Private Database Access With HE-over-ORAM Architecture

Craig Gentry, IBM Research
Shai Halevi, IBM Research
Charanjit Jutla, IBM Research
Mariana Raykova, SRI International

May 16, 2014

Abstract

Enabling private database queries is an important and challenging research problem with many real-world applications. The goal is for the client to obtain the results of its queries without learning anything else about the database, while the outsourced server learns nothing about the queries or data, including access patterns. The secure-computation-over-ORAM architecture offers a promising approach to this problem, permitting sub-linear time processing of the queries (after pre-processing) without compromising security.

In this work we examine the feasibility of this approach, focusing specifically on secure-computation protocols based on somewhat-homomorphic encryption (SWHE). We devised and implemented secure two-party protocols in the semi-honest model for the path-ORAM protocol of Stefanov et al. This provides access by index or keyword, which we extend (via pre-processing) to limited conjunction queries and range queries. Some of our sub-protocols may be interesting in their own right, such as our new protocols for encrypted comparisons and blinded permutations.

We implemented our protocols on top of the `HElib` homomorphic encryption library. Our basic single-threaded implementation takes about 30 minutes to process a query on a database with 2^{22} records and 120-bit long keywords, providing a cause for optimism about the viability of this direction, and we expect a better optimized implementation to be much faster.

Keywords. Comparison Protocols, Homomorphic Encryption, ORAM, PIR, Private Queries, Secure Computation

Supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center (DoI/NBC) contract number D11PC20202. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

1 Introduction

The recent explosive growth of data outsourcing raises the issue of privacy guarantees for the outsourced data. While encryption can protect the *content* of the outsourced data, it remains a challenging problem to *access* the data privately. Since it is often possible to deduce important information from the access pattern alone (see e.g., [16] for some examples), it is important to hide also the access pattern from the server.

Solutions for hiding the access pattern include the oblivious RAM (ORAM) of Goldreich and Ostrovsky [12] and private information retrieval (PIR) of Chor et al. [5]. Recent years saw a surge in the level of interest and volume of new work in this area, addressing better efficiency, increased functionality, new threat models, and more. Roughly speaking, solutions can be categorized as either PIR-like protocols that inherently work in linear time in the size of the database, or ORAM-based solutions that have linear-time pre-processing but sub-linear access time (at the price of keeping some secret storage at the client). The current work is of the latter type.

The problem of private queries becomes even harder in situations where we need to ensure that the client also does not learn too much. Below we sometimes refer to this setting as *symmetric* private queries (borrowing the terminology from symmetric-PIR). For example, consider an organization that wants to maintain its internal access-control policy for the data, even when this data it outsourced to the cloud. In this case it is not enough to require that the cloud provider does not learn anything about the data. We must also ensure that an individual client from the organization who queries the database only gets the data that it asked for (and was authorized to obtain[1]), and the access protocol does not inadvertently leak anything else about the data. Similar concerns arise for a government organization setting up a server with need-based access for its clients.

1.1 Previous and Concurrent Work

A promising direction for addressing (symmetric) private-query is the secure-computation-over-ORAM architecture of Ostrovsky and Shoup [21] and Gordon et al. [13]. Here the client and server use secure two-party protocols to simulate the actions of an underlying ORAM protocol. This way we can keep the sub-linear access time of the underlying ORAM, while ensuring that the parties do not learn anything beyond the output of the original protocol, i.e., the server learns nothing and the client only learns the answer to its query.

In [21, 13], this architecture was proposed as a solution for generic multi-party computation in RAM complexity, i.e., without having to transform the original insecure RAM computation into a binary circuit. The first implementation of a system along this line was due to Gordon et al. [13], using Yao-circuit-type two party protocols over the tree-ORAM of Shi et al. [24]. Gentry et al. later proposed a few optimizations for the underlying ORAM scheme [9], and also suggested to utilize low-degree homomorphic encryption for the two-party protocols over this ORAM, but did not implement any of these protocols. Recently Stefanov et al. [25] proposed the Path-ORAM protocol, which is a variant of tree-ORAM with better asymptotic efficiency.

Very recently, Liu et al. [20] developed an automated compiler for secure two-party computation, using the Gordon et al. architecture of Yao-based protocols over tree-ORAM (with many optimizations). Also, Keller and Scholl [18] extended the secure-computation-over-ORAM architecture to handle any number $n \geq 2$ of parties. They use the SPDZ framework [8] (with protocols based on algebraic-black-box approach with preprocessing) and use both tree-ORAM and path-ORAM as the underlying ORAM schemes. (The work of Keller and Scholl is concurrent to ours.)

Along a different direction, many recent works aimed at achieving extremely high speed by somewhat compromising privacy, leaking a small amount of information about the access pattern. Some notable examples of work along this direction is the CryptDB system of Popa et al. [23], and recent works on searchable symmetric encryption due to Pappas et al. [22] Cash et al. [4], and Jarecki et al. [17].

1.2 This Work

In this work we designed and implemented a system for symmetric private queries in the semi-honest adversary model, supporting private database access by either index or keyword. We focus on exploring the feasibility of the direction advocated by Gentry et al. [9], of using secure-computation

[1]This report only covers the implementation of the private query protocols themselves, we briefly comment on the related authorization issue in Appendix A.

protocols based on low-degree homomorphic encryption over the tree-ORAM scheme. Specifically, we used for the underlying ORAM a slight modification of the Path-ORAM protocol of Stefanov et al. [25], and implemented our two-party computation protocols based on the `HElib` homomorphic-encryption library [15].

Our results show cause for optimism regarding the feasibility of this direction: Our single-threaded implementation can query a moderate-size database with 2^{22} records on a 120-bit keyword in just over 30 minutes. This indicates that SWHE-based protocols are not as slow as commonly believed. Moreover there is a wide range of further optimizations that can be applied (both algorithmic and implementation-level), and we expect a better optimized system to be one to three orders of magnitude faster (see discussion in Section 5). In this report we describe all the sub-protocols that went into our implementation, and also describe some extensions of the basic system to support range queries, authorization, and even provide limited support for conjunctions via pre-processing.

Our work is similar in many ways to the concurrent work of Keller and Scholl [18]. In particular they also developed secure-computation-over-ORAM protocols for arrays (access-by-index) and dictionaries (access-by-keyword). Some important differences between our work and [18] include the following:

- Keller and Scholl target generic multiparty secure computation rather than data outsourcing. In particular in their system all the parties need to keep state as large as all of the data (since they use secret-sharing to share the entire state). Also the current work includes extensions that are more specific for data outsourcing such as range queries, conjunctive queries, and authorization.
- The protocols in [18] are all in the "algebraic black-box model" (using the SPDZ framework) while ours use SWHE as the basic tool. As we discuss below, introducing new SWHE-based secure protocols is one of the contributions of the current work.

We also note that our performance numbers cannot be directly compared to those from [18], since they only report the online numbers and not the "expensive" offline computations that are done by the SPDZ framework.

1.2.1 SWHE-based Secure Computation

Beyond the specific application of private queries, another contribution of the current work is in developing several new SWHE-based secure computation protocols that are interesting on their own.

Encrypted Equal-to-Zero and Comparisons. Comparing encrypted numbers is a common low-level task in many cryptographic protocols, and significant effort was invested in optimizing it, see e.g., [6, 26, 27, 19]. In our context, we need the result to be encrypted, i.e. we want the end result to be an encryption of the answer bit, zero or one.

In the simplest setting, we would like to transform an encryption of an n-bit value x into an encryption of a bit b such that $b = 0$ if $x = 0$ and $b = 1$ if $x \neq 0$. Computing b homomorphically from x without any interaction requires homomorphic degree roughly 2^n, or we can use a single communication round to get an encryption of the individual bits of x, and then can use degree-n homomorphism to compute the answer. But we can actually do much better. In Section 3.1 we describe a protocol that *uses only additive homomorphism*, works in $\log^* n$ communicating rounds, and requires $O(n)$ homomorphic addition operations. Moreover using batching techniques, this protocol can be implemented with only $O(\log n)$ additions and shifts. The end result has complexity $\tilde{O}(n + k)$ (with k the security parameter), which is asymptotically more efficient than previous protocols in the literature.

Our protocol relies on the flexibility of contemporary lattice-based encryption schemes that enable additive homomorphism relative to arbitrary moduli. The core of our new equal-to-zero protocol is a one-message sub-protocol that transforms the encryption of the n-bit x into an encryption of a $\log n$-bit y such that $y = 0$ if and only if $x = 0$. This size-reduction protocol uses the fact that an n-bit value is equal to zero if and only if the sum of its bits is zero, when using homomorphism modulo $m > n$. Applying the size-reduction protocol $\log^* n$ times reduces the problem to a constant-size instance, which we can solve using any of the existing techniques.

We also describe in Section 3.2 a protocol for comparing encrypted numbers, where on inputs x, y we obtain an encryption of a bit b such that $b = 1$ if $y > x$ and $b = 0$ otherwise. This protocol uses n parallel executions of the equal-to-zero protocol on $\log n$-bit values, and some local computation using additive homomorphism. Hence, it too takes $\log^* n$ rounds, and using ciphertext-packing can be made to run in complexity quasi-linear in $n + k$ (with k the security parameter).

The basic comparison protocol from Section 3.2 requires that we have encryptions of the separate bits of the numbers that we compare, but in our application one of these numbers comes from long-term storage and storing its encrypted bits would entail a somewhat large plaintext-to-ciphertext expansion ratio. Hence, we also describe in Section 3.2 another optimization that allows us to encrypt this number as a single integer (or a sequence of integer digits), so long as the integer(s) are stored in *reverse bit order.*

Blinded Permutation. This protocol, described in Section 3.3, allows two parties to shuffle obliviously an array. The input to this protocol is an encrypted array a and an encrypted permutation p, and the output is the encryption of the permuted array, namely a' such that $a'[p[i]] = a[i]$. The main idea of this protocol is that the server can "blind" the permutation p by permuting it randomly with another random permutation q that it knows, then send it to the client for decryption. The client decrypts and gets $q \circ p$, uses it to permute the array a and returns it to server, who now permutes by q^{-1} to get the final result. (Of course, more blinding is needed also to hide a from the client.)

