Concurrent Encrypted Multimaps

Archita Agarwal, Seny Kamara, and Tarik Moataz

MongoDB, New York, NY 10019, USA
{archita_agarwal,seny_kamara,tarik_moataz}@mongodb.com

Abstract. Encrypted data structures have received a lot of attention due to their use as building blocks in the design of fast encrypted search algorithms and encrypted databases. An important design aspect that, as far as we know, has not been considered is that modern server architectures are concurrent in the sense that they support the execution of multiple operations simultaneously. In this work, we initiate the study of concurrent encrypted data structures. We identify new definitional and technical challenges posed by concurrency in the setting of encrypted search. In order to formalize the security of these schemes, we extend the standard framework of structured encryption to capture, among other things, fine-grained leakage which occurs at the instruction level as well as schedule-dependent leakage which changes as a function of the order in which instructions are executed. The latter is particularly challenging to handle when the scheduler is adversarial and adaptive. We provide security definitions in the ideal/real-world model which allows us to capture both security and consistency together.
We combine techniques from structured encryption and concurrent data structures to design the first concurrent encrypted multi-map. We show that it is not only secure and efficient, but also satisfies a strong consistency guarantee called *linearizability* while supporting *lock-free* append operations and requiring no inter-client communication.

Keywords: Encrypted search · Concurrent data structures · Concurrent encrypted data structures.

1 Introduction

Encrypted multi-maps (EMM) are end-to-end encrypted data structures that store label/value pairs and support get and put operations in sub-linear time. EMMs are a core building block in the design of sub-linear encrypted databases and searchable symmetric encryption (SSE) schemes. As encrypted databases gain popularity and interest from industry, new problems at the intersection of cryptography and distributed systems are emerging. One example is the problem of designing encrypted *distributed* data structures studied in [1,2]. These are encrypted data structures designed to be stored and managed by clusters of machines as opposed to a single server as is traditionally considered in the encrypted search literature. Encrypted distributed structures are a crucial building block for the design of real-world encrypted databases since, in practice, most databases are distributed and run on clusters.

Another practical problem that, as far as we know, has received very little attention is the problem of designing *concurrent* encrypted structures by which we mean dynamic encrypted structures that can be accessed concurrently while providing strong consistency guarantees and high throughput. This is a fundamental problem because every real-world database system is concurrent. More precisely, the literature in encrypted search usually models database server executions as sequential in the sense that the server is assumed to execute operations in their entirety one after the other. For example, upon receiving a sequence of operations $(\mathsf{op}_1, \ldots, \mathsf{op}_n)$ from a client it executes op_i entirely before executing op_{i+1}. Even in multi-client settings, the server is assumed to order the operations of the clients and then execute them fully in that order.

Concurrency and consistency. In reality, database servers do not execute operations sequentially. They use multi-threading and often multiple cores to execute many operations simultaneously. This increases operation throughput since the server can make progress on an operation op_i while waiting for an expensive call from operation op_j to return (e.g., a call to disk). Database servers do not view operations as atomic objects that must be executed in their entirety at once but, instead, as sequences of lower-level atomic instructions that can be context-switched at any moment by the operating system scheduler. Concurrency introduces a host of challenges, the most important of which is that the traditional notions of correctness are not meaningful. Consider a multi-map that stores a label/tuple pair $(\ell, \mathbf{v})$, where $\mathbf{v} = (v_1, \ldots, v_{10})$, an append operation (ℓ, v_{11}) and a get operation on ℓ. Now suppose the append and get operations are concurrent. Should we require the get operation to return $(v_1, \ldots, v_{10})$ or $(v_1, \ldots, v_{11})$? Either answer is acceptable depending on exactly how the lower-level instructions of the two operations are scheduled. Because of this, correctness in concurrent settings is replaced with the notion of *consistency* which guarantees that the outputs of the operations are consistent with *some* sequential order of the operations; even if their instructions are actually interleaved. Returning to the example, if the get outputs $(v_1, \ldots, v_{11})$ then its output is consistent with the sequential order (append, get), whereas if it outputs $(v_1, \ldots, v_{10})$ its output is consistent with the sequential order (get, append).

Many different notions of consistency have been defined and studied. Some are weaker in the sense that they allow for more sequential orders and some are stronger in the sense that they allow for fewer. For example, linearizability [35] is a very strong consistency notion which, roughly speaking, guarantees that the output of the operations preserve their real-time ordering, i.e, if op_i completes before op_j begins, then any effect of op_i will be reflected in op_j's output. In our example above, linearizability guarantees that if the append finishes before the get starts, then the get's output will include the value added by the append. In contrast, the weaker notion of sequential consistency [48], would allow the get to output either $(v_1, \ldots, v_{10})$ or $(v_1, \ldots, v_{11})$ as long as the get and append were made by different clients.

Concurrent EMMs. Based on the discussion above, it should be clear that real-world encrypted databases and their underlying data structures need to support concurrent executions and guarantee some notion of consistency. As far as we know, the only semi-dynamic or dynamic EMM construction that can achieve high throughput and some form of consistency under concurrent operations is the OST scheme of [42] which underlies MongoDB's Queryable Encryption. A limitation of the scheme, however, is that the scheme itself is only designed to provide high throughput while consistency is achieved at the implementation level by making use of database transactions. This has two implications. The first is that the scheme itself is not provably consistent. The second is that the way transactions are used may not necessarily lead to optimal throughput. The design of concurrent EMMs is highly non-trivial and to see why we will go over various possible solutions and explain why they do not work.

The first solution one might think of is to start with a dynamic EMM and modify it using standard techniques from the concurrency literature. As an example, one might start with the following simplified version of the π_{bas} construction from [15]. At a high level it works by breaking down a label/tuple pair $(\ell, \mathbf{v})$, where $\mathbf{v} = (v_1, \ldots, v_n)$, into n pairs where the ith pair is of the form $(\ell||i, v_i)$. Each pair is then stored as $(F(K_{\ell,1}, i), \mathsf{Enc}(K_{\ell,2}, v_i))$ in a standard/plaintext dictionary, where F is a pseudo-random function, Enc is a symmetric encryption scheme, and $K_{\ell,1}$ and $K_{\ell,2}$ are label-specific keys computed using a pseudo-random function. To execute a get operation for ℓ, the client sends $K_{\ell,1}$ to the server who uses it as follows. It initializes a counter i to 1 and computes $\mathsf{tag}_i := F(K_{\ell,1}, i)$. It then queries the dictionary on tag_i. If the tag is in the dictionary, the server returns the associated ciphertext, increments the counter and repeats the process. If, on the other hand, tag_i is not in the dictionary it stops. To add a value v_{n+1} to ℓ's tuple, the client sends the pair $(F(K_{\ell,1}, n+1), \mathsf{Enc}(K_{\ell,2}, v_{n+1}))$. Notice that in order to guarantee correctness, the client has to keep local counters for every label (client state is needed by most semi-dynamic and dynamic EMMs that achieve standard notions of security).

Naive server-side synchronization. To achieve consistency with a multi-client stateful dynamic EMM, the clients need to synchronize on their state. This is clear in the case of π_{bas} and to see why consider a setting where two clients $\mathbf{C}_i$ and $\mathbf{C}_j$ concurrently append values v and v', respectively, to a label ℓ's tuple $(v_1, \ldots, v_n)$. In this case, both $\mathbf{C}_i$ and $\mathbf{C}_j$ will use their local state to send $(F(K_{\ell,1}, n+1), v)$ and $(F(K_{\ell,1}, n+1), v')$ to the server which will result in one pair overwriting the other. One approach to address this could be to encrypt and outsource the state to the server and synchronize using locks, which is a primitive that ensures exclusive access to shared resources. To append values, clients would then need to acquire the lock, retrieve and decrypt the state, send the new pair together with an updated and encrypted state and release the lock.

This naive form of server-side synchronization has both consistency and security issues. Suppose we have two clients $\mathbf{C}_i$ and $\mathbf{C}_j$ and that when $\mathbf{C}_i$ acquires the lock the counter is at 5 and when $\mathbf{C}_j$ acquires the lock the counter is at 6. In this case, $\mathbf{C}_i$ sends $(F(K_{\ell,1}, 6), \mathsf{Enc}(K_{\ell,2}, v_i))$ to the server whereas $\mathbf{C}_j$

sends $(F(K_{\ell,1}, 7), \mathsf{Enc}(K_{\ell,2}, v_j))$. Now, we will describe an execution schedule in which $\mathbf{C}_j$'s append will not be output by its own get which proves that the construction is not linearizable. This happens if $\mathbf{C}_j$'s append is scheduled but $\mathbf{C}_i$'s append is not. In this case, $(F(K_{\ell,1}, 7), \mathsf{Enc}(K_{\ell,2}, v_j))$ is inserted into the dictionary whereas $(F(K_{\ell,1}, 6), \mathsf{Enc}(K_{\ell,2}, v_i))$ is not. Notice that if $\mathbf{C}_j$ executes a get on ℓ after its append is completed, it will not receive v_j because when the server queries the dictionary on the tag $F(K_{\ell,1}, 6)$, it will not find it and, therefore, stop and return the values associated with counters 1 through 5. Linearizability requires gets to return all the values of appends that finished before the get started, so this construction is not linearizable. In fact, this construction does not even achieve the weaker notion of sequentially consistency and only satisfies the weakest form of consistency known as *eventual consistency.*

A general approach for linearizability. The problem with the previous approach is that only locking the state is not enough to synchronize append and get operations. A general-purpose way to fix this is to put both the state and the EMM—the dictionary in the case of π_{bas}—together under one lock. In this case, only one operation can change or read the structure/dictionary at a time. This solution, however, severely limits scalability and throughput and essentially operates like a sequential implementation.

Naive state access. Another issue with naive server-side synchronization is that naively accessing the state could leak additional information. Specifically, if the clients only retrieves the relevant (encrypted) counter during an append, the server would learn append-to-append correlations (i.e., whether two appends are for the same label or not). This can be avoided if the clients retrieve the entire state each time or store and query it using an oblivious RAM (ORAM) but both approaches are costly.

1.1 Our Contributions

In this work, we initiate the study of concurrent encrypted data structures. Specifically, we formalize, define and construct a multi-map encryption scheme that encrypts multi-maps in such a way that they can be accessed concurrently with high throughput and that satisfies linearizability. Though our construction is one of our main contributions, we also identify a variety of interesting definitional and modeling issues that need to addressed to even formalize the security of concurrent encrypted structures.

Instruction-level leakage. One of our core observations is that leakage in the concurrent setting is quite different than leakage in the sequential setting and, therefore, needs to be modeled differently. The observation stems from the fact that, as discussed above, operations are really sequences of instructions and that, in reality, leakage occurs at the instruction level and is produced little by little with every instruction that is executed. To see this, consider the simplified version of π_{bas} described above. Notice that during a get, the server learns information piece-by-piece as it queries the dictionary. For example, if the dictionary query for $F(K_{\ell,1}, i)$ is successful the server learns that the length of the tuple is *at least* i. And when the dictionary query fails, it finally learns the exact length of the tuple. The fact that leakage really occurs at the instruction level does not necessarily contradict the standard operation-level leakage model [23,20] which is (implicitly) used in the sequential setting. This is because, in the sequential setting, all the instructions of an operation are executed together before the instructions of the next operation are started. The consequence is that the operation-level leakage is the union of the instruction-level leakage. For example, in π_{bas} the operation-level get leakage is the length of the tuple which is also the instruction-level get leakage in the sequential setting.

Schedule-dependent leakage. Another important observation is that instruction-level leakage depends on how it is scheduled. For example, consider a π_{bas} EMM that stores a label pair $(\ell, \mathbf{v})$, where $\mathbf{v} = (v_1, \ldots, v_{10})$, and the following two concurrent operations: an append to ℓ's tuple and a get for ℓ. Now, consider one schedule where the append adds the pair $(F(K_{\ell,1}, 11), \mathsf{Enc}(K_{\ell,2}, v_{11}))$ to the underlying dictionary before the get operation queries the dictionary for $F(K_{\ell,1}, 11)$ and another schedule where the order is reversed. In the first schedule, the server learns that $\mathbf{v}$ has size at least 11 from the dictionary query for $F(K_{\ell,1}, 11)$, whereas in the second schedule it learns that the tuple length is exactly 11[1].

Adversarial schedulers. As discussed above, real-world database servers are multi-threaded and instructions are ordered and scheduled for execution by the OS scheduler. If the server is corrupted—which is the standard adversarial model considered in encrypted search—then the scheduler is also corrupted and the execution schedule of a sequence of operations will be adversarially-chosen and because of this the observations above have important implications on the security of encrypted structures. In particular, recall that it is standard for dynamic encrypted structures to achieve forward privacy. Roughly speaking, forward privacy guarantees that updates to the structure cannot be correlated with previous queries but can reveal correlations between queries and past updates. The observation that instruction-level leakage is schedule-dependent implies that, in the presence of an adversarial scheduler, the notions of forward and backward privacy are not meaningful. This is simply because an adversarial scheduler can render the guarantees of forward privacy useless by controlling the schedule. Specifically, given a sequence of queries followed by updates, forward privacy guarantees that the updates cannot be correlated to any of the queries. But by scheduling all the updates first followed by the queries these correlations can be revealed. A similar issue occurs with backward privacy which, roughly speaking, guarantees that queries do not reveal information about items that have been previously inserted and then deleted. As in the case of forward privacy, if an

[1] Note that, in this example, the ordering of the operations also impacts the leakage at the operation level.

adversarial scheduler re-orders the operations such that an item is inserted, then queried and then deleted, then it can learn whether the item was inserted.

