d range checks for the path bits, and d multiplication for determining the order of the two input leaves per `Poseidon2` hash), and verifying the commitment to the input key. Since we use a hash-based commitment, this step consists of another `Poseidon2` instance. We want to note that in a real deployment, depending on the concrete setup, more commitments

might be needed (e.g., committing to the output). This would only slightly increase the number of `Poseidon2` hashes, thus requiring comparably small additional runtime and network overhead.

For write, the statement to proof essentially doubles since we compute two Merkle-tree inclusion proofs for the same path (one for the old element with the old root and one for the written element with the new root), and we verify two commitments, one for the input key and one for the input value.

Further, we want to note that the proof generation is independent to the set size of the SMT and only scales with its depth.

Table 6. Runtime in `ms` of extending the map access operation to a ZK proof using coSNARKs for an SMT depth of 40 layers.

Operation	Witness Extension	CoGroth16
Read	163.5	60.1
Write	69.5	107.1

Inspecting Table 6, one can observe that the coSNARK generation is quite fast with 60 `ms` for a read and roughly double that for a write operation. Witness extension is also quite fast with 160 `ms` for the read operation and the even faster 70 `ms` for a write operation. While on first glance this looks like a mistake, it can be explained. During the read operation (without trace computation) no `Poseidon2` hash has to be computed. Thus, trace generation adds the substantial number of rounds for the 40 consecutive `Poseidon2` hashes. This is different in the write operation: Since we already have to compute 40 `Poseidon2` hashes for generating the new values for the SMT path, we can parallelize these `Poseidon2` hashes with the ones required for trace generation. Thus, no communication round is added reducing the runtime overhead of the witness extension.

7 Conclusion

In this writeup we propose techniques to build a private verifiable version of a hashmap which can be used to bring Solidity's mapping primitive to private blockchains while allowing it to hold private data from multiple parties without the need of revealing them to anyone. Our prototype is build using a sparse Merkle tree where each layer is an oblivious Cuckoo table which in turn is build from ORAMs. Using DuORAM, contrary to most MPC applications, network overhead is comparably small and the overall performance is heavily CPU bottlenecked, which is confirmed by first benchmarks of our (single-threaded) prototype. However, we conjecture that runtime can be drastically reduced by employing multithreading and potentially GPU acceleration. Thus, in future work our main focus will be optimizing the DPF generation algorithms which are repsonsible for the heavy CPU load.

References

[OB22] A. Ozdemir and D. Boneh, "Experimenting with Collaborative zk-SNARKs: Zero-Knowledge Proofs for Distributed Secrets," in *USENIX Security Symposium*, USENIX Association, 2022, pp. 4291–4308.

[Sha79] A. Shamir, "How to Share a Secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[Dam+12] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty Computation from Somewhat Homomorphic Encryption," in *CRYPTO*, in Lecture Notes in Computer Science, vol. 7417. Springer, 2012, pp. 643–662.

[MR18] P. Mohassel and P. Rindal, "ABY3: A Mixed Protocol Framework for Machine Learning," in *CCS*, ACM, 2018, pp. 35–52.

[Gro16] J. Groth, "On the Size of Pairing-Based Non-interactive Arguments," in *EUROCRYPT (2)*, in Lecture Notes in Computer Science, vol. 9666. Springer, 2016, pp. 305–326.

[DS17] J. Doerner and A. Shelat, "Scaling ORAM for Secure Computation," in *CCS*, ACM, 2017, pp. 523–535.

[VHG23] A. Vadapalli, R. Henry, and I. Goldberg, "Duoram: A Bandwidth-Efficient Distributed ORAM for 2- and 3-Party Computation," in *USENIX Security Symposium*, USENIX Association, 2023, pp. 3907–3924.

[GO96] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[Ste+13] E. Stefanov *et al.*, "Path ORAM: an extremely simple oblivious RAM protocol," in *CCS*, ACM, 2013, pp. 299–310.

[JW18] S. Jarecki and B. Wei, "3PC ORAM with Low Latency, Low Bandwidth, and Fast Batch Retrieval," in *ACNS*, in Lecture Notes in Computer Science, vol. 10892. Springer, 2018, pp. 360–378.