Security. The security property of all these protocols (in the semi-honest model) asserts that neither party learns anything during the execution of these protocols. That is, the view of each party consists only of ciphertexts under the other party's key and of random plaintext elements that are encrypted under it own key. (Hence the entire view can be simulated without knowledge of the encrypted values.)

Different flavor of protocols. Our equal-to-zero and comparison protocols are in some ways quite different than existing protocols in the literature: almost all HE-based protocols in the literature can be described in the arithmetic black-box model [7]. In that model there is an algebraic ring which is shared among parties, and sub-protocols for operations in the ring as used as the basis for everything else. (Usually the overriding complexity measure is the number of invocations of the ring operations.)

Our equal-to-zero protocol is different: while only using additive homomorphism, it does not fit in the algebraic black-box model since it relies on an interplay between different algebraic rings to get better efficiency. This approach, coupled with the ability to compute locally low-degree functions (not just linear), makes SWHE a very useful tool for designing efficient protocols.

Building secure-computation protocols based on SWHE is a new research direction, whereas protocols based on Yao circuits or additive-HE schemes have been investigated and optimized for over two decades. This work helps lay the groundwork for SWHE-based protocols, which are sure to find more uses.

1.2.2 Our Implementation

We implemented our private query solution with all its sub-protocols over the `HElib` software library [15]. We built our implementation to handle a moderate-size database of a few million entries.

Specifically, our choice of parameters for this implementation can handle a database of up to 2^{24} records, with keywords of up to 120 bits.[2]

We tested it on the equivalent of a 2^{22}-record database with 120-bit keywords, running on a five-year-old IBM BladeCenter HS22/7870, with two Intel X5570 (4-core) processors, running at 2.93GHz. However, one consequence of using `HElib` is that our implementation is inherently single-threaded (since `HElib` is not thread-safe), so we only utilized one of the eight cores available on that machine. Processing a single access-by-keyword request took over 32 minutes, of which just under three minutes were devoted to obtaining the information itself, and the rest for maintenance operations (i.e., updating the ORAM trees and running the eviction protocol). As we said above, we expect that a better-optimized implementation would be able to do much better (even if we don't count the $8\times$ speedup that one could get from just using all eight cores). We describe some possible optimizations in Section 5.

2 Background

2.1 The Path-ORAM Protocol

In the basic path-ORAM protocol [25], the server keeps an N-element database in a complete binary tree of height $h = \log N$, where each node in the tree contains a bucket large enough to store a small constant number Z of data elements. In addition there is also a moderate-size stash of S entries to keep elements that do not fit elsewhere (we think of the stash as being kept at the root of the tree). The content of all the buckets is encrypted under the client's key, in particular the server does not know how many elements are actually stored in each bucket.

Each database element with logical address $v \in [N]$ is associated with a random leaf L_v, and the client keeps an N-entry table of the mapping $v \mapsto L_v$. (I.e., entry v in the table contains the leaf number L_v.)

Denote by d_v the data corresponding to logical address v. The protocol maintains the invariant that the triple (L_v, v, d_v) is stored in one of the buckets on the path from the root to the leaf L_v. Access to logical address v consists of two subroutines, one for doing the actual access and another one to clean up after the access.

Access. To access the data in logical address v, the client looks up L_v in its table and asks the server for the entire path from the root to leaf L_v. Upon receiving all the buckets in this path, the client decrypts them, finds a triple of the form (L_v, v, d_v) in one of the buckets, and this value d_v is the requested data.

The client either leaves the data unchanged (if the operation is a read) or overwrites it with a new value (if it is a write). We denote the resulting data by d'_v. In either case, it chooses a new random leaf $L'_v \in [N]$ and updates its table with the new L'_v value. The client then removes the triple (L_v, v, d_v) from the bucket where it was found, and puts the triple (L'_v, v, d'_v) in the root bucket. Finally it re-encrypts all the buckets and send them back to the server, who replaces all the buckets on the path to L_v by the new encrypted buckets. Since the new triple is placed at the root, this operation maintains the tree invariant of the scheme.

Eviction. To prevent the root bucket from overflowing, the client and server run a "maintenance" subroutine whose goal is to evict triples from their current buckets and push them lower down the tree: The client and server agree on some "eviction path" (in [25] this is the same as the read path), and each entry in that path $e_i = (L_i, v_i, d_i)$ is pushed as far down that path as it can go toward its target leaf L_i. The stash is used to avoid over-filling the buckets (with conflicts resolved greedily).

[2]Both of these restrictions eventually stem from working with packed ciphertexts over the 6361'st cyclotomic field, which have 120 plaintext slots.

It is easy to see that as long as the stash does not overflow, the view of the server is computationally independent of the access pattern (assuming the security of the encryption scheme). Stefanov et al. proved in [24] that when using the read path for eviction and setting $S = O(\log N)$, the probability of the stash overflowing is negligible. In our implementation we instead use the deterministic eviction strategy that was proposed by Gentry et al. in [9]. We run experiments and found that this deterministic strategy allows us to use smaller buckets, namely only $Z = 2$ as opposed to $Z = 4$ which is needed when evicting along the read-path.

2.1.1 Putting it Together

In the complete construction, the ORAM also stores the mapping $v \mapsto L_v$. Specifically, the server keeps $\ell = \lceil \log(N) \rceil$ complete binary trees as above, with the level-i tree having $2^{\ell-i}$ leaves. In the largest tree ($i = 0$), each entry corresponds to one logical address $v \in \{0, \ldots, N-1\}$, and it contains the user data for that logical address. For the next tree ($i = 1$), each entry corresponds to two consecutive logical addresses, and it contains the two leaf-numbers in the largest tree that are currently assigned to those logical addresses. More generally, each entry in the tree at level $i+1$ corresponds to the union of two level-i intervals (which is altogether a size-2^{i+1} interval of logical addresses), and that entry contains two leaf-numbers of the level-i tree, namely the leaves that are currently assigned to the entries of those two level-i intervals. With each entry in every tree we store also the first logical address of the interval of that entry, as well as the leaf that is currently assigned to that entry (in the current tree). Thus each entry is of the form

$$\begin{array}{ll} \text{level } 0: & \left(\begin{array}{lll} L^*, & v, & \text{user-data} \end{array} \right) \\ \text{level} > 0: & \left(\begin{array}{lll} L^*, & v, & L_1, L_2 \end{array} \right) \end{array}$$

where L^* is the leaf currently assigned to that entry, $[v, v+2^i)$ is its interval, and (L_1, L_2) are the leafs in the next tree that are currently assigned to the two sub-intervals $[v, v+2^{i-1})$, $[v+2^{i-1}, v+2^i)$. Of course, all of the buckets in all of the trees are encrypted under a key known to the client.

The "tree at the last level ℓ", which has a single node, is kept by the client. That tree has just a single entry, corresponding to the interval $[0, 2^\ell)$, and containing two leaf-numbers of the tree at level $\ell - 1$ that are currently assigned to the entries of the sub-intervals $[0, 2^{\ell-1}), [2^{\ell-1}, 2^\ell)$.

ORAM Access Query. To access the logical address v, the client looks in its level-ℓ "tree" and determines the level-$(\ell-1)$ sub-interval containing v, namely j such that $(j-1)2^{\ell-1} \le v < j2^{\ell-1}$. The client sets $v_{\ell-1} = (j-1)2^{\ell-1}$ and $L^{(\ell-1)} = L_j^{(\ell-1)}$, chooses at random a new leaf $\hat{L}^{(\ell-1)}$ and replaces $L_j^{(\ell-1)}$ by this new value in the list. Then the client proceeds iteratively for $i = \ell - 1$ down to 0:

1. Request from the server all the buckets on the path from the root of the level-i tree down to the leaf $L^{(i)}$. Decrypt them and find in them an entry of the form $(L^{(i)}, v_i, \mathsf{data})$.
2. If $i > 0$ do the following:
   (a) Parse $\mathsf{data} = (L_1^{(i-1)}, L_2^{(i-1)})$, choose a new random leaf in the next tree, $\hat{L}^{(i-1)}$.
   (b) Determine the level-$(i-1)$ sub-interval containing v, namely $j = 1$ if $v < v_i + 2^{i-1}$ and $j = 2$ otherwise. If $j = 1$ then set $v_{i-1} = v_i + 2^{i-1}$ and otherwise $v_{i-1} = v_i$, and also set $L^{(i-1)} = L_j^{(i-1)}$.
   (c) Replace $L_j^{(i-1)}$ by $\hat{L}^{(i-1)}$ inside data, denoting the result by data'.

Else ($i = 0$), if this is a write operation then set data' to be the new value. Otherwise (read), set $\mathsf{data}' = \mathsf{data}$.

3. Remove the entry $(L^{(i)}, v_i, \mathsf{data})$ from the bucket where it was found, and place in the root bucket the entry $(\hat{L}^{(i)}, v_i, \mathsf{data}')$. Re-encrypt all the buckets and send to the server.

Finally, the client and server run the Eviction subroutine for each of the trees $i = 0, 1, \ldots, \ell - 1$. If this was a read operation then the return value is the data value from the last level $i = 0$.

2.1.2 Access by Keyword

Gentry et al. described in [9] how to extend this protocol to access elements by keyword rather than by index, when the database itself is sorted by that keyword: In an entry corresponding to an interval $[v, v + 2^i)$ we keep not only the two leaf values L_1, L_2 for the next tree, but also the keyword value K of the database record at the middle of this interval (i.e., at index $v + 2^{i-1}$). The access procedure is then modified so that in Step 2b above we choose the sub-interval by comparing the keyword K^* that we seek to the value K that is stored with the current entry, setting $j = 1$ if $K^* < K$ and $j = 2$ otherwise.