Note that, so far, we only discussed adversarial schedulers that choose fixed schedules, i.e., in a non-adaptive manner. But a scheduler could determine a schedule *adaptively*, as a function of previous instructions, their outputs and their leakage.

Modeling instruction-level leakage. To capture instruction-level leakage and to properly capture the interaction between adaptive schedulers and leakage, we formalize leakage profiles in a more fine-grained manner. Specifically, we define leakage functions as stateful functions that take as input a sequence of *instructions* (as opposed to a sequence of operations), a schedule for the instructions executed so far and the next instruction to be executed. Based on these inputs, the leakage function determines the leakage produced by the next instruction which, in our security definition, is provided to the adaptive scheduler.

Formalizing security. We formalize the security of a concurrent multi-map encryption scheme in the ideal/real-world paradigm. At a high level, in the real world the clients and server execute the real multi-map encryption scheme operations in the presence of a semi-honest adversary that corrupts the server. In the ideal world, the clients and server interact with an ideal concurrent multi-map functionality. Defining this ideal functionality is challenging for the following reasons. The ideal functionality should produce a sequence of outputs that is consistent. There are, however, many possible consistent output sequences. We could make the functionality produce a specific one of them but this would be too strong and not achievable since, in the real world, the adversary controls the scheduler and can, therefore, influence the outputs of the encrypted multi-map. To capture this, we need to relax the functionality and allow the simulator to provide it with information that it can use to generate an output sequence. But, crucially, we require the functionality to abort if the simulator leads it to output a sequence that violates consistency. This guarantees that the functionality and the scheme always produce consistent output sequences but that the adversary can influence which consistent output sequence it produces. As with traditional (i.e., sequential) STE security definitions, we explicitly model leakage in our definitions.

Another interesting feature of our definition is how adaptive adversarial schedulers are handled. Recall that adaptive schedulers choose the next atomic instruction to execute based on previous instructions, their results and possibly their leakage. During simulation, this is handled by the simulator forwarding the scheduler's next instruction to the functionality which returns the instruction-level leakage so that the simulation can proceed.

A new concurrent multi-map encryption scheme. We describe a linearizable multi-map encryption scheme called TST which, as far as we know, is the first such construction. In addition, TST achieves lock-free append operations which, roughly speaking, means that if an append gets scheduled it will never have to wait on another operation. At a very high level, the scheme works as follows. Suppose we have n clients $\mathbf{C}_1, \ldots, \mathbf{C}_n$. A label ℓ's tuple $\mathbf{v}$ can be split into n sub-tuples $(\mathbf{v}_{1,\ell}, \ldots, \mathbf{v}_{n,\ell})$, where $\mathbf{v}_{i,\ell}$ holds the values appended by $\mathbf{C}_i$. For all clients $\mathbf{C}_i$ and labels ℓ, the encrypted multi-map will store an encrypted form of a reverse linked list $\mathsf{list}_{i,\ell}$, where each node stores a value of $\mathbf{v}_{i,\ell}$ and a pointer to the previous node in the list. It also stores an encrypted structure that stores pointers to the heads of the list. To append a value v_{m+1} to ℓ's tuple, $\mathbf{C}_i$ sends a new node that stores v_{m+1} and points to the head of $\mathsf{list}_{i,\ell}$. To retrieve ℓ's tuple, the server walks each $\mathsf{list}_{i,\ell}$, for $i \in [n]$, starting from their heads and returns the nodes in $\cup_{i\in[n]}\mathsf{list}_{i,\ell}$. Note that, for security reasons, append operations do not update the structure that stores the heads of the list so that structure can store old/stale head pointers which could result in the get only returning a subset of ℓ's tuple. To address this, we augment the construction with additional encrypted data structures that the server can use at get time to find the unreachable nodes (i.e., the nodes that cannot be reached from the old head). This adds false positives to the server's response, but the client can locally filter them out. Additionally, the client can use the results to update the structures at the server so that future gets have less false positives (and depending on the distribution of operations possibly none). Like many dynamic EMM constructions [15,8,9,28,59], TST also uses *lazy deletion* to handle deletes, where deletions are treated as additions with special delete markers that the client can locally use to filter out the deleted values. The description provided here is very high level and ignores many subtleties and technical challenges which we discuss in detail in Section 5 and 6.

One of the main challenges in designing TST was to find a way to achieve linearizability, security and efficiency. Traditional techniques from concurrent data structures are designed to provide consistency and efficiency whereas traditional techniques from structured encryption are designed to achieve security and efficiency. In our setting, we need to find ways of using, adapting and creating new techniques so that we achieve all three. Interestingly, while TST is very different than all previous EMM constructions, it does make use of and combine ideas from previously-known influential constructions. Specifically, it is both a list-based scheme like the SSE-1 construction of [23] and a dictionary- and counter-based scheme like the π_{bas} construction of [15].

A new linearizable range dictionary. Our TST construction makes use of a plaintext range dictionary which stores label/value pairs where the labels are integers. In addition to get and put operations, the range dictionary also needs to support a greater-than ℓ operation that returns the set of values associated with labels greater than ℓ. We construct such an efficient and linearizable range dictionary which may be of independent interest.

2 Related Work

Structured Encryption. Structured encryption was introduced by Chase and Kamara [20] as a generalization of index-based searchable symmetric encryption (SSE) [58,23]. The most common and important type of STE schemes are multi-map encryption schemes which are a basic building block in the design of sub-linear SSE schemes [23,41,15], expressive SSE schemes [13,55,29,39,38] and encrypted databases [39,16]. STE and encrypted multi-maps have been studied along several dimensions including dynamism [41,40,15,55,33] and I/O efficiency [15,14,5,52,24,7,25]. The notion of forward privacy was introduced by Stefanov, Papamanthou and Shi [60] and formally defined by Bost [8], who also proposed the first forward-private encrypted multi-map construction. Kamara and Moataz pointed out in [38] that the definition of [8] does not necessarily capture the intuitive security guarantee of forward-privacy and suggested that it be formalized as requiring that updates be leakage-free. Backward privacy was introduced by Bost, Minaud and Ohrimenko [9]. Several follow up works showed how to improve on the constructions of [9], sometimes achieving both forward and backward privacy [30,45,9,28,59,3]. All these works focus on designing SSE/STE constructions for non-concurrent settings. While these non-concurrent constructions have several applications, concurrent constructions that allow multiple operations to operate on the encrypted data structure simultaneously are more practical and useful.

Multi-user schemes. Multi-user STE/SSE refers to schemes that allow multiple clients to operate on the encrypted structure/collection. Multi-user schemes can be single-writer multi-reader as proposed in [23], multi-writer single-reader, or multi-writer multi-reader. As far as we know all the multi-user constructions proposed—except for [42]—assume the server is sequential and do not support concurrent operations.

Oblivious parallel RAMs. Oblivious parallel RAM (OPRAM) was introduced by Boyle, Chung, and Pass [10] as a generalization of ORAM that compiles an m-CPU PRAM program into an oblivious m-CPU PRAM. Numerous subsequent work improved OPRAM overhead [17,18,37,22,53,19,6]. Although the notions of parallel and concurrent computation are related, they are not the same. Parallel computation involves executing multiple operations at the same time in order to accelerate computationally-intensive tasks by using multiple processing units. In contrast, concurrent computation involves executing multiple operations that can be interleaved with one of another. This means that operations can begin, run, and complete in any sequence and they can share resources such as memory and processors. Our work is focused on concurrency not parallelism.

Concurrent dictionaries. Our construction makes use of linearizable dictionaries which can be instantiated with hash tables based on closed or open addressing. Existing solutions can be lock-based [27,47,49], partially lock-free [36,34], lock-free [50,57,32], or wait-free [61]. Search trees can also be used to instantiate concurrent dictionaries and there are various lock-based designs [46,11] and lock-free linearizable implementations [62,26].

3 Preliminaries

Notation. The set of all binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. $[n]$ is the set of integers $\{1,\ldots,n\}$. $a := b$ means that a is set to b. The output y of a deterministic algorithm $\mathcal{A}$ on input x is denoted by $y := \mathcal{A}(x)$. If S is a set then $x \overset{\$}{\leftarrow} S$ denotes sampling from S uniformly at random. If S is a set then $\#S$ refers to its cardinality. Throughout, k will denote the security parameter.

Cryptographic primitives. We make use of CPA-secure symmetric encryption schemes $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and pseudo-random functions (PRF) in our construction. We denote the evaluation of a pseudo-random function F with a key K on an input x as $F(K, x)$. Sometimes for visual clarity, we denote $F(F(F(K, x_1), x_2), \ldots, x_n)$ as $F[K, x_1, \ldots, x_n]$. We refer the reader to [44] for standard notions of security for PRFs and symmetric encryption schemes.

Plaintext data structure schemes. A dictionary scheme $\Delta_{\mathsf{DX}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put})$ supports three algorithms where $\mathsf{Init}(k_1, k_2)$ initializes an empty dictionary DX that maps labels of length k_1 to values of length k_2, $\mathsf{Put}(\mathsf{DX}, \ell, v)$ adds a label/value pair (ℓ, v) to DX, and $\mathsf{Get}(\mathsf{DX}, \ell)$ returns the value v associated with label ℓ in DX. A cas-dictionary scheme $\Delta_{\overline{\mathsf{DX}}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put}, \mathtt{CompareAndSwap})$ is a dictionary scheme that supports, in addition to Get and Put, a $\mathtt{CompareAndSwap}$ algorithm. $\mathtt{CompareAndSwap}(\overline{\mathsf{DX}}, \ell, v_{\mathsf{old}}, v_{\mathsf{new}})$ compares the value of label ℓ with v_{old} and updates it with v_{new} if they are equal [51,56,54]. A range dictionary scheme $\Delta_{\mathsf{RDX}} = (\mathsf{Init}, \mathsf{Put}, \mathsf{GetGreater})$ is a dictionary scheme that supports $\mathsf{GetGreater}$ instead of Get. In particular, the $\mathsf{GetGreater}(\mathsf{RDX}, \ell)$ algorithm returns all the values in the dictionary that have labels ℓ' greater than ℓ. Finally, $\Delta_{\mathsf{CTR}} = (\mathsf{Init}, \mathsf{FetchAndInc})$ is a counter scheme where $\mathsf{Init}(v)$ initializes a counter $\mathsf{count_g}$ with value v, and $\mathsf{FetchAndInc}(\mathsf{count_g})$ returns the current value of $\mathsf{count_g}$ and increments the counter by 1.

Operations. We define an operation op as a tuple $(\mathsf{opid}, \mathsf{name}, \mathsf{inp}, \mathsf{cid})$ which includes a unique operation id, the operation's name, its input, and the id of the client who issued the operation. For example, $\mathsf{op} = (\mathsf{opid}, \mathsf{Get}, \ell, i)$ is a get operation that takes a label ℓ as its input and is issued by client $\mathbf{C}_i$.

Atomic instructions. We assume that each operation op consists of atomic instructions $(\mathsf{ins}_1, \ldots, \mathsf{ins}_\lambda)$, which are instructions that guarantee uninterrupted access and updates of shared single-word variables. We use $\mathsf{op}.[\mathtt{First}]$ and $\mathsf{op}.[\mathtt{Last}]$ to indicate the first and last atomic instruction in a sequence of atomic operations that make up an operation. We also use $\mathsf{op}.[\mathsf{ins}_i]$ or $\mathsf{opid}.[\mathsf{ins}_i]$ to refer to the ith instruction of operation op.

Execution schedules. We assume the server has a scheduler that is responsible for scheduling operations on the CPU. Given a set of operations $\Omega = \{\mathsf{op}_1, \ldots, \mathsf{op}_\lambda\}$, the execution process is as follows: (1) the scheduler schedules an operation; (2)

the CPU executes the *next* atomic instruction of the scheduled operation; and (3) then the scheduler de-schedules the operation. After that, the scheduler schedules another operation, the CPU which carries out the next atomic instruction of that operation, and so on. This process defines an *execution schedule*, sched_{Ω}, for a set of operations Ω.

It is important to note that the *next* atomic instruction of the scheduled operation is not necessarily next atomic instruction in the literal code of the operation. Instead, it is determined by accounting for branches and other control flow constructs. Consequently, two get operations with identical code but different inputs can follow entirely different execution paths due to different inputs and can, therefore, have different execution schedules.

We define a schedule sched_{Ω} for a set of operations Ω as a prefix of the following sequence: $(\mathsf{opid}_1.[\mathtt{First}], \ldots, \mathsf{opid}_1.[\mathtt{Last}]) \times \ldots \times (\mathsf{opid}_{\lambda}.[\mathtt{First}], \ldots, \mathsf{opid}_{\lambda}.[\mathtt{Last}])$, where the $\times$ operator denotes the the interleaving of executions. We stress that execution schedules need not be complete in the sense that they do not have to contain all the atomic instructions of an operation. Intuitively, they represent the execution that has happened *so far* on the machine. Given an operation set Ω, and a schedule sched_{Ω}, we partition Ω into two disjoint sets $\Omega_f \cup \Omega_p$, where Ω_f is the set of operations that are completed, and Ω_p is the set of operations that are partially completed. Formally an operation $(\mathsf{opid}, \star, \star, \star) \in \Omega$ is in Ω_f if $(\mathsf{opid}.[\mathtt{First}], \ldots, \mathsf{opid}.[\mathtt{Last}]) \in \mathsf{sched}_{\Omega}$, otherwise it is in Ω_p.