[WCS15] X. Wang, T.-H. H. Chan, and E. Shi, "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound," in *CCS*, ACM, 2015, pp. 850–861.

[Fal+23] B. H. Falk, R. Ostrovsky, M. Shtepel, and J. Zhang, "GigaDORAM: Breaking the Billion Address Barrier," in *USENIX Security Symposium*, USENIX Association, 2023, pp. 3871–3888.

[Bra+23] L. Braun, M. Pancholi, R. Rachuri, and M. Simkin, "Ramen: Souper Fast Three-Party Computation for RAM Programs," in *CCS*, ACM, 2023, pp. 3284–3297.

[SVG24] S. Sasy, A. Vadapalli, and I. Goldberg, "PRAC: Round-Efficient 3-Party MPC for Dynamic Data Structures," *Proc. Priv. Enhancing Technol.*, vol. 2024, no. 3, pp. 692–714, 2024.

[Cao+24] X. Cao *et al.*, "Towards Practical Oblivious Map," *Proc. VLDB Endow.*, vol. 18, no. 3, pp. 688–701, 2024.

[BGI15] E. Boyle, N. Gilboa, and Y. Ishai, "Function Secret Sharing," in *EUROCRYPT (2)*, in Lecture Notes in Computer Science, vol. 9057. Springer, 2015, pp. 337–367.

[BGI16] E. Boyle, N. Gilboa, and Y. Ishai, "Function Secret Sharing: Improvements and Extensions," in *CCS*, ACM, 2016, pp. 1292–1303.

[Guo+23] X. Guo *et al.*, "Half-Tree: Halving the Cost of Tree Expansion in COT and DPF," in *EUROCRYPT (1)*, in Lecture Notes in Computer Science, vol. 14004. Springer, 2023, pp. 330–362.

[Xin+25] P. Xing, H. Li, M. Hao, H. Chen, J. Hu, and D. Liu, "Distributed Function Secret Sharing and Applications," in *NDSS*, The Internet Society, 2025.

[Bun+20] P. Bunn, J. Katz, E. Kushilevitz, and R. Ostrovsky, "Efficient 3-Party Distributed ORAM," in *SCN*, in Lecture Notes in Computer Science, vol. 12238. Springer, 2020, pp. 215–232.

[BKO22] P. Bunn, E. Kushilevitz, and R. Ostrovsky, "CNF-FSS and Its Applications," in *Public Key Cryptography (1)*, in Lecture Notes in Computer Science, vol. 13177. Springer, 2022, pp. 283–314.

[ZYP24] G. Zyskind, A. Yanai, and A. 'Sandy' Pentland, "High-Throughput Three-Party DPFs with Applications to ORAM and Digital Currencies," in *CCS*, ACM, 2024, pp. 4152–4166.

[Bon+21] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Lightweight Techniques for Private Heavy Hitters," in *SP*, IEEE, 2021, pp. 762–776.

[CP22] L. de Castro and A. Polychroniadou, "Lightweight, Maliciously Secure Verifiable Function Secret Sharing," in *EUROCRYPT (1)*, in Lecture Notes in Computer Science, vol. 13275. Springer, 2022, pp. 150–179.

[GKS23] L. Grassi, D. Khovratovich, and M. Schofnegger, "Poseidon2: A Faster Version of the Poseidon Hash Function," in *AFRICACRYPT*, in Lecture Notes in Computer Science, vol. 14064. Springer, 2023, pp. 177–203.

[Yeo22] K. Yeo, "Cuckoo Hashing in Cryptography: Optimal Parameters, Robustness and Applications." [Online]. Available: https://eprint.iacr.org/2022/1455

[Alb+15] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner, "Ciphers for MPC and FHE," in *EUROCRYPT (1)*, in Lecture Notes in Computer Science, vol. 9056. Springer, 2015, pp. 430–454.

[ANS09] Y. Arbitman, M. Naor, and G. Segev, "De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results," in *ICALP (1)*, in Lecture Notes in Computer Science, vol. 5555. Springer, 2009, pp. 107–118.