Note that even if the keyword K^* that we search for is not in the ORAM, we will still return some data at the end of the access protocol, Namely the data corresponding to the smallest keyword $K' \geq K^*$ in the ORAM. Jumping ahead, in our private-query protocol we handle this matter by multiplying the data with the indicator bit $\chi(K = K^*)$.

2.2 Somewhat Homomorphic Encryption (SWHE)

Our implementation of the private database search protocol relies on the `HElib` library for implementing homomorphic encryption [15, 14]. One of the features of this library that we utilize is the ability to choose freely the plaintext space. In particular, we often mix homomorphic operations modulo different moduli (e.g., 2,16,128) in the same protocol. We denote homomorphic addition and multiplication by $\boxplus$ and $\boxtimes$, respectively.

Another feature of `HElib` that we rely on is the ability to "pack" many plaintext elements in a single ciphertext and apply to them operations in a SIMD manner. We refer to the different plaintext values in a single ciphertext as the "plaintext slots" of that ciphertext. (For the specific parameters that we chose for our implementation we get 120 plaintext slots per ciphertext.) Our protocols use in particular the `HElib` procedures for computing total sums and partial sums of the plaintext slots, and the efficient implementation of permuting the slots as described in [14]. We also use the ability to homomorphically extract the bits in the binary representation of the plaintext elements when the plaintext space is a power of two, as described in [11] and [1, Appendix B].

3 Main Building Blocks

Below we describe the main low-level protocols that we use in our implementation, for things like comparing numbers, permuting arrays, etc. These protocols could be useful in many other settings as well.

In all the protocols below we use encryption schemes that support at least additive homomorphism with function privacy (in the honest-but-curious model). Below we assume for simplicity that they all operate over plaintext space $R = Z_m$ for some integer m.[3] We assume that we can instantiate the

[3]Essentially the same protocols apply also to more complex plaintext spaces, such as vectors over rings and polynomial rings.

cryptosystem relative to an arbitrary plaintext space $R = Z_m$, and we use several different instances with different plaintext spaces. As mentioned in Section 2, contemporary lattice-based cryptosystems indeed support additive homomorphism (and more) with a free choice of the plaintext space.

In terms of security, all the sub-protocols below have the property that the view of each player consists only of ciphertexts relative to keys of the other player, and ciphertext under its own keys that encrypt uniformly random plaintext elements (independent of the input and output of the protocol). Although we do not argue here the security of the sub-protocols in isolation, we use that property when proving that the high-level protocol that uses them is secure (in the honest-but-curious model).

3.1 Equal-to-Zero Protocol

The server has an input ciphertext $c = HE_C(x)$, encrypting some $x \in R$ under the client key. The goal of the protocol is for the server to obtain an encryption of a single bit b under the client key, such that $b = 0$ if $x = 0$, and $b = 1$ otherwise. Let n be the number of bits that it takes to represent an element in R, so $|R| \leq 2^n$.

The protocol consists of multiple rounds, where in each round we transform an equal-to-zero instance with plaintext space of some size S into another equal-to-zero instance with plaintext space of size $O(\log S)$. After $\log^* n$ such rounds we arrive at an instance relative to a small constant plaintext-space, and then use standard protocols (e.g., a secure computation of the AND function) to compute the final bit encryption. The plaintext-space reduction protocol consists of only a single message flow (i.e., half a round) and it is described next.

3.1.1 Plaintext-space reduction

We begin by turning the encryption of x into encryption of (roughly) the bits of x. Namely, the server proceeds as follows:

S1. Choose a random $a \in R$ and use homomorphism to compute $c' \leftarrow c \boxplus a = HE_C(x + a)$.

S2. Denote the bit representation of a by $a_{n-1} \ldots a_1 a_0$. Encrypt the bits a_i under the server's key, but *relative to plaintext space* Z_{n+1}, getting $c_i = HE_S(a_i)$ for $i = 0, \ldots, n-1$.

The server sends to the client both c' and all the c_i's. The client then proceeds as follows:

C3. Decrypt c' to obtain the value $x' = x + a \in R$, and let $x'_{n-1} \ldots x'_0$ be the bit representation of this value. Note that $x' = a$ iff $x = 0$.

C4. Use the homomorphism to XOR the bit x'_i into the ciphertext c'_i for all i, by setting $c'_i = c_i$ if $x'_i = 0$ and $c'_i = 1 \boxminus c'_i$ if $x'_i = 1$.

Let $y_i = a_i \oplus x'_i$ be the value encrypted in the ciphertext c'_i, and observe that the y_i's are all zero if and only if $x = 0$.

C5. Use homomorphism to sum up all the c'_i's, thus getting a ciphertext $c'' \leftarrow \boxplus_i c'_i = HE_S(\sum_i y_i)$.

The crux of the protocol is that since the scheme HE_S is homomorphic relative to the plaintext space Z_{n+1}, and since c'' is the sum of n bits, then it encrypts zero if and only if all the y_i's are zeros, namely if and only if $x = 0$. Thus we reduced the original ciphertext c (which was relative to the plaintext space R of size up to 2^n), to a ciphertext c'' relative to the plaintext space Z_{n+1}, so that c'' encrypts a zero if and only if the original c encrypts a zero.

3.1.2 Equal-to-zero.

Our equal-to-zero protocol repeats the above plaintext-space reduction protocol for $\log^* n$ rounds, switching the client and server roles for each round, until we arrive at a plaintext space of constant size (which can be made as small as Z_3, but no smaller).

In the last step of the protocol, however, we replace the step C5 by a secure encrypted-AND protocol. (If the cryptosystem supports multiplicative homomorphism then we can use it directly. Otherwise, we can use any standard secure-computation protocol, e.g., based on OT.) In our implementation we stop at plaintext space Z_8, and then use multiplicative homomorphism to complete the protocol.

Once we have an encryption of the target bit relative to some small plaintext space, we can convert it to an encryption relative to the original plaintext space R (or any other desirable plaintext space), e.g., by a one-round protocol of blind/encrypt/re-encrypt/unblind.

We note that the original scheme (that determines the input and output to the protocol) need not even support full additive homomorphism: it is enough for it to be *blindable*, and indeed in our implementation we sometime apply this protocol to AES in counter mode. The intermediate schemes with smaller plaintext space, however, must be (at least) additively homomorphic, and for those we use lattice-based encryption schemes.

We also note that we can use essentially the same protocol to compute an encrypted bit b which is zero if *the lowest ℓ bits of x are zero* and one otherwise (for any value of $\ell \le n$ known to the client). The only difference is that in the first invocation of the plaintext reduction sub-protocol the client only computes the c'_i's for $i = 0, \ldots \ell - 1$ in step C4 (rather than all of them). We use this variant in our sub-protocol for computing the encrypted permutation during eviction, see Section 4.2.

3.2 Comparison Protocol

This protocol builds on the equal-to-zero protocol from above. For our basic protocol, we have the client holding an n-bit number y in the clear, and also holding the bit-wise encryption of another number x under the server's key. The goal of the protocol is for the client to obtain an encryption of a single bit b under the client key, such that $b = 0$ if $x \ge y$ and $b = 1$ if $y > x$. Later in this subsection we discuss some optimizations that we use when transforming our actual setting that we have in our implementation to the one needed for this protocol.

Input. The client holds a plaintext element $y \in Z_{2^n}$ and n ciphertexts $c_i = HE_S(x_i)$ under the server key that encrypt the bits of the integer $x = \sum_{i=0}^{n-1} x_i 2^i$, relative to plaintext space Z_{n+1}.

C1. The client XORs the bits of y into the c_i's, setting $c'_i = c_i$ if $y_i = 0$ and $c'_i = 1 \boxminus c_i$ if $y_i = 1$. Denote by $b_i = y_i \oplus x_i$ the bits that are encrypted in the c'_i's.

At this point we note that if $x = y$ then all the b_i's are zero, and if $x \ne y$ then some of the b_i's are ones. Moreover, the largest index i^* for which $b_i = 1$ corresponds to the top bit where x, y differ.

C2. The client uses additive homomorphism to compute the *partial sums*, i.e. for all i its sets $c''_i \leftarrow \boxplus_{i' \ge i} c'_i = HE_S(s_i)$, where $s_i = \sum_{j=i}^{b} b_i$.

Note that if $x = y$ then all the s_i's are zero, and if the top bit in which x, y disagree has index i^* then we have $s_i = 0$ for all $i > i^*$ and $s_i \ne 0$ for all $i \le i^*$ (since each of the latter s_i's is a sum of $\le n$ bits, not all of them zero).

EQ3. The client and server apply the equal-to-zero protocol from Section 3.1 to each of the ciphertexts c''_i. At the conclusion of these protocols the client holds $\hat{c}_i$, $i = 0, \ldots, n-1$, where $\hat{c}_i = HE_S(0)$ for $i > i^*$ and $\hat{c}_i = HE_S(1)$ for $i \le i^*$.

C4. Subtracting $\hat{c}_{i+1}$ from $\hat{c}_i$ for all $i < n$ yield ciphertexts $\tilde{c}_i$, all of which encrypt the bit 0 except $\tilde{c}_{i^*} = HE_C(1)$. (If $x = y$ then all the $\tilde{c}_i$'s encrypt zeros.)

C5. The client multiplies $c_i^* = y_i \boxdot \tilde{c}_i$.

Clearly we still have $c_i^* = HE_S(0)$ for $i \neq i^*$, but for $i = i^*$ we now have $c_{i^*}^* = HE_S(1)$ if $y_{i^*} = 1$ and $c_i^* = Enc_S(0)$ if $y_{i^*} = 0$. Recalling that i^* is the top bit where x, y disagree (if any), we have that $y > x$ if and only if $y_{i^*} = 1$. Hence all the c_i^*'s are encryption of 0's if $c \geq y$, and one of them is an encryption of 1 if $y > x$.