Concurrent schedules. Given a schedule sched_{Ω}, we write $\mathsf{time}_{\mathsf{sched}}(\mathsf{op}.[\mathsf{ins}])$ to refer to the logical time the instruction $\mathsf{op}.[\mathsf{ins}]$ is executed. For instance, for an operation op, $\mathsf{time}_{\mathsf{sched}}(\mathsf{op}.[\mathtt{First}])$ and $\mathsf{time}_{\mathsf{sched}}(\mathsf{op}.[\mathtt{Last}])$ refer to the start and end times of the operation. If $\mathsf{op}.[\mathtt{Last}] \notin \mathsf{sched}_{\Omega}$, then $\mathsf{time}_{\mathsf{sched}}(\mathsf{op}.[\mathtt{Last}]) = \infty$. A schedule sched_{Ω} is called *concurrent* if there exists a time when two operations are “active”. Formally, there exist a time t, such that $\mathsf{time}_{\mathsf{sched}}(\mathsf{op}.[\mathtt{First}]) \leq t \leq \mathsf{time}_{\mathsf{sched}}(\mathsf{op}.[\mathtt{Last}])$ and that $\mathsf{time}_{\mathsf{sched}}(\mathsf{op}'.[\mathtt{First}]) \leq t \leq \mathsf{time}_{\mathsf{sched}}(\mathsf{op}'.[\mathtt{Last}])$. We say that op and op' are concurrent if this condition is true.

Execution histories. An *execution history* is a record of the operations executed on a data structure, when they were executed, and their outputs were. We model it as a triple $H = (\Omega, \mathsf{sched}_{\Omega}, \mathsf{out}_{\Omega_f})$, where sched_{Ω} is the execution schedule of the operations in Ω and out_{Ω_f} is an output function that assigns an output to finished operations $\Omega_f \subseteq \Omega$.

Correctness of concurrent data structures. A concurrent data structure is a data structure that allows multiple operations to be executed simultaneously without violating its correctness. The correctness of a concurrent data structure is formalized by a notion of consistency which, intuitively, guarantees that the operations executed on a data structure should always appear to be executed one after the other, even if their executions were interleaved. For instance, a get operation on a multi-map should output all the values that appear to have been added by “previous” appends, without including any of the values that appear to have been added by “later” appends. We capture the notion of *sequential* *correctness* with a function called exec. It takes as input a total order seq_Ω of operations Ω, an operation $\mathsf{op} \in \Omega$, and returns the output of op as if the operations were executed in the order of seq_Ω. In the particular case of a multi-map, exec is defined as follows. If $\mathsf{op} = (\mathsf{opid}, \mathsf{Get}, \ell, \star)$, $\mathsf{exec}(\mathsf{seq}_\Omega, \mathsf{op}) = \{v : (\mathsf{opid}', \mathsf{Append}, (\ell, v), \star) \in \Omega$ and seq_Ω orders opid' before $\mathsf{opid}\}$. Various consistency notions define how operations in Ω can be ordered in relation to their original ordering in sched_Ω, determining what is considered "previous" or "later". Stricter consistency results in fewer possible orderings, while weaker consistency allows for more. We formalize this intuition in the definition in the definition below where we use a predicate χ to capture constraints on how operations can be ordered. Given a schedule sched_Ω and a sequential order seq_Ω, it outputs 1 if seq_Ω orders certain operation pairs in the same way as sched_Ω, and 0 otherwise.

Definition 1 (χ-consistent histories). *A history $H = (\Omega, \mathsf{sched}_\Omega, \mathsf{out}_{\Omega_f})$ is χ-consistent if there exists a sequence seq_Ω such that:*

$$\chi(\mathsf{sched}_\Omega, \mathsf{seq}_\Omega) = 1 \quad \textit{and} \quad \forall\ \mathsf{op} \in \Omega_f,\ \mathsf{exec}(\mathsf{seq}_\Omega, \mathsf{op}) = \mathsf{out}_{\Omega_f}(\mathsf{op}),$$

where $\Omega_f \subseteq \Omega$ is the set of completed operations, and exec is a function that assigns op an output that conforms to the sequential correctness of the data structure when executing operations in Ω in the sequential order seq_Ω.

We say that a concurrent data structure is called χ-consistent, if all its execution histories are χ-consistent.

Linearizability. Linearizability [35] is a popular and strong consistency notion that requires operations to preserve their real-time ordering, i.e, if op completes before operation op' begins, then op should take effect before op. Said differently, it implies that operations should appear to be interleaved at the granularity of complete operations, and the order of non-overlapping operations is preserved. We formally define this notion below.

Definition 2 (Linearizability). *An execution history $H = (\Omega, \mathsf{sched}_\Omega, \mathsf{out}_{\Omega_f})$ is* linearizable *if the two conditions below are verified:*

1. (span membership)*: for each operation $\mathsf{op} \in \Omega$, there exists a point in time $\mathsf{linp}(\mathsf{op}) \in \mathbb{R}$, called its* linearization point*, such that*

$$\mathsf{time}_{\mathsf{sched}}(\mathsf{op}.[\mathit{First}]) \leq \mathsf{linp}(\mathsf{op}) \leq \mathsf{time}_{\mathsf{sched}}(\mathsf{op}.[\mathit{Last}])$$

2. (correctness)*: for all $\mathsf{op} \in \Omega_f$, $\mathsf{exec}(\mathsf{seq}_\Omega^{\mathsf{linp}}, \mathsf{op}) = \mathsf{out}_\Omega(\mathsf{op})$, where $\mathsf{seq}_\Omega^{\mathsf{linp}}$ is created from the linearization points as follows. Let op and op' be two operations in Ω and let opid and opid' be their operation ids. If $\mathsf{linp}(\mathsf{op}) < \mathsf{linp}(\mathsf{op}')$, then order opid before opid' in $\mathsf{seq}_\Omega^{\mathsf{linp}}$.*

Intuitively, the linearization point of an operation captures the instant when the operation appears to have taken effect. For example, the linearization point of an append operation is the point in its execution before which its value was not in the multi-map but after which it definitely is.

Progress guarantees. The concept of termination is more complicated in concurrent execution than in sequential execution because an operation's completion depends not only on its own execution but also on the execution of other operations. For instance, an operation that is waiting on a lock cannot proceed until the operation that holds the lock releases it. Progress guarantees define conditions under which an operation is ensured to complete. These guarantees are broadly classified as non-blocking and blocking, where the former allows other operations to proceed even if one operation is delayed, while the latter does not. Non-blocking guarantees are further classified into wait-freedom and lock-freedom. An operation is considered wait-free if its executions complete in a finite number of scheduler steps. In contrast, an operation is lock-free if it guarantees that some operation call will finish in a finite number of scheduler steps, even if not all calls will. Similarly, blocking guarantees are further classified into starvation-freedom and deadlock-freedom. An operation is starvation-free if its executions can make progress provided that the locks are not held infinitely by the other executions. On the other hand, an operation is deadlock-free if some call will make progress.

4 Definitions

Structured encryption (STE) was introduced in [20] as a generalization of index-based[2] SSE schemes [23]. The notion of SSE was introduced in [58] and formalized in [23]. There are several forms of structured encryption. The original definition of [20] considered schemes that encrypt both a structure and a set of associated data items (e.g., documents, emails, user profiles etc.). In [21], the authors also describe *structure-only* schemes which only encrypt structures. One can also distinguish between *response-hiding* and *response-revealing* schemes: the former reveal the response to queries whereas the latter do not.

Definition 3 (Append-only multi-map encryption scheme). *A response-hiding multi-client append-only multi-map encryption scheme* $\Sigma_{\mathsf{MM}} = (\mathsf{Init}, \mathsf{Append}, \mathsf{Get})$ *consists of three two-party protocols that are executed by* n *clients* $\mathbf{C}_1, \ldots \mathbf{C}_n$ *and a server* $\mathbf{S}$ *and work as follows:*

- $(K_1, \mathsf{st}_1; \ldots; K_n, \mathsf{st}_n; \mathsf{EMM}) \leftarrow \mathsf{Init}_{\mathbf{C}_1,\ldots,\mathbf{C}_n,\mathbf{S}}(1^k; \ldots; 1^k; 1^k)$*: is a probabilistic algorithm that takes as input from the clients and server a security parameter* 1^k. *It outputs to a client* $\mathbf{C}_i$ *a key* K_i *and state* st_i *and to the server an encrypted multi-map* EMM*;*
- $(\mathsf{st}'_i; \mathsf{EMM}') \leftarrow \mathsf{Append}_{\mathbf{C}_i,\mathbf{S}}(K_i, \mathsf{st}_i, (\ell, v); \mathsf{EMM})$*: takes as input from the client its key* K_i *and state* st_i *and a label/value pair* (ℓ, v)*; and from the server an encrypted multi-map* EMM. *It outputs to the client an updated state* st'_i *and to the server an updated encrypted multi-map* EMM'*;*
- $(\mathsf{st}'_i, \mathbf{v}; \perp) \leftarrow \mathsf{Get}_{\mathbf{C}_i,\mathbf{S}}(K_i, \mathsf{st}_i, \ell; \mathsf{EMM})$*: takes as input from the client its key* K_i *and state* st_i *and a label* ℓ*; and from the server an encrypted multi-map*

[2] In the literature structure-based schemes are also called index-based schemes.

EMM. *It outputs to the client a (possibly) updated state* st_i' *and a tuple* $\mathbf{v}$ *and to the server* $\perp$*;*

We stress that all the protocols (except the Init) can be executed by many clients concurrently. For parameters $\theta, \lambda \in \mathbb{N}_{\geq 1}$, we use $\mathbb{L}_{\mathsf{MM}} = \{0,1\}^\theta$ to denote the label space and $\mathbb{V}_{\mathsf{MM}} = \{0,1\}^\lambda$ to denote the value space of the multi-map.

4.1 Security Definition

We now turn to formalizing the security of a *concurrent* multi-map encryption scheme. We do this by combining the definitional approaches used in secure multi-party computation [12] and in structured encryption [23,20]. The security of multi-party protocols is generally formalized using the ideal/real-world paradigm. To capture the fact that a protocol could leak information to the adversary, we parameterize the definition with a leakage profile that consists of a leakage function $\mathcal{L}$ that captures the information leaked by the execution of the operations.

Adversarial model. In this work, we consider semi-honest adversaries that corrupt the server and, therefore, see all its stored data, randomness, client operations, and shared memory instructions. Furthermore, we assume that the adversary has control over the scheduler and can determine which atomic instruction is executed at any given time. This implies that the adversary selects a schedule sched_Ω for a given operation set Ω.

The real-world execution. The real-world experiment is executed between a set of n clients $\mathbf{C}_1, \ldots, \mathbf{C}_n$, a server $\mathbf{S}$, an environment $\mathcal{Z}$ and an adversary $\mathcal{A}$. Given $z \in \{0,1\}^*$, the environment $\mathcal{Z}$ sends a message to the adversary $\mathcal{A}$ to corrupt the server $\mathbf{S}$. The clients and the server then execute $\Sigma_{\mathsf{CMM}}.\mathsf{Init}(1^k)$. $\mathcal{Z}$ then adaptively chooses a polynomial number of operations $(\mathsf{op}_1, \ldots, \mathsf{op}_q)$, where op_j is either a (Get, ℓ, i) tuple or a $(\mathsf{Append}, (\ell, v), i)$ tuple. For all $j \in [q]$, $\mathcal{Z}$ sends op_j to client $\mathbf{C}_i$. If op_j is a get operation $\mathbf{C}_i$ executes $\Sigma_{\mathsf{CMM}}.\mathsf{Get}$ with the server but if it is an append operation, $\mathbf{C}_i$ executes $\Sigma_{\mathsf{CMM}}.\mathsf{Append}$. The adversary also communicates with the environment throughout the run of the experiment. Since the adversary controls the scheduler and also communicates with the environment, $\mathcal{A}$ and $\mathcal{Z}$ decide how to schedule operations. When an operation op_j finishes, the server returns the response to the right client $\mathbf{C}_i$ which, in turn, sends it to the environment $\mathcal{Z}$. After all the operations are executed, the adversary $\mathcal{A}$ sends a message m to $\mathcal{Z}$ who returns a bit that is output by the experiment. We let $\mathbf{Real}_{\mathcal{A},\mathcal{Z}}(k)$ be the random variable denoting $\mathcal{Z}$'s output bit.

The ideal-world experiment. The experiment is executed between a set of n dummy clients $\mathbf{C}_1, \ldots, \mathbf{C}_n$, an environment $\mathcal{Z}$ and a simulator Sim, where the environment and the simulator can communicate at any point in the experiment. Each party also has access to the ideal functionality $\mathcal{F}_{\mathsf{CMM}}^{\chi,\mathcal{L}}$. Given $z \in \{0,1\}^*$, $\mathcal{Z}$ sends a message to the simulator Sim to corrupt the simulated server $\mathbf{S}$. $\mathcal{Z}$

then adaptively chooses a polynomial number of operations $(\mathsf{op}_1, \ldots, \mathsf{op}_q)$, where op_j is either a (Get, ℓ, i) tuple or an $(\mathsf{Append}, (\ell, v), i)$ tuple. For all $j \in [q]$, $\mathcal{Z}$ sends op_j to a dummy client $\mathbf{C}_i$ which forwards to it to the functionality $\mathcal{F}^{\chi,\mathcal{L}}_{\mathsf{CMM}}$. Upon receiving a message the functionality executes its prescribed procedure from Fig 1 with simulator Sim. When a dummy client receives an output from the functionality, it forwards it to $\mathcal{Z}$. In the end, Sim computes a message m from its view and sends it to $\mathcal{Z}$. Finally, $\mathcal{Z}$ returns a bit that is output by the experiment. We let $\mathbf{Ideal}_{\mathsf{Sim},\mathcal{Z}}(k)$ be the random variable denoting $\mathcal{Z}$'s output bit.