C6. Summing up the c_i^*'s yields $c^* = HE_S(b)$ where $b = 1$ if $y > x$ and $b = 0$ if $x \geq y$, as needed.

Encrypting integers in reverse bit-order. In our implementation, we use the encrypted comparison protocol to compare the keyword held by the client to the pivots that are stored encrypted on disk as part of the path-ORAM structure. This means that at the beginning of the protocol the server has the value x (pivot) encrypted under the client key, and the client has the value y (keyword) in the clear.

If the pivot value x is encrypted bitwise in the ORAM structure then transforming it to the starting state needed for the protocol above would be a straightforward one-flow blind-decrypt-unblind protocol. However, to save on bandwidth in other parts of the protocol we would prefer to encrypt the pivot as either a single integer or a sequence of integer digits, which makes it harder to extract the bits. To handle this issue without resorting to higher-degree homomorphism we encrypt the integer x in *reverse bit order*, and modify the basic comparison protocol from above very slightly as follows:

Input. The client holds a plaintext element $y \in Z_{2^n}$, and the server holds an encryption of an n-bit integer x under the client key in reverse bit order. Namely, a ciphertext $c = Enc_C(\tilde{x})$ with $\tilde{x} = \sum_{i=0}^{n-1} 2^{n-i-1} x_i$. We also denote $\tilde{y} = \sum_{i=0}^{n-1} 2^{n-i-1} y_i$.

S(i). Server chooses a random $a \in Z_{2^n}$ and computes $c' \leftarrow c \boxplus a = HE_C(\tilde{x} + \tilde{a})$, where $\tilde{a} = \sum_{i=0}^{n-1} 2^{n-i-1} a_i$ is the reverse-bit ordering of a.

S(ii). The server encrypts the bits a_i under its own key, relative to plaintext space Z_{n+1}, getting $c_i = HE_S(a_i)$ for $i = 0, \ldots, n-1$.

The server sends to the client both c' and all the c_i's.

C(iii). The client decrypts c and subtract $\tilde{y}$ from the result, thus getting the n-bit plaintext $\tilde{z} = \tilde{x} - \tilde{y} + \tilde{a}$, and let z be the reverse-bit ordering of $\tilde{z}$.

Observe that if $x = y$ then clearly $\tilde{z} = \tilde{a}$ and therefore also $z = a$. If the top bit in which x, y differ is bit i^*, then the lowest $n - i^* - 1$ bits of $\tilde{z}$ and $\tilde{a}$ agree (due to the reverse-bit order), and therefore i^* is also the top bit where z, a disagree. We can therefore apply steps C1 through C4 of the comparison protocol from above to the plaintext z and encrypted a_i's, but multiply the bits y_i (rather than z_i) in step C5. Correctness follows by the exact same argument as above

We also note that the same protocol can be applied if we have x encrypted as a sequence of digits, so long as each individual digit is encrypted in reverse bit order. Indeed in our implementation we encrypt x as a sequence of 4-bit digits.

3.3 Blinded Permutations

As input to this protocol, the server has an encryption under the client key of a size-ℓ array a and another size-ℓ array p containing a permutation of the index set $\{1, 2, \ldots, \ell\}$ (over some plaintext space $\mathbb{Z}_m$ with $m \geq \ell$). The output of the server is an encrypted array a' which is obtained by permuting A according to p. Namely, $a'[p[i]] = a[i]$ for all i.

The protocol is described in Figure 1. We denote the key-pairs of the client and server by $(\mathsf{sk}_C, \mathsf{pk}_C)$ and $(\mathsf{sk}_S, \mathsf{pk}_S)$, respectively. Also we denote plaintext values by lowercase letters and ciphertext values by uppercase letters.

	Client$(\mathsf{sk}_C, \mathsf{pk}_S)$		**Server**$(\mathsf{sk}_S, \mathsf{pk}_C,\ A = \mathsf{Enc}_C(a),\ P = \mathsf{Enc}_C(p))$
1.			Blind the entries in A: for $i = 1..\ell$:
1a.			$r[i] \leftarrow \mathbb{Z}_m$
1b.			$B[i] := A[i] \boxplus r[i]$
2.			Encrypt $R[i] \leftarrow \mathsf{Enc}_S(r[i])$
3.			$q[1..\ell] \leftarrow$ random permutation over $\{1, \ldots, \ell\}$
4.			Permute B, P, R according to q: for $i = 1..\ell$:
4a.			$\tilde{B}[q[i]] \leftarrow B[i]$
4b.			$\tilde{P}[q[i]] \leftarrow P[i]$
4c.			$\tilde{R}[q[i]] \leftarrow R[i]$
		$\longleftarrow \tilde{B}, \tilde{P}, \tilde{R} \longleftarrow$	
5.	Decrypt $\tilde{P}$: for $i = 1..\ell$:		
5a.	$\tilde{p}[i] \leftarrow \mathsf{Dec}_{\mathsf{sk}_C}(\tilde{P}[i])$		
6.	Permute $\tilde{B}, \tilde{R}$ according to $\tilde{p}$: for $i = 1..\ell$:		
6a.	$B^*[\tilde{p}[i]] \leftarrow \tilde{B}[i]$		
6b.	$R^*[\tilde{p}[i]] \leftarrow \tilde{R}[i]$		
7.	Blind B' and R': for $i = 1..\ell$:		
7a.	$s[i] \leftarrow \mathbb{Z}_m$		
7b.	$B'[i] = B^*[i] \boxplus s[i]$		
7c.	$R'[i] = R^*[i] \boxplus s[i]$		
		$\longrightarrow B', R' \longrightarrow$	
8.			Decrypt and unblind: for $i = 1..\ell$:
8a.			$r'[i] \leftarrow \mathsf{Dec}_{\mathsf{sk}_S}(R'[i])$
8b.			$A'[i] \leftarrow B'[i] \boxminus r'[i]$

Figure 1: Blinded permutation protocol

To see that the protocol from Figure 1 returns the correct output, denote by a', b', r' the plaintext arrays encrypted in A', B', R', respectively. We need to show that $a'[p[i]] = a[i]$. This holds because (by line 4b) we have $p[i] = \tilde{p}[q[i]]$ for all i, and therefore:

$$\begin{aligned} b'\big[p[i]\big] &\overset{(C7b)}{=} b^*\big[p[i]\big] + s\big[p[i]\big] \overset{(S4b)}{=} b^*\big[\tilde{p}[q[i]]\big] + s\big[p[i]\big] \\ &\overset{(C6a)}{=} \tilde{b}\big[q[i]\big] + s\big[p[i]\big] \overset{(S4a)}{=} b\big[i\big] + s\big[p[i]\big], \end{aligned}$$

where the numbers above the equalities refer to line numbers in Figure 1. The same argument shows that $r'[p[i]] = r[i] + s[p[i]]$, hence we have

$$\begin{aligned} a'[p[i]] &\overset{(S8b)}{=} b'[p[i]] - r'[p[i]] = \big(b[i] + s[p[i]]\big) - \big(r[i] + s[p[i]]\big) \\ &= \ b[i] - r[i] \overset{(S1b)}{=} a[i]. \end{aligned}$$

In terms of security, we again note that the view of each party contains ciphertexts under the other party's key, and ciphertext under the party's own key that encrypt random elements that are independent of the protocol input and output (or a random permutation in the case of $\tilde{P}$).

4 Protocols for Private Queries

Below we describe on a high level the main protocols in our implementation. More detailed description is available in Appendix C. On a high-level, every database access proceed tree by tree, and processing each tree is done in two phases. First the server reads the root-leaf "read-path" from the tree and the client and server engage in a *Read-and-Update* protocol. Then the server reads a (potentially different) root-leaf "evict path" from the tree, and the client and server engage in an *Eviction* protocol.

We logically use additive two-out-of-two secret sharing to share the ORAM state between the client and server, but rely on an optimization that allows the client to hold just a single AES key instead of a long share. Namely, the ORAM trees themselves are stored at the server, encrypted using AES-CTR under the client's key.

4.1 ORAM Read & Update

The read-phase protocols are used to read a path from one tree in the encrypted ORAM structure, extract from it the information that we need in order to read the next tree, and update the read path. At the beginning of the read phase, the server is holding a single root-leaf path, with each entry encrypted separately using AES-CTR under the client's key. In addition the server is also holding an AES-CTR encryption of a tag t^*, identifying the entry to extract from this path, and the client is holding in the clear the keyword that it is looking for (which should be compared to the pivot in that entry).

We also maintain the invariant that prior to processing the current tree, the client and server compute two values d_c, d_s, respectively, whose XOR will be assigned as the new leaf value to the matching entry. This phase consists of four parts:

Extract. Extract a single entry from the path containing the information that we seek. More details on this step are given in Figure 2 and Appendix C.1.

Compare. Compare the pivot in the extracted entry against the keyword that we are searching for. Compute a single encrypted bit that contains the result of that comparison. This is done using the comparison protocol from Section 3.2. The low-level details are described in Appendix C.2.

Oblivious-Transfer. Extract one of the two data-items in the entry, depending on the value of the encrypted bit, getting in the clear the path to read in the next tree, and also an encryption of the identifier tag to seek in that path. This is a fairly standard 1-of-2 OT protocol, details are provided in Appendix C.3.

Update. Update the path in the current tree, marking the entry that was extracted as "empty", and copying its content to an available empty slot in the root bucket. Also update the leaf value for that entry to a new random leaf. This protocol is fairly standard on a high level, but uses some HE-specific optimizations to speed up low-level operations, see details in Appendix C.4.

When processing the largest tree (that contains the data itself), then in the OT step we also execute an equality protocol to check that the keyword matches the one that we search for, and multiply the returned data by the resulting bit, thus zero-ing it out if the keyword does not exist in the database.

Extraction protocol. Server has encryption of all entries e_i on the read path under the client AES-CTR key. Each $e_i = (t_i, p_i, d_i)$ with t_i and identifier tag, p_i a pivot, and d_i some data. The server also holds an encryption of the target tag t^* under the client AES-CTR key.

S1. Server chooses R^* to mask t^* and R_i's to mask e_i's.
- R'_i is the part of R_i masking the tag t_i.

S2. Server sends to client:
$C' = AES.CTR_C(t^* + R^*)$,
$\{C'_i = AES.CTR_C(e_i - R_i)\}_{i=1}^{\ell}$,
$\{C''_i = HE_S(R_i)\}_{i=1}^{\ell}$, and
$\{C'''_i = HE_S(R^* + R'_i)\}_{i=1}^{\ell}$

C3. Client decrypts $t' = t^* + R^*$,$\{e'_i = e_i - R_i\}_{i=1}^{\ell}$
- Let (t'_i, p'_i, d'_i) be the (tag,pivot,data) parts in e'_i, so $t'_i = t_i - R'_i$

C4. Client adds $(t'_i - t')$ to C'''_i, getting
$C^*_i = C'''_i \boxplus (t'_i - t') = HE_S(\delta_i)$
- $\delta_i = (t_i - R'_i) - (t^* + R^*) + (R^* + R'_i) = t_i - t^*$

C5. Client multiplies each C^*_i by a random nonzero value (over a field), getting $\{\tilde{C}_i = HE_S(Z_i)\}_{i=1}^{\ell}$
- $Z_{i^*} = 0$ and $Z_i \neq 0$ random for all $i \neq i^*$

C6. Client chooses masking values S_i to mask the e'_i's and a random rotation amount x.
- S'_j is the part of S_j that masks (p_j, d_j).

C7. Client sends to server:
$\{\Gamma_j = \tilde{C}_{j-x} = HE_S(Z_{j-x})\}_{j=1}^{\ell}$,
$\{\Delta_j = C''_{j-x} \boxplus S_{j-x} = HE_S(R_{j-x} + S_{j-x})\}_{j=1}^{\ell}$
$\{\Psi_j = HE_C((p'_{j-x}, d'_{j-x}) - S'_{j-x})\}_{j=1}^{\ell}$
- Recall (p'_j, d'_j) is (pivot,data) part of $e_j - R_j$, $(p'_j, d'_j) - S'_j$ is (pivot,data) part of $e_j - R_j - S_j$

S8. Server decrypts Γ_j's, finds unique index j^* such that Γ_{j^*} encrypts a zero
- Note that $j^* = i^* - x \pmod{\ell}$
- All other Γ_j's encrypt random nonzero values

S9. Server decrypts Δ_{j^*}, gets $R_{i^*} + S_{i^*}$.
- Let (p^*, d^*) be (pivot,data) parts of $R_{i^*} + S_{i^*}$

S10. Server outputs $\hat{C} = \Psi^*_j \boxplus (p^*, d^*) = HE_C(p_{i^*}, d_{i^*})$

The server outputs encryption of the (pivot,data) parts of the entry with matching tag $t_{i^*} = t^*$.

Figure 2: The extraction sub-protocol.

4.2 ORAM Eviction

Eviction consists of first computing (an encryption of) the permutation to apply to the entries along the eviction path, and then applying it using the protocol from Section 3.3. At the beginning of the eviction phase, the client and server agree on the eviction path, and the server has the content of all the buckets along that path, which are all encrypted under the client AES key. Each entry of every bucket contains a target-leaf field, we begin the protocol with one round of blind/decrypt/re-encrypt/unblind that converts these AES ciphertexts to HE ciphertexts and also packs them in the slots of a single HE ciphertext.

For a height-h tree with Z-size buckets and S-size stash, we therefore have $hZ + S$ plaintext elements packed in one HE ciphertext, each of them an h-bit string. In our implementation we use $Z = 2$, $h \leq 22$ and $S = 24$, and use 120-slot ciphertexts, so a single ciphertext can hold (more than) $2hZ + S$ target-leaf fields. We will need the extra hZ slots to hold "dummy entries" in the protocol below. The eviction phase consists of several sub-protocols, as described below.

Sub-protocol 1: position bits. Denote the target leaf of the i'th entry in the path by $l[i]$, and denote the leaf at the bottom of the eviction path by l^*. For every level $j = 1 \dots h$ in the tree (with $j = 0$ the root and $j = h$ the leaves), we first want to compute ciphertexts $C_j[i]$ under the client key that encrypt one if $l[i]$ and l^* agree on the first (lowest) j bits, and zero otherwise. This means that entry i wants to get evicted at least as far down as level j. These bits should be encrypted wrt plaintext space Z_m for $m \geq 2hZ + S$, in our implementation we use $m = 128$.

To compute the $C_j[i]$'s, we use additive homomorphism to subtract l^* from the $l[i]$'s, getting encryption of $\delta[i] = l[i] - l^*$, and then apply our equal-to-zero protocol from Section 3.1 h times to each $\delta[i]$, each time computing whether the bottom j bits of $\delta[i]$ are zero (for $j = 0 \dots h-1$). Note that if the $\delta[i]$'s are all packed in a single ciphertext then we just need to perform h executions of the protocol, one per j, and we get packed ciphertexts $C_j[0 \dots 119]$. Also we can perform most of the first plaintext-reduction step in the equal-to-zero sub-protocol only once (rather than for every j separately).

Position indexes. Once we have the encrypted bits $C_j[i]$, we can sum them up to get an encryption of the level to which this entry wants to be evicted. Denote this index by $v[i]$. Although the protocol below does not use the encryption of $v[i]$, it is nonetheless convenient to use the $v[i]$'s to explain the working of this protocol. Roughly, in this protocol we would want to sort the entries by their position index.

Sub-protocol 2: adding dummy ciphertexts. Next we add encryption of some dummy entries, to ensure that for any level below the root $j > 0$ we have at least $(h - j + 1)Z$ entries with position indexed $v[i] \geq j$. The reason is that we must ensure that once the entries are sorted by their position index, no entry is sent further down the path below the level that that it wants to get to. Hence if we have less than $(h-j+1)Z$ entries that want to get to level j or below, we need to fill these levels with dummy entries so that entries that want to go to higher levels will not get sorted into the lower ones.

We begin by computing encrypted counts E_j of how many entries want to be evicted to levels j and below, simply by summing $E_j = \boxplus_i C_j[i]$. Similarly the number of entries that want to go exactly to level j is $E'_j = E_j \boxminus E_{j-1}$. Let e_j denote the number encrypted in the ciphertext E_j, and e'_j denote the number encrypted in the ciphertext E'_j.

Next we use the E_j's to compute for each level j how many dummy entries (between 0 and Z) are needed at that level. I.e., for all $j = 1 \dots h$ and $k = 1 \dots Z$ we compute an encryption of the bit $\sigma_{j,k}$ which is one if we need to add k or more dummies to level j and zero otherwise. It can be verified that the condition we need is

$$\sigma_{j,k} = 0 \text{ iff } \exists j' \geq j \text{ s.t. } \Big(\sum_{t=j}^{j'} e'_j\Big) > (j' - j)Z + k. \tag{1}$$

That is, if there are more than $(j' - j)Z + k$ entries that want to be evicted to levels between j and j' (for some j'), then we need to add less than k dummies to level j.

Unfortunately we cannot use the comparison protocol from Section 3.2 to compute the bits $\sigma_{j,k}$ from the E'_i's, since the e'_i's are sum of bits, so they are integers which are not encoded in reverse bit order. However, the e'_i's are relatively small (at most $2hZ + S = 112$) hence even the naive protocol is reasonably efficient. Specifically for each j' we subtract $(\boxplus_{t=j}^{j'} E'_t) \boxminus ((j' - j)Z + k)$, over plaintext space Z_{128}, and then use homomorphic bit extraction to get the MSB of the result, which is the indicator bit $\chi(\sum_t e'_t \leq (j' - j)Z + k)$. Computing the AND of these indicator bits gives us the bit $\sigma_{j,k}$ that we seek. We can actually pack these comparisons and run them in a SIMD manner. We note that this sub-protocol is the most time-consuming part of the entire ORAM-access procedure. In our implementation it accounts for roughly 35% of the total running time.

Once we have the $\sigma_{j,k}$'s, we prepare encryption of Zh dummy entries, where the position index of the (j,k) entry is set as $\sigma_{j,k} \cdot j$. This means that we get exactly the right number of dummies with position index $v[i] = j$, and the rest of the dummies have position index $v[i] = 0$. More specifically, we compute the encrypted bits $C_j[i]$ for these dummies: if we put the (j,k) dummy in some index i, then for any $j' = 1 \ldots h$, the bit encrypted in $C_{j'}[i]$ is zero if $j' > j$, and it is $\sigma_{j,k}$ if $j' \leq j$.

Sub-protocol 3: sorting by position indexes. All that is left now is to sort by position indexes. Note that because we added the dummies, then an entry that wants to go to level j will not be moved to a deeper level $j' > j$ in the sorted order, because there are at least $(h-j)Z$ entries that want to go to levels below j.

We update the counts E_j and E'_j, counting the $C_j[i]$'s of the dummies too. Also we compute $C'_j[i] = C_j[i] \boxminus C_{j+1}[i]$ for all i, j, which is 1 if entry i wants to go exactly to level j. Then for every entry i we compute its position in the sorted order as

$$P[i] = \boxplus_j \left(\, C'_j[i] \boxtimes \left(\left(\boxplus_{i'<i} C'_j[i'] \right) \boxplus E_{j+1} \right) \, \right).$$

That is, if entry i wants to be at level j, then before it in the order will come all the entries that want to go to $j' > j$ (there are e_{j+1} such entries) and all the entries that want to go to level j and have index smaller than i in the current array.