Definition 4 ($(\chi, \mathcal{L})$-security). *We say that a concurrent encrypted multi-map scheme $\Sigma_{\mathsf{MM}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Append})$ is $(\chi, \mathcal{L})$-secure, if for all PPT adversaries $\mathcal{A}$, and all PPT environments $\mathcal{Z}$, there exists a PPT simulator* Sim *such that for all $z \in \{0,1\}^*$, $|\Pr[\mathbf{Real}_{\mathcal{A},\mathcal{Z}}(k) = 1] - \Pr[\mathbf{Ideal}_{\mathsf{Sim},\mathcal{Z}}(k) = 1]| \leq \mathsf{negl}(k)$.*

Note that in both experiments, the environment can send an operation to any client at any time so there can be multiple operations executed concurrently at the server.

4.2 An Ideal Concurrent Multi-Map Functionality

Our ideal functionality captures all the properties of a secure concurrent multi-map, in particular, its consistency and security guarantees.

Capturing consistency in the ideal functionality is challenging. We begin by discussing two challenges that arise in capturing the consistency guarantees of a concurrent multi-map in the ideal/real-world paradigm. The first challenge is that operations take time to execute and do not finish instantaneously in the real world. To address this, we need to create an ideal functionality that can account for this behavior. One option is to allow the functionality to choose an arbitrary time to return output. However, this approach is problematic because in the real world, the adversary controls when an operation ends, and there is no way for the functionality to know this. The second challenge is determining the outputs that the functionality should produce for operations. We want to achieve χ-consistency, but there are multiple possible output sequences that can satisfy this. Creating a functionality that chooses a specific sequence is also not achievable because the scheduler can influence the outputs by forcing a specific interleaving of atomic instructions. To address these challenges, we relax the functionality and allow the simulator (i.e., the ideal adversary) to influence the functionality's output. When an operation ends, the simulator gives the functionality a sequential order of operations that it uses to compute the operation's output.

The functionality overview. We formally describe the ideal concurrent multi-map functionality $\mathcal{F}^{\chi,\mathcal{L}}_{\mathsf{CMM}}$ in Figure 1. The functionality stores two operation sets Ω and Ω_f, where Ω stores all the client operations, and $\Omega_f \subseteq \Omega$ stores all the completed operations. Both the sets start out as empty sets. The functionality also stores a schedule sched of operations, and an output function out, both of which it builds over time. When the functionality receives an operation op from a client $\mathbf{C}_i$, the functionality assigns the operation a unique opid, adds the operation to Ω, and sends to the simulator the operation id, its name, and the client id. Note that our functionality implicitly leaks the type of the operation, and the client making that operation. As previously discussed, in concurrent settings, the scheduler selects the next instruction to be executed. This means that the simulator must receive information on the leakages at the instruction level. In order to obtain the leakages of an operation op's instruction ins, the simulator sends a message $(\mathsf{opid}, \mathsf{ins})$ to the functionality to obtain its leakage. The functionality first checks its local schedule sched to confirm if ins is the next instruction that needs to be executed for op. If yes, it sends $\mathcal{L}(\Omega, \mathsf{sched}, \mathsf{opid.ins})$ to the simulator and updates the schedule sched with $\mathsf{opid.ins}$. Otherwise, it aborts.

Moreover, when $\mathsf{opid.ins}$ is the last instruction for operation op, the simulator additionally sends the functionality a sequential order seq of operations. Recall that we want to capture the notion of χ-consistency in the functionality. Therefore, the functionality makes the checks required by the definition of χ consistency. In particular, it checks if seq is sequential, if $\chi(\mathsf{sched}, \mathsf{seq}) = 1$, and if for all the operations $\mathsf{op}' \in \Omega_f$ completed so far, if $\mathsf{exec}(\mathsf{seq}, \mathsf{op}') = \mathsf{out}(\mathsf{op}')$. If seq passes all the checks, the functionality accepts seq, else it aborts. It finally computes the output $r = \mathsf{exec}(\mathsf{seq}, \mathsf{op})$ of the operation just completed and returns it to $\mathbf{C}_i$. Finally, it updates its set of completed operations Ω_f and adds op to it.

5 A (Plaintext) Linearizable Multi-Map

In this section, we describe a linearizable multi-client plaintext multi-map. This structure underlies our main multi-map encryption scheme TST and will make its description in Section 6 easier to understand. We start with a straw-man construction and gradually build towards a final construction while highlighting various security, efficiency and concurrency considerations.

5.1 An Initial Design

We construct a multi-client plaintext multi-map data structure $\mathsf{MM} = (\mathsf{dDX}, \mathsf{cDX}_1, \ldots, \mathsf{cDX}_n)$ that is composed of a *data dictionary* dDX and n *checkpoint dictionaries* cDX_i, where n is the number of clients. For simplicity, we assume the structure is accessed by a fixed number of clients, $\mathbf{C}_1, \ldots, \mathbf{C}_n$, but note that the structure can handle a variable number of clients. The data dictionary dDX will store labels and tuple values and the checkpoint dictionaries will store meta-data necessary for correctness and fast get operations.

Intuitively, the structure stores a reverse linked list $\mathsf{list}_{i,\ell}$ for each client $\mathbf{C}_i$ that inserts a label/tuple pair $(\ell, \mathbf{v}_i)$. The list $\mathsf{list}_{i,\ell} = (\mathsf{node}_{i,\ell,1}, \ldots, \mathsf{node}_{i,\ell,m})$ is

Functionality $\mathcal{F}_{\mathsf{CMM}}^{\chi,\mathcal{L}}$

The functionality stores sets Ω and Ω_f both initialized as empty sets. It also stores a schedule sched initialized as empty and a dictionary out that stores the outputs of operations.

- upon receiving an operation name $\mathsf{name} \in \{\mathsf{Get}, \mathsf{Append}\}$ and its input inp from a client $\mathbf{C}_i$, the functionality
  1. generates a unique opid for the operation
  2. sets $\mathsf{op} := (\mathsf{opid}, \mathsf{name}, \mathsf{inp}, i)$
  3. adds op to Ω
  4. sends $(\mathsf{opid}, \mathsf{name}, i)$ to the simulator Sim
- upon receiving $(\mathsf{id}, \mathsf{ins}, \{\mathsf{seq}, \perp\})$ from the simulator Sim, the functionality
  1. lets $\mathsf{op} := (\mathsf{opid}, \mathsf{name}, \mathsf{inp}, i)$ be the operation from Ω where $\mathsf{opid} = \mathsf{id}$
  2. uses sched to check if $\mathsf{opid.ins}$ is the next instruction to be executed for op; if not, it aborts
  3. sends $\mathcal{L}(\Omega, \mathsf{sched}, \mathsf{opid.ins})$ to the simulator Sim
  4. concatenates $\mathsf{opid.ins}$ to the schedule, i.e., $\mathsf{sched} = \mathsf{sched} \circ \mathsf{opid.ins}$
  5. if $(\mathsf{ins} = \mathtt{Last})$, it:
     (a) makes the following checks, and if any of them fails, it aborts
        i. checks if $\chi(\mathsf{sched}, \mathsf{seq}) = 1$
        ii. checks if for all $\mathsf{op}' \in \Omega_f$, $\mathsf{exec}(\mathsf{seq}, \mathsf{op}') = \mathsf{out}(\mathsf{op}')$
     (b) computes $r := \mathsf{exec}(\mathsf{seq}, \mathsf{op})$
     (c) sets $\mathsf{out}(\mathsf{op}) := r$
     (d) updates $\Omega_f := \Omega_f \cup \{\mathsf{op}\}$

Fig. 1. The concurrent multi-map functionality parameterized with consistency guarantee χ and a leakage function $\mathcal{L}$.

composed of $m = \#\mathbf{v}_i$ nodes $\mathsf{node}_{i,\ell,j} = (v_j, \mathsf{addr}_{i,\ell,j-1})$, each of which stores a value $v_j \in \mathbf{v}_i$ and a pointer $\mathsf{addr}_{i,\ell,j-1}$ to the previous node, where $\mathsf{addr}_{i,\ell,0} = \perp$. The address of $\mathsf{list}_{i,\ell}$'s head is then stored in $\mathbf{C}_i$'s checkpoint dictionary cDX_i. Storing label/tuple pairs using per-client lists has several advantages, one of which is that it enables concurrent appends without needing clients to synchronize on their state. Typically, nodes of the lists are stored in memory and pointers are memory addresses but, in our case, we store them in the data dictionary dDX so addresses and pointers are dDX labels. Specifically, each node $\mathsf{node}_{i,\ell,j} = (v_j, \mathsf{addr}_{i,\ell,j-1})$ is stored in dDX by setting $\mathsf{dDX}[\mathsf{addr}_{i,\ell,j}] := \mathsf{node}_{i,\ell,j}$, where $\mathsf{addr}_{i,\ell,j}$ is a k-bit string chosen uniformly at random. The address $\mathsf{addr}_{i,\ell,m}$ of $\mathsf{list}_{i,\ell}$'s head is then stored in $\mathbf{C}_i$'s checkpoint dictionary by setting $\mathsf{cDX}_i[\ell] := \mathsf{addr}_{i,\ell,m}$. Throughout, we will sometimes refer to the nodes of $\mathsf{list}_{i,\ell}$ as (i, ℓ)-nodes, to the nodes in dDX inserted by client $\mathbf{C}_i$ as i-nodes and to the nodes in dDX that hold ℓ's values as ℓ-nodes.

Get. To get the tuple associated with a label ℓ, a client $\mathbf{C}_i$ sends ℓ to the server. The latter then retrieves the head address of every ℓ-list and recovers the values from those lists. More precisely, for all $i \in [n]$, the server computes $\mathsf{addr}_{i,\ell,m} := \mathsf{cDX}_i[\ell]$ and $\mathsf{node}_{i,\ell,m} := \mathsf{dDX}[\mathsf{addr}_{i,\ell,m}]$, parses $\mathsf{node}_{i,\ell,m}$ as $(v_m, \mathsf{addr}_{i,\ell,m-1})$, adds v_m to the response $\mathbf{r}$, and then performs the same steps for the node at address $\mathsf{addr}_{i,\ell,m-1}$ until it reaches a node such that $\mathsf{addr}_{i,\ell,j-1} = \perp$.

Append. To append a value v_{m+1} from client $\mathbf{C}_i$ to the tuple of a label ℓ, the client sends (ℓ, v_{m+1}) to the server. The latter computes $\mathsf{addr}_{i,\ell,m} := \mathsf{cDX}_i[\ell]$, creates a new node $\mathsf{node}_{i,\ell,m+1} = (v_{m+1}, \mathsf{addr}_{i,\ell,m})$, samples a new dDX label $\mathsf{addr}_{i,\ell,m+1} \overset{\$}{\leftarrow} \{0,1\}^k$ and inserts the node into the data dictionary by setting $\mathsf{dDX}[\mathsf{addr}_{i,\ell,m+1}] := \mathsf{node}_{i,\ell,m+1}$. It then updates $\mathbf{C}_i$'s checkpoint dictionary by setting $\mathsf{cDX}_i[\ell] := \mathsf{addr}_{i,\ell,m+1}$.

5.2 Towards a Secure Design

The structure described above is efficient and is straightforward to encrypt using any of a variety of practical dictionary encryption schemes. The resulting encrypted structure would also be efficient but it would not achieve the level of security we want. To see why, consider the case where $\mathbf{C}_i$ performs an append on ℓ twice. Even if the multi-map is encrypted, the server would learn that $\mathbf{C}_i$ appended to the same label twice because the two operations cause the server to set $\mathsf{cDX}_i[\ell]$ twice. These append-to-append correlations are problematic because they reveal the length of the tuple already at append time. Another issue is that this approach also reveals get-to-append correlations which can be exploited using adaptive injection attacks [63].

Leakage-free appends through client state. To address this, we want a solution with no append leakage at all. This could be achieved by storing and accessing the checkpoint dictionaries using black-box ORAM simulation or by using a leakage-free dictionary [43,31], but this would result in high overhead and multiple rounds of interaction. Instead, we take a different approach and modify the append operation as follows. Specifically, appends will store the new node in the data dictionary dDX but will not update the checkpoint dictionaries. We also require the clients to store local state that maps labels to the head addresses of their own lists (but not of other clients' lists). During an append operation for (ℓ, v_{m+1}), the client $\mathbf{C}_i$ sends to the server a pair $(\mathsf{addr}_{i,\ell,m+1}, \mathsf{node}_{i,\ell,m+1})$, where $\mathsf{addr}_{i,\ell,m+1} \overset{\$}{\leftarrow} \{0,1\}^k$ and $\mathsf{node}_{i,\ell,m+1} := (v_{m+1}, \mathsf{addr}_{i,\ell,m})$ and $\mathsf{addr}_{i,\ell,m}$ is retrieved from its local state. The client then updates its local state to map ℓ to $\mathsf{addr}_{i,\ell,m+1}$ and the server inserts the new node in the data dictionary dDX by setting $\mathsf{dDX}[\mathsf{addr}_{i,\ell,m+1}] := \mathsf{node}_{i,\ell,m+1}$.

This guarantees that updates are leakage-free (modulo the fact that an append occurred) but introduces a correctness issue for get operations. Specifically, the addresses stored in the checkpoint dictionaries can be out of date in the sense that they do not necessarily point to the heads of the lists anymore. This, in turn, means that there are nodes in each list that could be unreachable and not returned by get operations. Throughout, we will refer to the addresses stored in the checkpoint dictionaries as *checkpoint addresses* and to the nodes pointed to by those addresses as *checkpoint nodes* and recall that, with the current design, checkpoint nodes are not necessarily heads.

Scanning and filtering. One way to solve the correctness issue above is to store the label ℓ in the nodes so that they now have the form $(v_j, \mathsf{addr}_{i,\ell,j-1}, \ell)$ instead of $(v_j, \mathsf{addr}_{i,\ell,j-1})$ and to change the get operations to work in two phases as follows. The first phase works as before; that is, for all $i \in [n]$, the server retrieves the checkpoint address $\mathsf{addr}_{i,\ell,j} := \mathsf{cDX}_i[\ell]$ and traverses the list to recover the values stored in the nodes. In the second phase, it recovers the unreachable nodes by scanning all the untouched nodes in dDX (i.e, the nodes in dDX it did not access in the first phase), checking if they hold ℓ or not and, if so, returning the value in the node.

5.3 Towards a Secure and Efficient Design

With the changes made so far, we solved the leakage and correctness issues but introduced non-trivial efficiency overhead. While appends remain optimal, gets are now linear in the size of the multi-map due to the scanning step. In this section, we show how solve this issue.

Time. We first provide an intuitive explanation of how we can address this limitation and then show how to instantiate it concretely. Recall that the purpose of the linear scan in the second phase of gets is to find the nodes that are unreachable from the address stored in the checkpoint dictionary. Our solution will be to build a new set of data structures that map labels to the unreachable nodes so that we can replace the linear scan with an efficient data structure query. Building such structures is possible because of the following key observation: the nodes in the lists $\mathsf{list}_{1,\ell}, \ldots, \mathsf{list}_{n,\ell}$ that are unreachable are the ones that were inserted into dDX after the checkpoint nodes. Based on this observation we can modify the operations to work as follows. First, we make the server include a timestamp in every node when it inserts them in dDX during an append. The server also maintains n auxiliary range dictionaries $\mathsf{RDX}_1, \ldots, \mathsf{RDX}_n$ such that RDX_i maps timestamps to the labels/addresses of $\mathbf{C}_i$'s nodes in dDX with the associated timestamp. During a get, the server then does the following. For all $i \in [n]$, it will retrieve the ith checkpoint node by computing $\mathsf{addr}_{i,\ell,j} := \mathsf{cDX}_i[\ell]$ and $\mathsf{node}_{i,\ell,j} := \mathsf{dDX}[\mathsf{addr}_{i,\ell,j}]$. Recall that $\mathsf{node}_{i,\ell,j}$ now has form $(v_j, \mathsf{addr}_{i,\ell,j-1}, \ell, \mathsf{time}_j)$. It then retrieves the unreachable nodes by: (1) computing $(\mathsf{addr}_1, \ldots, \mathsf{addr}_p) := \mathsf{RDX}_i[\mathsf{time}_j, \infty]$; (2) retrieving nodes $\mathsf{node}_z := \mathsf{dDX}[\mathsf{addr}_z]$, for $z \in [p]$; and (3) filtering out the nodes that hold ℓ. Note that this time-based solution relies on the assumption that timestamps are strictly increasing; that is, even if two append operations on the same label are concurrent, the server will never assign their nodes the same timestamp and will never assign them a timestamp smaller than any previous append. This could possibly be achieved in practice with a clock that has high enough resolution but we provide an instantiation that does not rely on any assumption.

Our approach is to implement the timestamps using a linearizable counter $\mathsf{count_g}$. The linearizability of the counter guarantees that no two nodes get assigned the same counter and that if an append occurs before another, the former's counter value will be strictly smaller than the latter's. So nodes now have the form $\mathsf{node}_{i,\ell,j} = (v_j, \mathsf{addr}_{i,\ell,j-1}, \ell, \mathsf{count})$ and the range dictionaries RDX_i map counter values to the addresses of $\mathbf{C}_i$'s nodes in dDX that have that counter. We denote the counter of a node $\mathsf{node}_{i,\ell,j}$ as $\mathsf{count}_{i,\ell,j}$.

Updating the checkpoint dictionaries. Up to this point, our solution is correct (i.e., the unreachable nodes are now returned), sub-linear (i.e., we do not need to scan anymore) but we can still improve it. Notice that the efficiency of gets now depends on the number of nodes inserted in dDX since the checkpoint node. It follows then that the more up to date the checkpoint dictionaries are the better. To achieve this, we update the checkpoint dictionaries during get operations (as opposed to append operations). More precisely, at the end of a get the server returns the reachable nodes of $\mathsf{list}_{1,\ell}$ through $\mathsf{list}_{n,\ell}$, and all the nodes inserted after the ith checkpoint node together with their addresses. The client then parses this set of nodes and finds, for all $i \in [n]$, the (i,ℓ)-nodes with the highest counter. Note that these nodes are the heads of the lists. The client returns the heads together with their addresses to the server who can now update the checkpoint dictionaries.

5.4 Towards an Optimal Design

The construction so far has optimal append, a single round of interaction, optimal storage overhead, but the efficiency of gets is

$$\#\mathsf{MM}[\ell] + \sum_{i\in[n]} \sum_{\ell' \neq \ell} \#\mathsf{list}_{i,\ell'}\left[\mathsf{count} \geq c_{i,\ell}\right] = O\left(\#\mathsf{MM}[\ell] + \sum_{\ell'\neq\ell} \#\mathsf{list}_{\ell'}[\mathsf{count} \geq \min_{i\in[n]} c_{i,\ell}]\right)$$

where $c_{i,\ell} = \mathsf{chkcount}_{i,\ell}$ is the counter of the (i,ℓ) checkpoint node, $\mathsf{list}_{\ell'} = \cup_{i\in[n]}\mathsf{list}_{i,\ell'}$, and $\mathsf{list}[\mathsf{cond}]$ is the set of nodes in the list that satisfy the condition specified by cond. Notice that the second term in the get complexity depends on the number of non-ℓ-nodes inserted, where ℓ is the queried label. For certain workloads this can lead to non-trivial overhead so we show how to avoid it.

Skipping. To solve the issue above, we add n *skip dictionaries* $\mathsf{skDX}_1, \ldots, \mathsf{skDX}_n$ such that skDX_i maps a label ℓ to the time when an i-node was inserted last, i.e, the largest counter in an i-node. We will refer to the counters stored in skDX_i as *skip counters*. Recall that previously, during the second phase of a get the server would query the range dictionaries RDX_i for all i-nodes with counters greater than the ith checkpoint counter. Now, instead, the server queries the range dictionaries for all i-nodes with counters greater than the ith skip counter. Similar to the checkpoint dictionaries, the skip dictionaries are also updated during gets, where the client parses the set of nodes it retrieves from the server and finds for $i \in [n]$, the i-nodes with the highest counter. The client returns the counters of these nodes to the server who then updates the skip dictionaries.

Efficiency. With the changes described, the complexity of appends, the round complexity and the storage complexity remain the same, whereas the complexity of gets is

$$\#\mathsf{MM}[\ell] + \sum_{i\in[n]}\sum_{\ell'\neq\ell} \#\mathsf{list}_{i,\ell'}\,[\mathsf{count} \geq s_{i,\ell}] = O\left(\#\mathsf{MM}[\ell] + \sum_{\ell'\neq\ell} \#\mathsf{list}_{\ell'}[\mathsf{count} \geq \min_{i\in[n]} s_{i,\ell}]\right)$$

where $s_{i,\ell} = \mathsf{skcount}_{i,\ell}$ is the (i,ℓ) skip counter. Observe that for all $\ell \in \mathbb{L}_{\mathsf{MM}}$,

$$\min_{i\in[n]} \mathsf{skcount}_{i,\ell} \geq \min_{i\in[n]} \mathsf{chkcount}_{i,\ell},$$

since $\mathsf{skcount}_{i,\ell} \geq \max_{\ell\in\mathbb{L}_{\mathsf{MM}}} \mathsf{chkcount}_{i,\ell}$.

Note. We slightly modify the gets to perform the label-based filtering at the client instead of doing it at the server. While this modification does not change our asymptotics, it is going to be essential to describe the changes we are going to make due to various concurrency issues. Moreover, in the final TST protocol, we will encrypt the labels which makes server-side filtering impossible.

5.5 A Concurrent Design

So far we designed our plaintext structure without considering concurrency. We now present several challenges that come up when the structure is accessed concurrently. Recall that our goal is to achieve linearizability which essentially means that the operations should appear to be interleaved at the granularity of complete operations, and the order of non-overlapping operations should be preserved. In simpler terms, we should be able to order the operations in a way that a get should always output all the values added by the append operations ordered before it. Additionally, if an operation finishes before another one starts, the former should be ordered before the latter.

5.5.1 Need for Atomic Instructions

Consider the following scenario involving two clients $\mathbf{C}_1$ and $\mathbf{C}_2$. Suppose $\mathsf{list}_{1,\ell}$ includes 100 nodes at addresses $\mathsf{addr}_{1,\ell,1}, \ldots, \mathsf{addr}_{1,\ell,100}$ in dDX and assume the $(1,\ell)$ checkpoint address is $\mathsf{addr}_{1,\ell,1}$. If $\mathbf{C}_2$ executes a get on label ℓ it retrieves the reachable nodes of $\mathsf{list}_{1,\ell}$ and $\mathsf{list}_{2,\ell}$ as well as the nodes inserted after the $(2,\ell)$ skip counter. $\mathbf{C}_2$ will then determine that $\mathsf{node}_{1,\ell,100}$ is the head of $\mathsf{list}_{1,\ell}$ and will send $\mathsf{addr}_{1,\ell,100}$ to the server so that it can update the checkpoint dictionary cDX_1. Now assume that before the server updates $\mathsf{cDX}_1[\ell]$, the scheduler pauses the execution of the get operation and that $\mathbf{C}_1$ executes a hundred appends followed by one get all on label ℓ. The one hundred appends result in the creation of one hundred nodes with addresses $\mathsf{addr}_{1,\ell,101}, \ldots, \mathsf{addr}_{1,\ell,200}$ and the get operation results in $\mathbf{C}_1$ sending the server a new checkpoint address $\mathsf{addr}_{1,200}$ which the server will use to update the checkpoint dictionary cDX_1. If the scheduler resumes $\mathbf{C}_2$'s get operation at this moment, the server will set

$\mathsf{cDX}_1[\ell]$ to $\mathsf{addr}_{1,\ell,100}$ which is clearly wrong. For simplicity, we do not describe the updates of the skip dictionaries, but the same issue applies, where it is possible to overwrite the correct skip counter value $\mathsf{count}_{1,\ell,200}$ with an out-of-date value $\mathsf{count}_{1,\ell,100}$.

As seen, with our current design it is possible to update the checkpoint dictionaries with addresses that are out of date and the consequence is that the next get operation for ℓ will return incorrect results. This is the case because the server will first retrieve the old checkpoint address $\mathsf{addr}_{1,\ell,100}$ from cDX_1, then recover the reachable nodes of $\mathsf{list}_{1,\ell}$ from dDX starting at $\mathsf{addr}_{1,100}$. It will then retrieve the skip counter $\mathsf{count}_{1,\ell,200}$ from the skip dictionary skDX_1 and query the range dictionary RDX_1 which will return the addresses that were inserted after $\mathsf{count}_{1,\ell,200}$. The server then uses these addresses to recover the remaining nodes from dDX but this set of nodes is incorrect because it is missing the nodes with addresses between $\mathsf{addr}_{1,\ell,101}$ and $\mathsf{addr}_{1,\ell,200}$.

Efficiency issue. Now consider the scenario above except that the skip dictionaries are the ones updated with out-of-date counters instead of the checkpoint dictionaries. During a get on label ℓ, the server retrieves an old counter $\mathsf{count}_{1,\ell,100}$ from the skip dictionary instead of the correct counter $\mathsf{count}_{1,\ell,200}$. Given this counter, the server will query the range dictionary and recover the addresses added after $\mathsf{count}_{1,\ell,100}$. But notice that the server already retrieved the nodes between $\mathsf{addr}_{1,\ell,1}$ and $\mathsf{addr}_{1,\ell,200}$ since the checkpoint dictionary was updated correctly so the server could potentially retrieve a large number of unnecessary nodes just to filter them out at the end. In particular, the structure could have get efficiency

$$\#\mathsf{MM}[\ell] + \sum_{i\in[n]} \sum_{\ell'\neq\ell} \#\mathsf{list}_{i,\ell'} \left[\mathsf{count} \geq \mathsf{acount}_i\right]$$

where acount_i is the counter value of the latest i-node at the time of an arbitrarily old get operation—instead of being exactly at the time of the previous get operation as intended. In the worst case, the efficiency of the get can be linear in the size of the multi-map.