Sub-protocol 4: applying the permutation. Now that we have an encryption of the permutation that we need to apply to the entries, we use our blinded permutation protocol from Section 3.3 to effect this permutation. This means that we pack all the data of the entries in a HE ciphertext, then apply the protocol from Section 3.3 to this ciphertext, and then convert these ciphertexts back to AES-encrypted ciphertext. In our implementation we need two HE ciphertexts to pack all the data from all the entries in the path so we apply the blinded-permutation protocol twice.

Note that, since we initially put the dummy entries at the end of the packed ciphertext, the last Zh entries after sorting must be dummies, so we can just ignore them when converting back to AES encryption.

5 Implementation

We implemented our protocols over the `HElib` implementation [15] of the BGV scheme [3], which is currently the only publicly available implementation of SWHE that supports most of the functionality that we need.

For our target setting, we used a database with 2^{22} records with 120-bit keywords and only a few bytes worth of data. As explained in Appendix A, we can handle large records by using a two-tier system, using a database as above just to get the index of the target record and then use standard ORAM without the secure-computation layer to get the records themselves.

In retrospect, the size of the records and keywords does not have much impact on the performance, indeed over 95% of the time is spent on sub-protocols which are not affected by the record/keyword sizes, and the ones that are affected only have complexity linear in that size. (For example, extrapolating from our timing results we could have handled keywords of size over 6000 bits with a moderate change of the implementation and without changing any of the parameters, and it would have added perhaps two minutes to the query time.)

Parameters and design choices. Since the analysis of the parameters for the bucket size in the path-ORAM constructions is not tight, for the implementation of our system we ran experiments to find the number of entries needed in the root (the parameter S from Section 2.1) and intermediate

Extract	Compare	OT	Update	Total read	Evict1	Evict2	Evict3	Evict4	Total evict
38 sec.	92 sec.	41 sec.	70 sec.	= 241 sec.	91 sec.	757 sec.	487 sec.	331 sec.	= 1663 sec.

Table 1: Running times of different sub-protocols in our implementation.

nodes (the parameter Z). We tested two eviction strategies, the one from [25] that uses the read path also as eviction path, and the one from [9] that deterministically covers all the paths in reverse-bit order. For each of these two strategies we tried several different sizes for the non-root nodes, and for each of those we run the ORAM for 2^{24} accesses and recorded the largest size that the stash at the root ever grows to.

Our experiments show that for the eviction strategy from [25] we need $Z = 4$ entries in the non-root nodes before the stash size stabilizes, whereas $Z = 2$ entries were enough for the deterministic strategy from [9]. Moreover for the latter strategy with $Z = 2$, the stash never grew beyond $S = 5$ entries, so we expect that setting $S = 24$ gives a reasonable security margin. This means that the entire root-to-leaf path in our largest tree needs to hold $hZ + S = 22 \cdot 2 + 24 = 68$ entries. However, our sub-protocol 2 from Section 4.2 for computing permutations requires that we add Z more dummy entries per non-root node, thus for that sub-protocol we need to handle $2hZ + S = 112$ entries.

At this point, our design choices were dictated by the interfaces that are available (or not) in `HElib`. `HElib` is built to provide an effective use of ciphertext-packing techniques [10], and in particular it provides the ability to view the multiple plaintext elements encrypted in a single ciphertext as an array and arbitrarily permute that array. However, it does not (yet) provide an interface to group several ciphertexts into a single larger array and permute that larger array. As a result, we were careful to choose our HE parameters so that all the arrays of plaintext elements that we need fit in a single ciphertext. In particular, this means that we need to fit at least 112 plaintext elements in each ciphertext.

The largest circuit depth that we need to handle in our protocols is $\lceil \log 112 \rceil = 7$ (in Sub-protocol 2 from Section 4.2), and the heuristic estimate provided by `HElib` indicates that for this depth we have a lower-bound of $\phi(m) \geq 6157$ on the m-th cyclotomic ring that we need to use (for security parameter $\lambda = 80$). Adding the constraint that the number of plaintext slots (which is the order of the quotient group $Z_m^*/(2)$) must be at least 112, we chose to work with $m = 6361$, for which $\phi(m) = 6360$, we have $|Z_m^*/(2)| = 120$ slots, and each slot can hold an element of the field $GF(2^{53})$. (Also the fact that m is a prime number makes the permutation implementation in `HElib` slightly faster.)

Finally, a modulo-2 ciphertext space would have let us pack at most 6360 plaintext bits per ciphertext, but to fit all the relevant information of an entire root-to-leaf path in the deepest tree into a single ciphertext, we needed to use plaintext space somewhat larger than that. Hence we chose to encrypt some of the data relative to plaintext space modulo $2^4 = 16$, which lets us pack four times more bits in each ciphertext. We also make use of a modulo-128 plaintext space for some of our sub-protocols.

Performance. With these parameters, a native homomorphic multiplication in `HElib` takes roughly 50ms, and permuting the 120-slot arrays takes just under one second. Our implementation of the entire protocol with these parameters runs in about 32 minutes per access (1904 seconds). Table 1 summarizes the breakout of this time into the different sub-protocols from Section 4. In that table, Extract, Compare, OT, and Update are the four sub-protocols of the read phase, and Evict1-4 are the four sub-protocol of the eviction phase.

As seen in Table 1, the most expensive are Sub-protocols 2 and 3 in the eviction phase, which between them take roughly 2/3 of the entire access time. In particular, computing the bits $\sigma_{j,k}$ from the e'_j's as in Equation (1) takes 669 seconds (35% of the total). It is conceivable that a sub-protocol with better complexity exists, this is left for future work.

We note that only the first three sub-protocols in the read phase are on the critical path for obtaining the information, all other sub-protocols can be executed "off line" after the information was obtained. Hence our current implementation features a latency of about three minutes per query, but throughput limitation of 32 minutes per query.

In terms of the time to process the separate trees, the read-and-update phase takes roughly 11 seconds per tree, regardless of the height of that tree (since this implementation manipulates a single packed ciphertext for any tree up to height 24). The current implementation of the eviction phase takes about $5h + 18$ seconds to process a height-h tree, so the first tree takes 25 seconds, and the last (height-22) tree takes 130 seconds. Overall, the running time of this implementation on a size-2^h database ($h \leq 24$) would be

$$\mathsf{Time}(2^h) \approx 2.5h^2 + 31.5h \text{ seconds},$$

of which only about $8h$ seconds are on the critical path. As we mentioned above, the keyword size does not make a big difference in our implementation: shorter keywords will not save us any time, and longer keywords will not cost us much (but would require some change in the implementation).

We view these numbers as encouraging; they indicate that SWHE-based protocols are not as slow as commonly believed. Moreover, this is only a first-step implementation and there is much room for improvement. Below we list a few promising avenues:

Parallelism. The `HElib` library is unfortunately not thread safe, so our implementation is entirely single-threaded. The operations that we do, on the other hand, are easily parallelizable, so an architecture that can exploit this would immediately yield roughly an $N\times$ improvement when using N cores in the computation.

Optimizing for low-degree parallelism. `HElib` is not specifically optimized for low-degree homomorphism. For our sub-protocols that rely only on additive homomorphism (such as our equal-to-zero, comparison, and blinded permutation), it is quite likely that using other lattice-based schemes (e.g., [2]) would yield somewhat better parameters and hence improved efficiency.

Different parameters for different sub-protocols. To simplify the development effort we chose to work with a single parameter setting throughout the protocol. However, this means that we use the largest setting of parameter that can fit everything we need in the protocol. A better-optimized implementation would use different parameters for different sub-protocols and different ORAM trees, resulting in much faster operations for the lower-degree protocols and the smaller trees.

Other optimizations. We expect that there will be many other optimizations that can be applied to our system, both at the algorithmic level at at the level of implementation. For example, one optimization that was used in the Keller-Scholl system [18] and is equally applicable to ours, is to limit the path-ORAM evictions to only pushing elements no more than 5 levels down the tree. They have experimental results showing that the ORAM overflow probability does not increase much by doing this. Using their optimization should have cut the eviction running time for the 2^{22}-record database by a factor of three or four.

We believe that implementing these improvements is likely to result in at least an order of magnitude improvement, and perhaps as much as three orders of magnitude, and that it would make this SWHE-based approach quite competitive with the approaches based on SPDZ or on Yao circuits. We would like to stress again that SWHE-based secure protocols are very new, whereas protocols based on Yao circuits and algebraic black box have been studied and implemented for at least the last two decades. Our current work takes a few steps toward making such protocols practical, and certainly more steps will follow.

Data format. In this implementation we spent a lot of effort on packing as much data as possible in a single HE ciphertext, so that we would only need to manipulate one ciphertext at a time. As a result, much of the implementation time was devoted to translating from one data representation to another. For example, the tag value that we use in the read-and-update phase is sometimes encrypted in bits, and other times relative to a mod-16 plaintext space, so we need to convert it back and forth for every tree that we process. Also, due to our "extreme packing," the update sub-protocol needs to replace just a few coefficients in the slot corresponding to the extrated entry, so we had to devise some low-level algebraic tricks to move data around inside a single slot (see the Client step in Appendix C.4.2).

References

[1] Jacob Alperin-Sheriff and Chris Peikert. Practical bootstrapping in quasilinear time. In Ran Canetti and Juan A. Garay, editors, *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2013.

[2] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.

[3] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, pages 309–325, 2012.

[4] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO (1)*, pages 353–373, 2013.

[5] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[6] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[7] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology - CRYPTO'03*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[8] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology - CRYPTO'12*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[9] Craig Gentry, Kenny Goldman, Shai Halevi, Charanjit Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *PETS*, 2013.

[10] Craig Gentry, Shai Halevi, and Nigel Smart. Fully homomorphic encryption with polylog overhead. In *"Advances in Cryptology - EUROCRYPT 2012"*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012. Full version at `http://eprint.iacr.org/2011/566`.