Compare and swap operations. The problem that leads to the issues above is that the time at which the server was supposed to update the checkpoint dictionary cDX_1 for $\mathbf{C}_2$'s get on ℓ and the time at which the server actually updates it are distinct and during this interval of time the server receives and executes additional append and get operations from $\mathbf{C}_1$. One possible solution is to make the server check whether the to-be-written checkpoint address is the latest one and only update the checkpoint dictionary if it is still so. This approach, however, suffers from the same issue above since the time between the check and the actual write are distinct. We solve this by making use of atomic operations and, specifically, dictionaries that support `CompareAndSwap` operations. These are dictionaries which, at a high level, execute a comparison and an update operation in one atomic step. The atomicity guarantees that there is no interruption between the comparison and the update so the approach mentioned above will solve the problem. We provide more details below.

The get operation now works the same as before, except for how the server updates the checkpoint and the skip dictionaries. We focus on the case of the checkpoint dictionaries but the same modifications apply to the skip dictionaries. When the server receives a new checkpoint address $\mathsf{addr}^\star_\ell$, it first retrieves $\mathsf{node}^\star_\ell := \mathsf{dDX}[\mathsf{addr}^\star_\ell]$. It then parses $\mathsf{node}^\star_\ell$ as $(v^\star, \mathsf{prevAddr}^\star, \ell, \mathsf{count}^\star)$ and retrieves the node at the current checkpoint address $\mathsf{addr}^\times_\ell$ which it parses as $(v^\times, \mathsf{prevAddr}^\times, \ell, \mathsf{count}^\times)$. If $\mathsf{count}^\times < \mathsf{count}^\star$, the server executes

$$\mathtt{CompareAndSwap}(\mathsf{dDX}, \ell, \mathsf{addr}^\times_\ell, \mathsf{addr}^\star_\ell).$$

If the operation fails, there was an update to the checkpoint dictionary so the server retries all the previous steps until `CompareAndSwap` is successful or until the current checkpoint address corresponds to a node that has a higher counter than the new one, i.e., when $\mathsf{count}^\star < \mathsf{count}^\times$. We refer to dictionaries that support *cas* operations as cas-dictionaries and denote such schemes as $\Delta_{\overline{\mathsf{DX}}}$.

5.5.2 Need for Locking

Recall that our goal is to design a linearizable encrypted multi-map which means that we need to ensure that all possible execution histories are linearizable. In the following, we show that our construction so far is not linearizable which leads to situations in which concurrent gets for the same label have incoherent responses; that is, neither is a subset of the other.

First, we introduce some useful terminology. Consider an append and get operation on two possibly distinct labels ℓ and ℓ', respectively, and let node_ℓ be the node inserted into the data dictionary by the append operation. We say that the get *sees the append* if the client who initiated the get on ℓ' retrieves node_ℓ. Recall that as part of a get, a client retrieves all the nodes that are reachable through the checkpoint dictionaries and the nodes with addresses returned by the range queries on the range dictionaries RDX_i, for $i \in [n]$. Moreover, we say that the get *outputs the append* if the get outputs the value of the node that was inserted by the append operation. Note that it is possible for a get to see an append but to not output it. This can occur due to the client-side filtering step when the non-ℓ nodes.

Why the structure is not linearizable. Assume there are four clients $\mathbf{C}_1, \ldots, \mathbf{C}_4$ and let $\mathsf{get}_{1,\ell}$ and $\mathsf{get}_{2,\ell}$ be two concurrent gets on label ℓ initiated by $\mathbf{C}_1$ and $\mathbf{C}_2$, respectively. For this example, we solely focus on the accesses to the range dictionaries. In particular, consider the part where $\mathsf{get}_{1,\ell}$ and $\mathsf{get}_{2,\ell}$ access the range dictionaries RDX_3 and RDX_4 corresponding to $\mathbf{C}_3$ and $\mathbf{C}_4$ in the following order: first, $\mathsf{get}_{1,\ell}$ accesses RDX_3, then $\mathsf{get}_{2,\ell}$ accesses RDX_3 and RDX_4 and, finally, $\mathsf{get}_{1,\ell}$ accesses RDX_4. We now show that the responses of the two gets are incoherent which breaks linearizability. In the example above, $\mathsf{get}_{2,\ell}$ accesses RDX_3 after $\mathsf{get}_{1,\ell}$ so it is possible that $\mathsf{get}_{2,\ell}$ sees and outputs appends from $\mathbf{C}_3$ that $\mathsf{get}_{1,\ell}$ does not see. This could happen, for instance, if $\mathbf{C}_3$ executes an append after $\mathsf{get}_{1,\ell}$ finishes reading RDX_3. Similarly, since $\mathsf{get}_{1,\ell}$ reads RDX_4

after $\mathsf{get}_{2,\ell}$, it could see and output appends from $\mathbf{C}_4$ that $\mathsf{get}_{2,\ell}$ does not see. Therefore, in this execution there are appends that one get will output but that the other does not. However, for linearization we must be able to order the two gets such that the second get outputs (at least) the responses of the first get. Since neither $\mathsf{get}_{1,\ell}$ nor $\mathsf{get}_{2,\ell}$ have responses that are a superset of the other, they cannot be ordered appropriately and, therefore, the execution is not linearizable.

Coarse-grained locking. The reason the outputs of the gets do not have the superset structure required for linearizability is that they do not synchronize on their access to the range dictionaries. In particular, the concurrent gets can access the range dictionaries in an interleaved manner which leads to the incoherent outputs described above. To synchronize the gets' accesses, we could wrap all the range dictionaries under one big lock and require the gets to acquire the lock before accessing them. This would solve the issue since there can be no interleaved accesses but now a get might need to wait until the lock is released by another concurrent get which can greatly decrease throughput.

Hand-over-hand locking. Instead of using coarse-grained locking, we solve the interleaving problem using a more granular form of locking called *hand-over-hand locking.* More precisely, given two get operations $\mathsf{get}_{1,\ell}$ and $\mathsf{get}_{2,\ell}$ on the same label ℓ, if $\mathsf{get}_{1,\ell}$ accesses RDX_i before $\mathsf{get}_{2,\ell}$, we need to ensure that $\mathsf{get}_{1,\ell}$ accesses RDX_{i+1} before $\mathsf{get}_{2,\ell}$ does, and this has to hold for all $i \in [n-1]$. Accessing the range dictionaries prevents any form of interleaving and can be achieved as follows. Instead of a single monolithic lock, we use a series of locks $\mathsf{rLock}_1, \ldots, \mathsf{rLock}_n$, one per range dictionary. When a get acquires rLock_i it then queries RDX_i but only releases the lock when it acquires the next lock RDX_{i+1}. Hand-over-hand locking leads to much better throughput.

5.5.3 Synchronizing the Gets and Appends

So far, we focused on the various synchronization issues of concurrent get operations and showed how to extend our structure with atomic instructions and hand-over-hand locking. We now highlight other synchronization issues between get and append operations which can also result in non-linearizable execution histories. Consider three clients $\mathbf{C}_1$, $\mathbf{C}_2$ and $\mathbf{C}_3$ such that $\mathbf{C}_3$ executes $\mathsf{get}_{3,\ell}$ which queries RDX_1. After accessing RDX_1 but before accessing RDX_2, $\mathbf{C}_3$'s get is paused and $\mathbf{C}_1$ executes an append $\mathsf{append}_{1,\ell}$ followed by $\mathsf{append}_{2,\ell}$ executed by $\mathbf{C}_2$. After the two appends, server resumes $\mathbf{C}_3$'s get and accesses RDX_2 where it sees and outputs $\mathsf{append}_{2,\ell}$ but not $\mathsf{append}_{1,\ell}$ since it did not see it at the time it queried RDX_1. The execution history in this example is not linearizable. To see why, observe that even though $\mathsf{append}_{1,\ell}$ precedes $\mathsf{append}_{2,\ell}$, the get outputs $\mathsf{append}_{2,\ell}$ but not $\mathsf{append}_{1,\ell}$ which violates the superset structure necessary for linearizability.

Using counters to synchronize. The problem above is due to the fact that the get and append operations are not synchronizing their accesses to the range dictionaries. More concretely, the gets have no way of knowing if they missed an append operation in a range dictionary that they have already read. Similarly to the previous problem, this issue can be solved with coarse-grained locking; specifically, by requiring that every append and get acquire a lock on all the range dictionaries. However, as discussed, this approach would affect throughput. Instead, we synchronize get and append operations with a counter as follows. The idea is to use a lineralizable counter to assign a time to a get operation in such a way that clients can distinguish between appends that are before and after the get. Recall that the appends already make use of a lineralizable counter $\mathsf{count_g}$ to timestamp the nodes. We now make use of $\mathsf{count_g}$ to generate a counter value $\mathsf{count_{get}}$ whenever a client starts executing a get. More precisely, the gets generate a counter value right before they try to acquire the lock for RDX_1. Then, if they see an append with a larger counter than $\mathsf{count_{get}}$, they discard it and do not output it. This avoids situations where a get outputs a later append, but misses an earlier one because it did not see that append. With this extension, and going back to the example above, the get operation will not see either of the appends, $\mathsf{append}_{1,\ell}$ and $\mathsf{append}_{2,\ell}$, which solves the linearizability issue.

Final details. The structure so far is almost complete except for one detail. The new counter described above and the client-side filtering may some times violate the superset structure required for linearizability.[3] Now that the counter is part of the get operation, the synchronization between the gets achieved with hand-over-hand locking is violated and a total order can no longer be attained. To solve this, we make a simple extension and wrap the counter $\mathsf{count_g}$ with a lock cLock. We also ensure that cLock and $\mathsf{rLock}_1, \ldots, \mathsf{rLock}_n$ are connected through hand-over-hand locking in the sense that a get operation first needs to acquire the cLock and then wait to acquire rLock_1, then rLock_2 and so on and so forth.

6 TST: a Linearizable Multi-Map Encryption Scheme

Our construction $\mathsf{TST} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Append})$ makes black-box use of a linearizable dictionary $\Delta_{\mathsf{DX}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put})$, a linearizable cas-dictionary $\Delta_{\mathsf{DX}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put}, \mathtt{CompareAndSwap})$, a linearizable range dictionary $\Delta_{\mathsf{RDX}} = (\mathsf{Init}, \mathsf{Put}, \mathsf{GetGreater})$, a linearizable counter $\Delta_{\mathsf{CTR}} = (\mathsf{Init}, \mathsf{FetchAndInc})$, a pseudo-random function $F : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$ and a symmetric encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. Due to space limitations, the details of the scheme are in the full version. At a high level, it works as follows.

Init. The init protocol is executed between a trusted party $\mathbf{T}$, n clients $\mathbf{C}_1, \cdots, \mathbf{C}_n$ and the server $\mathbf{S}$. All parties input the security parameter k. First, the trusted party $\mathbf{T}$ samples two keys $K_e, K_s \overset{\$}{\leftarrow} \{0,1\}^k$ and sends them to all clients. For all $i \in [n]$, client $\mathbf{C}_i$ instantiates a state dictionary $\mathsf{sDX}_i \leftarrow \Delta_{\mathsf{DX}}.\mathsf{Init}(\theta, 2k)$ that will be updated during append operations and that maps labels ℓ to a pair composed

[3] This issue is very similar to the one discussed in Section 5.5 so we do not expand on it in more detail.

of (1) the address of the head of the linked list $\mathsf{list}_{i,\ell}$ and (2) the key that encrypts the head. The server $\mathbf{S}$ initializes a data dictionary $\mathsf{dDX} \leftarrow \Delta_{\mathsf{DX}}.\mathsf{Init}(k, \theta+\lambda+3k)$. For all $i \in [n]$, $\mathbf{S}$ initializes a checkpoint dictionary $\mathsf{cDX}_i \leftarrow \Delta_{\mathsf{DX}}.\mathsf{Init}(k,k)$, a skip dictionary $\mathsf{skDX}_i \leftarrow \Delta_{\mathsf{DX}}.\mathsf{Init}(k,k)$, a range dictionary $\mathsf{RDX}_i \leftarrow \Delta_{\mathsf{RDX}}.\mathsf{Init}(k,k)$, and a range lock $\mathsf{rLock}_i := \Lambda.\mathsf{Init}(\cdot)$. The server $\mathbf{S}$ also initializes a counter lock $\mathsf{cLock} := \Lambda.\mathsf{Init}(\cdot)$. Finally, it outputs the encrypted multi-map

$$\mathsf{EMM} := \big(\mathsf{dDX}, (\mathsf{cDX}_i)_{i\in[n]}, (\mathsf{RDX}_i)_{i\in[n]}, (\mathsf{skDX}_i)_{i\in[n]}, \mathsf{count_g}, \mathsf{cLock}, (\mathsf{rLock}_i)_{i\in[n]}\big)\,.$$

Append. The append protocol is executed between a client $\mathbf{C}_i$ and the server $\mathbf{S}$. It takes as input a key K, a state st, a label ℓ and a value v from the client and the encrypted multi-map EMM from the server. It is a single-round protocol that works as follows:

- *(client)* The client first retrieves from sDX_i the address dtag_{ℓ^-} of the head of $\mathsf{list}_{i,\ell}$ and its corresponding key K_{ℓ^-}. If this is the first time $\mathbf{C}_i$ appends a value for ℓ, the entry in the state dictionary will be empty and the client sets both dtag_{ℓ^-} and K_{ℓ^-} to $\perp$. The client then samples a new data tag $\mathsf{dtag}_\ell \overset{\$}{\leftarrow} \{0,1\}^k$ and a new key $K_\ell \overset{\$}{\leftarrow} \{0,1\}^k$ and creates a new node composed of five ciphertexts
$$(\mathrm{ct}_\ell, \mathrm{ct}_v, \mathrm{ct}_{\ell^-}, \mathrm{ct}_{K_{\ell^-}}, \mathrm{ct}_{K_\ell}),$$
where $\mathrm{ct}_\ell := \mathsf{Enc}_{K_e}(\ell)$ is an encryption of the label with key K_e, $\mathrm{ct}_v := \mathsf{Enc}_{K_e}(v)$ is an encryption of the value under key K_e, $\mathrm{ct}_{\ell^-} := \mathsf{Enc}_{K_\ell}(\mathsf{dtag}_{\ell^-})$ is an encryption of the previous data tag under the new key K_ℓ, $\mathrm{ct}_{K_{\ell^-}} := \mathsf{Enc}_{K_\ell}(K_{\ell^-})$ is an encryption of previous key under the new key K_ℓ, and $\mathrm{ct}_{K_\ell} := \mathsf{Enc}_{K_e}(K_\ell)$ is an encryption of the new key under key K_e. The client then updates the state dictionary with the new data tag and key and finally sends to the server an append token $\mathsf{atk} := (\mathsf{dtag}_\ell, \mathsf{node}_{i,\ell})$ composed of the new data tag along with the encrypted node.

- *(server)* Once the server receives the append token atk, it first retrieves and increments the global counter $\mathsf{count} \leftarrow \Delta_{\mathsf{CTR}}.\mathsf{FetchAndInc}(\mathsf{count_g})$. It then appends the counter value to the encrypted node by setting $\mathsf{node}_{i,\ell} := (\mathsf{node}_{i,\ell}, \mathsf{count})$. The server then inserts the encrypted node $\mathsf{node}_{i,\ell}$ in the data dictionary at the address provided by the client by computing $\mathsf{dDX} \leftarrow \Delta_{\mathsf{DX}}.\mathsf{Put}(\mathsf{dDX}, \mathsf{dtag}, \mathsf{node})$. It also inserts the pair $(\mathsf{count}, \mathsf{dtag}_\ell)$ in the range dictionary by computing $\mathsf{RDX}_i \leftarrow \Delta_{\mathsf{RDX}}.\mathsf{Put}(\mathsf{RDX}_i, \mathsf{count}, \mathsf{dtag})$. Finally, the server outputs the updated encrypted multi-map.

Get. The get protocol is executed between a client $\mathbf{C}_i$ and the server $\mathbf{S}$. The protocol takes as input a key K, a state st, a label ℓ from $\mathbf{C}_i$ and the encrypted multi-map from $\mathbf{S}$. The get protocol is a three-round protocol that works as follows:

- *(client round 1)* The client first generates the checkpoint tag $\mathsf{ctag}_{\ell,j}$ for label ℓ and all $j \in [n]$ by computing $\mathsf{ctag}_{\ell,j} := F[K_s, \ell, j, 1]$. The client then sends to the server the get token $\mathsf{gtk}_1 := (\mathsf{ctag}_{\ell,j})_{j\in[n]}$ composed of all checkpoint tags;

- *(server round 1)* Once the server receives the first get token gtk_1, it initializes an empty set $\mathbf{R}$ and retrieves the data tag dtag_j from the checkpoint dictionary by computing, for all $j \in [n]$, $\mathsf{dtag}_j \leftarrow \mathit{\Delta}_{\mathsf{DX}}.\mathsf{Get}\big(\mathsf{cDX}_j, \mathsf{ctag}_j\big)$. If $\mathsf{dtag}_j \neq \perp$, the server retrieves the corresponding node node_j from the data dictionary by computing $\mathsf{node}_j \leftarrow \mathit{\Delta}_{\mathsf{DX}}.\mathsf{Get}\big(\mathsf{dDX}, \mathsf{dtag}_j\big)$, and appends it to $\mathbf{R}$. Note that the absence of a tag dtag_j simply means that either the jth client never appended a value for ℓ or that the ongoing get is the first get initiated by any client. The server sends the set $\mathbf{R}$ to $\mathbf{C}_i$.

- *(client round 2)* For all $j \in [n]$, the ith client parses node_j in $\mathbf{R}$ and decrypts the ciphertext of the previous key by computing $K_{\ell^-,j} := \mathsf{Dec}_{K_e}(\mathrm{ct}_{K_{\ell^-}})$. The client also computes the checkpoint tag ctag_j as well as the skip tag sktag by computing

$$\mathsf{ctag}_{\ell,j} := F[K_s, \ell, j, 1] \quad \text{and} \quad \mathsf{sktag}_{\ell,j} := F[K_s, \ell, j, 2].$$

  The client sends to the server the second get token gtk_2 composed of the old key $K_{\ell^-,j}$, the checkpoint tag $\mathsf{ctag}_{\ell,j}$ and the skip tag $\mathsf{sktag}_{\ell,j}$ for all $j \in [n]$.

- *(server round 2)* Once the server receives the second get token gtk_2, it initializes n empty sets $(\mathbf{R}_j)_{j\in[n]}$, retrieves the counter from the skip dictionaries and the data tag from the checkpoint dictionary by computing

$$\mathsf{skcount}_j \leftarrow \mathit{\Delta}_{\mathsf{DX}}.\mathsf{Get}\left(\mathsf{skDX}_j, \mathsf{sktag}_{\ell,j}\right) \text{ and } \mathsf{dtag}_j \leftarrow \mathit{\Delta}_{\mathsf{DX}}.\mathsf{Get}\left(\mathsf{cDX}_j, \mathsf{ctag}_{\ell,j}\right).$$

  The server then traverses $\mathsf{list}_{j,\ell}$ starting from the head node located at dtag_j and populates the result sets $\mathbf{R}_j$, for $j \in [n]$, as follows. It first retrieves the node from the data dictionary by computing $\mathsf{node}_j \leftarrow \mathit{\Delta}_{\mathsf{DX}}.\mathsf{Get}\left(\mathsf{dDX}, \mathsf{dtag}_j\right)$, adds the pair $(\mathsf{dtag}_j, \mathsf{node}_j)$ to $\mathbf{R}_j$, parses the node as $(\mathrm{ct}_\ell, \mathrm{ct}_v, \mathrm{ct}_{\ell^-}, \mathrm{ct}_{K_{\ell^-}}, \mathrm{ct}_{K_\ell}, \mathsf{count})$, decrypts the ciphertext of the previous data tag by computing $\mathsf{dtag}_j^- := \mathsf{Dec}_{K_{\ell^-,j}}(\mathrm{ct}_{\ell^-})$ (which becomes the new head), and decrypts the ciphertext of the previous key $K_{\ell^-,j} := \mathsf{Dec}_{K_{\ell^-,j}}(\mathrm{ct}_{K_{\ell^-}})$ (which becomes the new key). The server reiterates this process until it reaches a data tag dtag_j^- equal to $\perp$.

  The server also accesses the range dictionary to retrieve the data tags. In particular, it first waits and acquires the counter lock cLock, retrieves and increments the counter $\mathsf{count}_{\mathsf{Get}} \leftarrow \mathit{\Delta}_{\mathsf{CTR}}.\mathsf{FetchAndInc}\big(\mathsf{count}_{\mathsf{g}}\big)$ and then unlocks cLock. For all $j \in [n]$, the server then waits and acquires the range lock of the range dictionary rLock_j, retrieves all the data tags that have a counter larger than $\mathsf{skcount}_j$ such that $\mathbf{r}_j \leftarrow \mathit{\Delta}_{\mathsf{RDX}}.\mathsf{GetGreater}\left(\mathsf{RDX}_j, \mathsf{skcount}_j\right)$, releases the lock, and then retrieves all the nodes from the data dictionary located at the corresponding data tags. Note that the acquisition and release of the lock follows a hand-over-hand locking mechanism (refer to Section 5.5 for more details). In particular, for all $\mathsf{dtag} \in \mathbf{r}_j$, the server computes $\mathsf{node} \leftarrow \Delta_{\mathsf{DX}}.\mathsf{Get}\,(\mathsf{dDX}, \mathsf{dtag})$ and adds $(\mathsf{dtag}, \mathsf{node})$ to $\mathbf{R}_j$. Finally, the server sends to the client $(\mathbf{R}_j)_{j\in[n]}$ along with the get counter $\mathsf{count}_{\mathsf{Get}}$.

– *(client round 3)* In this round, the client filters the nodes and only keeps the ones that need to be part of final response. The client also computes, for all $j \in [n]$, the new head $\mathsf{dtag}^{\star}_j$ of the linked list to be stored in the checkpoint dictionary and the most recent counter $\mathsf{skcount}_j$ to be stored in the skip dictionary. To filter the nodes, it initializes a set $\mathbf{v}$ and for all $j \in [n]$, performs the following steps. For all $(\mathsf{dtag}_z, \mathsf{node}_z) \in \mathbf{R}_j$, it parses the node as $(\mathrm{ct}_{\ell,z}, \mathrm{ct}_{v,z}, \mathrm{ct}_{\ell^-,z}, \mathrm{ct}_{K_{\ell^-,z}}, \mathrm{ct}_{K_\ell,z}, \mathsf{count}_z)$ and decrypts the ciphertext of the label by computing $\ell_z := \mathsf{Dec}_{K_e}(\mathrm{ct}_{\ell,z})$. If $\ell_z = \ell$ and count_z is smaller than $\mathsf{count}_{\mathsf{Get}}$, the client computes $v := \mathsf{Dec}_{K_e}(\mathrm{ct}_{\ell,z})$ and adds it to $\mathbf{v}$. Note that the second condition is necessary to synchronize between the get and the append operations which is crucial for linearizability as discussed in Section 5.5. As a second step, client computes the new head of the linked list, $\mathsf{dtag}^{\star}_j$, by first identifying the node for label ℓ with the largest counter and then setting $\mathsf{dtag}^{\star}_j := \mathsf{dtag}_{z^\star}$, $\mathsf{chkcount}_j := \mathsf{count}_{z^\star}$, where

$$z^\star := \arg\max_{z\in\mathbf{Z}}\,(\mathsf{count}_z) \quad \text{and} \quad \mathbf{Z} = \{z \in [|\mathbf{R}_j|] : \ell_z = \ell\}.$$

For the skip counter, the client needs to identify the node with the largest counter irrespective of the underlying label, i.e., $\mathsf{skcount}_j := \max_z \big(\mathsf{count}_z\big)$. The client also computes the checkpoint tag as well as the skip tag

$$\mathsf{ctag}_{\ell,j} := F[K_s, \ell, j, 1] \quad \text{and} \quad \mathsf{sktag}_{\ell,j} := F[K_s, \ell, j, 2],$$

which are necessary to update the jth checkpoint dictionary and the jth skip dictionary. Finally, the client sends the third get token gtk_3 which is composed of the checkpoint tag $\mathsf{ctag}_{\ell,j}$, the checkpoint counter $\mathsf{chkcount}_j$ and the new data tag $\mathsf{dtag}^{\star}_j$ which is the address of the new head of the linked list. The token also includes of the skip counter $\mathsf{skcount}_j$ as well as the skip tag $\mathsf{sktag}_{\ell,j}$, for all $j \in [n]$.

– *(server round 3)* Once the server receives the third get token, it updates the checkpoint dictionary as well as the skip dictionary. In particular, for all $j \in [n]$, the server first retrieves the old data tag by computing $\mathsf{dtag}^{\times}_j \leftarrow \Delta_{\mathsf{DX}}.\mathsf{Get}\big(\mathsf{cDX}_j, \mathsf{ctag}_j\big)$. It then retrieves the corresponding node from the data dictionary from which it extracts the counter count_j. The server only updates the checkpoint dictionary if the old counter count_j is strictly smaller than the new checkpoint counter $\mathsf{chkcount}_j$. For this, it makes use of the compare and swap atomic instruction $\mathtt{CompareAndSwap}(\mathsf{cDX}_j, \mathsf{ctag}_j, \mathsf{dtag}^{\times}_j, \mathsf{dtag}^{\star}_j)$, so that $\mathsf{cDX}_j[\mathsf{ctag}_j]$ is updated to $\mathsf{dtag}^{\star}_j$ if and only if $\mathsf{cDX}_j[\mathsf{ctag}_j] = \mathsf{dtag}^{\times}_j$. If the $\mathtt{CompareAndSwap}$ fails, the server performs the same steps as above until $\mathsf{count}_j \geq \mathsf{chkcount}_j$. It performs the same steps to update the skip dictionary; i.e., for all $j \in [n]$, it computes $\mathtt{CompareAndSwap}(\mathsf{skDX}_j, \mathsf{sktag}_j, \mathsf{count}^{\times}_j,$

$\mathsf{skcount}_j)$, where $\mathsf{count}_j^{\times}$ is the old counter in the skip dictionary such that $\mathsf{count}_j^{\times} \leftarrow \Delta_{\mathsf{DX}}.\mathsf{Get}\left(\mathsf{skDX}_j, \mathsf{sktag}_j\right)$.

The client finally outputs the final response $\mathbf{v}$ whereas the server outputs the updated encrypted multi-map EMM.