[11] Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping in fully homomorphic encryption. In *Public Key Cryptography - PKC 2012*, volume 7293 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

[12] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[13] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[14] Shai Halevi and Victor Shoup. Algorithms in HElib. Cryptology ePrint Archive, Report 2014/106, 2014. `http://eprint.iacr.org/`.

[15] Shai Halevi and Victor Shoup. HElib - An Implementation of homomorphic encryption. `https://github.com/shaih/HElib/`, 2014.

[16] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS'12*. The Internet Society, 2012.

[17] Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Outsourced symmetric private information retrieval. In *ACM Conference on Computer and Communications Security - ACM-CCS'13*, pages 875–888. ACM, 2013.

[18] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. Cryptology ePrint Archive, Report 2014/137, 2014. `http://eprint.iacr.org/`.

[19] Helger Lipmaa and Tomas Toft. Secure equality and greater-than tests with sublinear online complexity. In *ICALP'13 PartII*, volume 7966 of *Lecture Notes in Computer Science*, pages 645–656. Springer, 2013.

[20] Chang Liu, Yan Huang, Elaine Shi, Michael Hicks, and Jonathan Katz. Automating efficient ram-model secure computation. In *Proceedings of 35th IEEE Symposium on Security and Privacy*, 2014.

[21] Rafail Ostrovsky and Victor Shoup. Private information storage. In *STOC*, 1997.

[22] Vasilis Pappas, Mariana Raykova, Binh Vo, Steven M. Bellovin, and Tal Malkin. Private search in the real world. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 83–92, 2011.

[23] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: processing queries on an encrypted database. *Commun. ACM*, 55(9):103–111, 2012.

[24] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[25] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM Conference on Computer and Communications Security*, pages 299–310. ACM, 2013.

[26] Tomas Toft. Sub-linear, secure comparison with two non-colluding parties. In *Public Key Cryptography - PKC'11*, volume 6571 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2011.

[27] Ching-Hua Yu. Sign modules in secure arithmetic circuits. Cryptology ePrint Archive, Report 2011/539, 2011. `http://eprint.iacr.org/`.

A Extending the Architecture

The protocols so far let us look up a record in the database by its logical address or keyword, but realistic applications often need to handle more complicated queries. In particular we may want to be able to search by different attributes, e.g., by either lastName or birthDate or both. Also we may want to use conjunctive and range queries, for example obtaining all record corresponding to "lastName='Smith' AND birthDate from 'Jan-1-1970' to 'Dec-31-1979'."

Multiple attributes and conjunctions. A simple method for allowing access by a small number of attributes is to replicate the database with a copy per searchable attribute. For example, to enable searching by either lastName or birthDate (but not both) we would have two copies of the database, with the first copy sorted by keyword 1.lastName and the second copy by keyword 2.birthDate. Then rather than looking for lastName='Smith' we would search for keyword='1.Smith', and rather than searching for birthDate='Apr-1-1974' we would search for keyword='2.Apr-1-1974'.

Handling conjunctive queries can be done by pre-processing the database, introducing a new *compound attribute* for every conjunction. For example, above we would have a new attribute for lastName.And.BirthDate and then we can query the database for e.g., lastName.And.BirthDate='Smith.Jan-1-1970'. This can be combined with the method above for handling multiple searchable attributes, for example, we could have three copies of the database indexed by 1.lastName, 2.birthDate, and 3.lastName.And.BirthDate. These simple solutions essentially reduce the cases of multiple attributes and conjunctive queries to the case of a single keyword, but they require that we know at pre-processing time all the searchable attributes (including the compound ones), and that there are not too many of them.

In the case where the records themselves are large, we can mitigate the effect of replicating the database per searchable attribute by switching to a two-tier system. Specifically keep the records themselves in a *data-ORAM* that we do not replicate, and keep the indexes of records at an *index-ORAM* that we replicate for each searchable attribute. When accessing the database we would first search for the desired keyword in the index-ORAM, getting the index (sequence number) of the actual record, and then access the data-ORAM using that index.

We note that the index value should not be leaked to either the client or the server. However, we could assign to each record in the data ORAM a random identifier and store that identifier in the index ORAM (so revealing it will not leak any additional information), and use the identifier as a keyword to fetch the record from the data-ORAM.

An optimization. When using the two-ORAM solution, we can often use plain ORAM for the data without the secure-computation layer above it.[4] The reason is that for many ORAM protocols (including path-ORAM), the entire view of the client during the access protocol is uniquely determined by the results of all queries to date, so nothing new is leaked to the client during access to the data ORAM.

[4] If we have more than one data item per leaf in the largest tree of the data-ORAM then we would need a simple oblivious-transfer protocol for retrieving only one of them.

We note that this "no new leakage" argument does not always hold. For example, in the client-access-control application that we sketched in the introduction we have different client instances running different queries and we want to hide information between different instances. In these cases we cannot just use plain ORAM for the data, but perhaps if we only use access-by-index to this ORAM then we can devise cheaper solutions than our protocol from Section 4.

Range queries. In a range query, the client wishes to obtain all the records in some range and we allow the server to learn the number of records returned. Of course, for such queries to be well defined we need the values of the searchable attribute to have a linear order (or else we can impose such ordering over them).

Given the transformations from above, we can assume a single searchable keyword for the range queries. Also for simplicity of presentation we assume that every value of the searchable attribute occurs at most once in the database (this is really w.l.o.g. since we can transform every database to this form by adding sequence numbers).

To handle range queries we can again use a two-ORAM solution with the index-ORAM containing indexes of records and the data-ORAM containing the records themselves. Recall that for our access-by-keyword protocol we need the database to be sorted by the keyword values, hence if we learn the index of the first record in the range we can fetch all the other records by index. Recall from Section 2.1.2 that the ORAM protocol natively returns the first record with keyword value *greater than or equal* to the one we are searching for. Our basic approach for range queries is therefore to look up the beginning value of the range in the index ORAM, then look up the records one by one in the data ORAM.

One problem that must be addressed, however, is that we cannot reveal to either client or server the indexes of the relevant records. (For example, learning that the first index in the range query for birthDate='Apr-1-1974' is 107 leaks the information that 106 of the records belong to people that were born before that date.) This can be solved by using secure-computation over the data ORAM, rather than using plain ORAM.

Another problem to address is that we need to avoid over-shooting the data, i.e., we must be able to ensure that the server does not return records past the end of the range. This is done by running the comparison protocol before returning each record, outputting the result in the clear so that the server can terminate the interaction once a record is retrieved that has too large a keyword value.

Finally, we note that if the records themselves are large then we can optimize this solution further by splitting the data ORAM into two parts (so we have a total of three ORAM structures) with the records stored in a results-ORAM which is accessed by random identifiers (so it can use plain ORAM without the multi-party computation layer) and the middle layer being used to translate the sequential indexes to the random identifiers.

Query authorization. In many applications we need to be able to restrict the queries that the client makes so as to comply with some external policy. For example, in the client-access-control example from the introduction an individual client would need to get some stamp of approval on its query before it can query the server. Or we may have a fixed policy that governs all queries, e.g., the low and high ends of a data range query cannot differ by more than three years.

Incorporating such a query-authorization mechanism into our private-query mechanism is fairly straightforward (at least in principle): we would have in the system an "authorizer" entity whose role is to enforce the policy, and the client will run a secure computation protocol with the authorizer to get authorization before it can interact with the server. For simple access-control policies, we expect such authorization protocol to be significantly cheaper than the access protocol, so adding authorization should not have a large impact on performance.

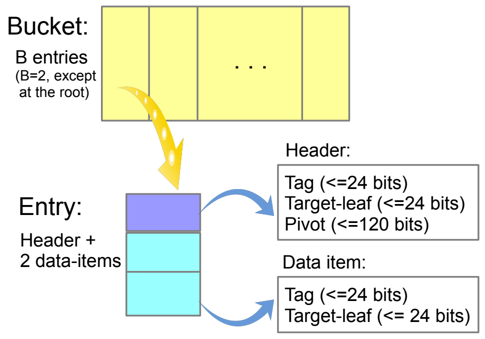

Figure 3: The bucket structure

In many cases the authorizer can be unified with the server itself, and the access control policy can be either encrypted or in the clear. If they are not unified then the authorizer can communicate its decision to the server by signing the (encrypted) query. In some cases we may want to ensure that the server (or even the client) does not learn the result of the authorization process, and that an unauthorized query looks the same as a query with no matching records. In this case the authorization protocol can output an encryption of the decision bit, and this bit can be used as a flag that controls the output of the access protocol, either returning the result from the ORAM or the fixed “no records found” result.

Dynamic databases. All the protocols that we presented in this work assume that the database is static and does not change after the initial pre-processing. Adjusting them to the dynamic-database setting requires that we modify the search-by-keyword mechanism of Gentry et al. [9] to handle insertions and deletions. This can presumably be done using balanced binary search trees, but this is a topic for future work.

B Data Format in Our Architecture

As in all previous work on secure-computation-over-ORAM, the ORAM state is shared between the client and server and the computation is done on the shared state. In our implementation we logically use additive two-out-of-two sharing, but rely on an optimization that allows the client to hold just a single AES key instead of a long share. Specifically, the ORAM trees themselves are stored at the server, encrypted using AES-CTR under the client’s key, but throughout the protocol we maintain the invariant that the client never sees the actual ciphertext data. Instead, whenever the client needs to access some ciphertext, the server first add to it a random mask, using the malleability of counter mode.

The basic logical data structure that we maintain (in a secret-shared form) is a bucket which is associated with a node in some tree, and holds the ORAM data and metadata. Every bucket other than the root of the trees contains only two entries, each entry containing two data items (that point to two leaves in the next tree). The bucket structure is depicted in Figure 3. In more detail, an entry contains the following fields:

- A 24-bit tag that identifies it,
- A 24-bit leaf value, identifying the leaf that this entry “belongs to” (used for eviction),
- A 120-bit pivot value, to be compared against the keyword that we are searching for,

- Two data items that point to the next tree, each consisting of a 24-bit identifying tag and a 24-bit target-leaf value.