Making TST *fully-dynamic.* TST can be extended to support delete operations using *lazy deletion* which has been used in many dynamic multi-map encryption constructions [15,8,9,28,59]. The lazy deletion of a label/value pair (ℓ, v) is implemented using an append of (ℓ, v) with an additional delete marker. During gets, the client retrieves both pairs and uses the delete markers to filter out the deleted values. In the context of TST, the client retrieves the nodes of the appended and deleted pairs and filters out the deleted values as follows. For each append node with value v, if there is a delete node with value v but a greater counter, then it removes v from the response; otherwise, it keeps v. The construction can be made linearizable with a minor change to the append protocol on the server side. When a client $\mathbf{C}_i$ sends a new node to the server, it uses hand-over-hand locking over cLock and rLock_i to increment count_g and to update RDX_i. Unfortunately, this makes appends and deletes deadlock-free instead of lock-free. All the operations linearize at the time they lock cLock. Due to space constraints, the proof will appear in the full version of this work. We leave it as future work to design a linearizable lock-free fully-dynamic encrypted multi-map.

7 Efficiency, Linearizability and Security of TST

Efficiency analysis. Due to space constraints, we defer the efficiency analysis of TST (including time, storage, round complexity, and progress guarantees) to the full version. We first analyze the asymptotic behavior of TST in a black-box manner, and then, examine its concrete efficiency by considering specific instantiations of the underlying plaintext data structures. Additionally, we investigate both the worst-case and best-case scenarios for the get complexity.

Linearizability. We show that TST is linearizable. In particular, we first introduce a linearizable procedure LZP (in the full version) which defines the linearization points for all possible execution histories H in TST. As a second step, given the output of the linearizable procedure we prove that TST verifies both the *span membership* and the *correctness* conditions described in Definition 2. While proving span membership is relatively straightforward, proving correctness is more challenging as it requires showing that for all append operations append_ℓ and get operations get_ℓ for label ℓ in the history H, the inequality, $\mathsf{linp}(\mathsf{append}_\ell) < \mathsf{linp}(\mathsf{get}_\ell)$ holds if and only if the appended value in append_ℓ is part of the output of the get operation get_ℓ. Conversely, if the append operation is not part of the output of the get operation, then append_ℓ should be linearized after get_ℓ. Due to space constraints, we defer the details to the full version, and only state the main result here.

Corollary 1. *If Δ_{DX}, $\Delta_{\overline{\mathsf{DX}}}$, Δ_{RDX} and Δ_{CTR} are linearizable, then all execution histories H are linearizable and, therefore,* TST *is linearizable.*

Security. We describe TST's instruction-level leakage $\mathcal{L}$. During an append operation, there is no leakage. During get operations, TST leaks the operation equality pattern, i.e., correlations between operations on the same label. In the first round of server communication, when the server receives the first get token gtk_1, it infers correlations with get operations that have the same label and for which the server has also received the first get token. This is because the first get token for two gets that query the same label will be the same. Next, during the third round, when the server receives new checkpoint addresses in gtk_3, it learns new correlations between the current get and append operations. These correlations are with the appends that added their counters to the range dictionary before the get operation read that range dictionary. This is because when the server sees the new checkpoint address, it learns that all these nodes have the same label as the current get operation. Due to space constraints, a formal and detailed description of the leakage profile $\mathcal{L}$ and proof of the theorem are in the full version.

Theorem 1. *If F is pseudo-random,* SKE *is a CPA-secure, and Δ_{DX}, $\Delta_{\overline{\mathsf{DX}}}$, Δ_{RDX}, and Δ_{CTR} are all linearizable, then* TST *is $(\chi_{\mathsf{lz}}, \mathcal{L})$-secure in the random oracle model.*

Note. In the sequential setting, our construction achieves *forward privacy* which has been extensively studied in the STE literature and is the standard security goal for dynamic encrypted multi-maps. As pointed out in [63], forward privacy can protect against certain injection attacks but recent work [4] has shown that forward privacy has some limitations. In the concurrent setting, there are no standard security notions for leakage profiles and, as far as we know, there are no concurrent-specific leakage attacks (e.g., that also exploit adversarial scheduling), therefore we do not have a baseline for comparison other than the one in the sequential setting.

References

1. Agarwal, A., Kamara, S.: Encrypted distributed hash tables. Tech. Rep. 2019/1126, IACR ePrint Cryptography Archive (2019), https://eprint.iacr.org/2019/1126.pdf
2. Agarwal, A., Kamara, S.: Encrypted key-value stores. In: Progress in Cryptology–INDOCRYPT 2020: 21st International Conference on Cryptology in India, Bangalore, India, December 13–16, 2020, Proceedings 21. pp. 62–85. Springer (2020)
3. Amjad, G., Kamara, S., Moataz, T.: Structured encryption secure against file injection attacks (2021), (under submission at CRYPTO '21)
4. Amjad, G., Kamara, S., Moataz, T.: Injection-secure structured and searchable symmetric encryption. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 232–262. Springer (2023)

5. Asharov, G., Naor, M., Segev, G., Shahaf, I.: Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In: ACM Symposium on Theory of Computing (STOC '16). pp. 1101–1114. STOC '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2897518.2897562, http://doi.acm.org/10.1145/2897518.2897562
6. Asharov, G., Komargodski, I., Lin, W.K., Peserico, E., Shi, E.: Optimal oblivious parallel RAM. In: Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 2459–2521. SIAM (2022)
7. Asharov, G., Segev, G., Shahaf, I.: Tight tradeoffs in searchable symmetric encryption. In: Annual International Cryptology Conference. pp. 407–436. Springer (2018)
8. Bost, R.: Sophos - forward secure searchable encryption. In: ACM Conference on Computer and Communications Security (CCS '16) (2016)
9. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: ACM Conference on Computer and Communications Security (CCS '17) (2017)
10. Boyle, E., Chung, K.M., Pass, R.: Oblivious parallel RAM and applications. In: Theory of Cryptography Conference. pp. 175–204. Springer (2015)
11. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. ACM Sigplan Notices **45**(5), 257–268 (2010)
12. Canetti, R.: Security and composition of multi-party cryptographic protocols. Journal of Cryptology **13**(1) (2000)
13. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Advances in Cryptology - CRYPTO '13. Springer (2013)
14. Cash, D., Tessaro, S.: The locality of searchable symmetric encryption. In: Advances in Cryptology - EUROCRYPT 2014 (2014)
15. Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation. In: Network and Distributed System Security Symposium (NDSS '14) (2014)
16. Cash, D., Ng, R., Rivkin, A.: Improved structured encryption for sql databases via hybrid indexing. In: Applied Cryptography and Network Security: 19th International Conference, ACNS 2021, Kamakura, Japan, June 21–24, 2021, Proceedings, Part II. pp. 480–510. Springer (2021)
17. Chan, T.H.H., Chung, K.M., Shi, E.: On the depth of oblivious parallel RAM. In: Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23. pp. 567–597. Springer (2017)
18. Chan, T.H.H., Guo, Y., Lin, W.K., Shi, E.: Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In: Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23. pp. 660–690. Springer (2017)
19. Chan, T.H.H., Nayak, K., Shi, E.: Perfectly secure oblivious parallel RAM. In: Theory of Cryptography Conference. pp. 636–668. Springer (2018)
20. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Advances in Cryptology - ASIACRYPT '10. Lecture Notes in Computer Science, vol. 6477, pp. 577–594. Springer (2010)
21. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. Tech. Rep. 2011/010.pdf, IACR Cryptology ePrint Archive (2010)

22. Chen, B., Lin, H., Tessaro, S.: Oblivious parallel RAM: improved efficiency and generic constructions. In: Theory of Cryptography: 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II 13. pp. 205–234. Springer (2016)
23. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: Improved definitions and efficient constructions. In: ACM Conference on Computer and Communications Security (CCS '06). pp. 79–88. ACM (2006)
24. Demertzis, I., Papamanthou, C.: Fast searchable encryption with tunable locality. In: ACM International Conference on Management of Data (SIGMOD '17). pp. 1053–1067. SIGMOD '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3035918.3064057, http://doi.acm.org/10.1145/3035918.3064057
25. Demertzis, I., Papadopoulos, D., Papamanthou, C.: Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In: Advances in Cryptology - CRYPTO '18. pp. 371–406. Springer (2018)
26. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing. pp. 131–140 (2010)
27. Ellis, C.S.: Concurrency in linear hashing. ACM Transactions on Database Systems (TODS) **12**(2), 195–217 (1987)
28. Etemad, M., Küpccü, A., Papamanthou, C., Evans, D.: Efficient dynamic searchable encryption with forward privacy. PoPETs **2018**(1), 5–20 (2018). https://doi.org/10.1515/popets-2018-0002, https://doi.org/10.1515/popets-2018-0002
29. Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., Steiner, M.: Rich queries on encrypted data: Beyond exact matches. In: European Symposium on Research in Computer Security (ESORICS '15). Lecture Notes in Computer Science. vol. 9327, pp. 123–145 (2015)
30. Garg, S., Mohassel, P., Papamanthou, C.: TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In: Advances in Cryptology - CRYPTO 2016. pp. 563–592 (2016). https://doi.org/10.1007/978-3-662-53015-3_20, https://doi.org/10.1007/978-3-662-53015-3_20
31. George, M., Kamra, S., Moataz, T.: Structured encryption and dynamic leakage suppression. In: Advances in Cryptology - EUROCRYPT 2021 (2021)
32. Greenwald, M.: Two-handed emulation: how to build non-blocking implementations of complex data-structures using dcas. In: Proceedings of the twenty-first annual symposium on Principles of distributed computing. pp. 260–269 (2002)
33. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: ACM Conference on Computer and Communications Security (CCS '14). pp. 310–320. CCS '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2660267.2660297, http://doi.acm.org/10.1145/2660267.2660297
34. Herlihy, M., Shavit, N., Luchangco, V., Spear, M.: The art of multiprocessor programming. Newnes (2020)
35. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS) **12**(3), 463–492 (1990)
36. Hsu, M., Yang, W.P.: Concurrent operations in extendible hashing. In: VLDB. vol. 86, pp. 25–28 (1986)
37. Hubert Chan, T.H., Shi, E.: Circuit OPRAM: Unifying statistically and computationally secure orams and oprams. In: Theory of Cryptography Conference. pp. 72–107. Springer (2017)
38. Kamara, S., Moataz, T.: Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: Advances in Cryptology - EUROCRYPT '17 (2017)

39. Kamara, S., Moataz, T.: SQL on Structurally-Encrypted Data. In: Asiacrypt (2018)
40. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Financial Cryptography and Data Security (FC '13) (2013)
41. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: ACM Conference on Computer and Communications Security (CCS '12). ACM Press (2012)
42. Kamara, S., Moataz, T.: Design and analysis of ost. Tech. rep., MongoDB (2022)
43. Kamara, S., Moataz, T., Ohrimenko, O.: Structured encryption and leakae suppression. In: Advances in Cryptology - CRYPTO '18 (2018)
44. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman & Hall/CRC (2008)
45. Kim, K.S., Kim, M., Lee, D., Park, J.H., Kim, W.H.: Forward secure dynamic searchable symmetric encryption with efficient updates. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1449–1463 (2017)
46. Korenfeld, B.: CBTree: a Practical Concurrent Self-adjusting Search Tree. University of Tel-Aviv (2012)
47. Kumar, V.: Concurrent operations on extendible hashing and its performance. Communications of the ACM **33**(6), 681–694 (1990)
48. Lamport: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE transactions on computers **100**(9), 690–691 (1979)
49. Lea, D.: Hash table util. concurrent. concurrenthashmap, revision 1.3. JSR-166, the proposed Java Concurrency Package (2003)
50. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures. pp. 73–82 (2002)
51. Microsoft: ConcurrentDictionary.TryUpdate Method. https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2.tryupdate?view=net-8.0, accessed: September 16, 2024
52. Miers, I., Mohassel, P.: Io-dsse: Scaling dynamic searchable encryption to millions of indexes by improving locality. Cryptology ePrint Archive, Report 2016/830 (2016), http://eprint.iacr.org/2016/830
53. Nayak, K., Katz, J.: An oblivious parallel RAM with $O(log^2 N)$ parallel runtime
54. Oracle: ConcurrentMap.computeIfPresent Method. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html#computeIfPresent-K-java.util.function.BiFunction-, accessed: September 16, 2024
55. Pappas, V., Krell, F., Vo, B., Kolesnikov, V., Malkin, T., Choi, S.G., George, W., Keromytis, A., Bellovin, S.: Blind seer: A scalable private dbms. In: Security and Privacy (SP), 2014 IEEE Symposium on. pp. 359–374. IEEE (2014)
56. Popovitch, G.: parallel-hashmap: modify_if() function. https://github.com/greg7mdp/parallel-hashmap/tree/8a889d3699b3c09ade435641fb034427f3fd12b6, accessed: September 16, 2024
57. Shalev, O., Shavit, N.: Split-ordered lists: Lock-free extensible hash tables. Journal of the ACM (JACM) **53**(3), 379–405 (2006)
58. Song, D., Wagner, D., Perrig, A.: Practical techniques for searching on encrypted data. In: IEEE Symposium on Research in Security and Privacy. pp. 44–55. IEEE Computer Society (2000)

59. Song, X., Dong, C., Yuan, D., Xu, Q., Zhao, M.: Forward private searchable symmetric encryption with optimized i/o efficiency. IEEE Transactions on Dependable and Secure Computing **17**(5), 912–927 (2018)
60. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: Network and Distributed System Security Symposium (NDSS '14) (2014)
61. Triplett, J., McKenney, P.E., Walpole, J.: Resizable, scalable, concurrent hash tables via relativistic programming. In: 2011 USENIX Annual Technical Conference (USENIX ATC 11) (2011)
62. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing. pp. 214–222 (1995)
63. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: USENIX Security Symposium (2016)