The entry data fields are denoted $(t, L, \mathsf{pivot},\ (t_0, L_0),\ (t_1, L_1))$. We use the convention that empty entries have tag and target-leaf values with $LSB = 1$, and non-empty entries have tag and target-leaf values with $LSB = 0$. Hence XOR-ing 1 to the tag and the target-leaf of a non-empty entry will turn it into an empty one, and vice versa.

C Detailed Protocols

For our HE parameters we use the m-th cyclotomic field mod-p with $m = 6361$ and $p = 2$, which means that each plaintext polynomial has 120 plaintext slots, each holding an element of $GF(2^{53})$. Some of our instances will use the same parameter $m = 6361$ but with plaintext space $p = 2^4$ or $p = 2^7$, hence we get the same 120 plaintext slots but each slot holding a degree-52 polynomial modulo 16 or modulo 128, respectively. We denote these encryption methods by $HE2, HE16$, and $HE128$, respectively. In general we will be encrypting the (t, L) pair in the mod-2 plaintext slots and the other fields in the mod-16 plaintext slots.

We encrypt each entry is in two parts, one part containing the identifier tag and target-leaf value for that entry, and the other containing everything else (i.e., the two data items and the pivot). Moreover we use slightly different encryption methods for the two parts, the (t, L) pair is encrypted under AES-CTR with the encrypted counter XORed into the data, while the rest is encrypted under AES-CTR with each four-bit nibble of the encrypted counter added mod-16 to the corresponding nibble of the data. In other words, we use two-out-of-two secret sharing mod-2 of each of bit of (t, L), and two-out-of-two secret sharing mod-16 of each nibble of the rest of the entry. We denote these encryption methods by $AES2_c(\cdot)$ and $AES16_c(\cdot)$ (with the underscore $\star_c$ denoting encryption under the client's key).

C.1 Extraction Protocol

Server extraction step 1. Let us denote by $\vec{t}$ the vector of tag/leaf pairs $\vec{t} = (t_i, L_i)_i$, and denote by $\vec{p}$ the vector of all the other parts of the entries, $\vec{p} = \big((t0_i, L0_i), (t1_i, L1_i), \mathsf{pivot}_i\big)_i$. The server thus holds an AES-CTR encryption of $\vec{t}$ with XOR, an AES-CTR encryption of $\vec{p}$ with mod-16 addition, and an AES-CTR encryption of the target tag t^* with XOR.

The server begins by choosing randomness to mask all these quantities, which we denote by $\vec{r_t}$, $\vec{r_p}$, and r^*, respectively. It XORs $\vec{r_t}$ into the encrypted $\vec{t}$ and r^* into the encrypted t^* and adds mod-16 the nibbles of $\vec{r_p}$ into the encryption of $\vec{p}$.

In addition, it also encrypts under its own HE key the two vectors $\vec{r_p}$ (using mod-16 plaintext space) and $\vec{r_t} \oplus \vec{r^*}$ (using mod-2 plaintext space), where $\vec{r^*} = (r^*\ r^* \ldots r^*)$ is a a vector that has r^* in every entry. (In fact it only uses for that last ciphertext the part of $\vec{r_t}$ that is used to mask the tags, not the part that masks the leaf values, we abuse notations here and refer to this redacted vector as $\vec{r_t}$.) The vectors are encrypted by putting each entry of them in a different plaintext slot. We denote

$$\begin{aligned} z_1 &= AES16_c(\vec{p} + \vec{r_p}) & z_4 &= HE16_s(\vec{r_p}) \\ z_2 &= AES2_c(\vec{t} \oplus \vec{r_t}) & z_5 &= HE2_s(\vec{r_t} \oplus \vec{r^*}) \\ z_3 &= AES2_c(t^* \oplus r^*). \end{aligned}$$

The server sends $(z_1, z_2, z_2, z_4, z_5)$ to the client.

Client extraction step. The client decrypts the AES-encrypted values to get in the clear $\vec{u} = \vec{p} + \vec{r}_p$, $\vec{v} = \vec{t} \oplus \vec{r}_t$, and $v^* = t^* \oplus r^*$. Using additive homomorphism it XORs $\vec{v} \oplus \vec{v^*}$ into each slot of z_5, thus getting a ciphertext z_5' that encrypts in its i'th slot the quantity

$$(\vec{t}[i] \oplus \vec{r}_t[i]) \oplus (\vec{r}_t[i] \oplus r^*) \oplus (t^* \oplus r^*) \ = \ \vec{t}[i] \oplus t^*.$$

Since we have a-priory guarantee (in the honest-but-curious model) that there is exactly one entry in the path with tag t^*, then there is exactly one slot in z_5' that contains 0, and all the other slots contain non-zero elements from $GF(2^{53})$. Denote by i^* the index of the zero slot, which is the unique entry such that $\vec{t}[i^*] = t^*$.

Then the client picks random nonzero constants for all the slots and multiply them into z_5', thus getting a ciphertext z_5'' with zero in slot i^* and all the other slots containing *random* nonzero values.[5] Next the client picks a random rotation amount x and rotate the slots by x positions to the left, getting the zero slot into position $i^* - x$. We note that the plaintext encrypted in the resulting ciphertext is independent of the original ciphertext z_5 (assuming that z_5 indeed had only a single slot with $\vec{t}[i] = t^*$). The client now "blinds-by-zero" this ciphertext by adding noise without changing the encrypted value, and we denote the resulting ciphertext by z_6 (which we can view as encrypting just the value $j^* = i^* - x$).

In addition, the client homomorphically subtract $\vec{u}$ from z_1 and rotate by the same amount x, thus getting z_1' that encrypts $\vec{p} \ll x$ under the server's key. Similarly it homomorphically XORs $\vec{v}$ into z_2 and rotate by x, getting z_2' that encrypts $(\vec{t} \ll x) \oplus \vec{r^*}$.

Finally, the server chooses two random vectors $\vec{s}_p$ and $\vec{s}_t$, uses them to blind z_1', z_2', respectively, and also encrypts them under the client HE key (using mod-16 and mod-2 plaintext spaces, respectively). We denote

$$\begin{array}{lll} z_6 = \text{``}HE_s(j^*)\text{''} & & \\ z_7 = HE16_s((\vec{p} \ll x) + \vec{s}_p) & & z_8 = HE16_c(-\vec{s}_p) \\ z_9 = HE2_s((\vec{t} \ll x) \oplus \vec{r^*} \oplus \vec{s}_t) & & z_{10} = HE2_c(\vec{s}_t). \end{array}$$

The client sends all these ciphertexts to the server (some of them are only used in later sub-protocols).

Server extraction step 2. The server decrypts z_6 to recover j^* (which is the index of the only zero slot), and also decrypts z_7, z_9 to get in the clear $(\vec{p} \ll x) + \vec{s}_p$ and $(\vec{t} \ll x) \oplus \vec{r^*} \oplus \vec{s}_t$. XOR-ing $\vec{r^*}$ it can also gets in the clear $(\vec{t} \ll x) \oplus \vec{s}_t$. Adding these vectors homomorphically to z_8, z_{10}, respectively, the server gets

$$\mathsf{pp} \ = \ HE16_x(\vec{p} \ll x) \text{ and } \mathsf{tt} \ = \ HE2_c(\vec{t} \ll x).$$

Finally, the server rotates pp and tt by j^* positions to the left, getting the content of the matching entry $\vec{p}[i^*]$ and $\vec{t}[i^*]$ to slot zero, and zero out all the other slots. We denote these two ciphertexts by

$$\mathsf{p}_{i^*} \ = \ HE16_c(\vec{p}[i^*]) \text{ and } \mathsf{t}_{i^*} \ = \ HE2_c(\vec{t}[i^*]).$$

C.2 Compare-to-pivot Protocol

Recall that p_{i^*} is encrypted mod-16 under the client's key, and its slot 0 contains $\big((t0_{i^*}, L0_{i^*}), (t1_{i^*}, L1_{i^*}), \mathsf{pivot}_{i^*}\big)$. The comparison protocol compares pivot_{i^*} to the keyword kw held by the client.

Crucially for the protocol below, both the pivot and the keywords are encoded in a mod-16 *reverse bit order.* Namely the nibbles are indexed from high to low order (i.e., *little endian ordering*), and each nibble is itself encoded in reverse bit order, with the mod-16 number $a + 2b + 4c + 8d$ encoded as $d + 2c + 4b + 8a$. This means that the LSB of the first nibble is the most-significant bit of the pivot, and the MSB of the last nibble is the least significant.

[5]This method of randomizing the nonzero slots only works if the slot space is a field, which is why we must work with a mod-2 plaintext space for the $\vec{t}[i]$'s and cannot use mod-16 for everything.

Server comparison step. The server chooses a random degree-52 polynomial mod-16 to mask slot 0 of p_{i^*}. We denote that polynomial by $R3$, and also denote the coefficients that mask the different parts of slot 0 by $R3^{t0}, R3^{l0}, R3^{t1}, R3^{l1}, R3^{piv}$, respectively. Adding $R3$ to the first slot of p_{i^*} (and adding noise to "blind" this ciphertext without changing the encrypted value), it gets an encryption of $\vec{p}[i^*] + R3$ under the client key.

The server also encrypts the individual bits of $R3^{piv}$ under its own key, putting them in the slots of a ciphertext wrt a mod-128 plaintext space, the the same reverse ordering as explained above. The server also encrypts another ciphertext under its own key with $(R3^{t0}, R3^{l0})$ in slot 0 and $(R3^{t1}, R3^{l1})$ in slot 1, encrypted wrt mod-16 plaintext space. We denote:

$$z_{11} = HE16_c(\vec{p}_j + R3),\ z_{12} = HE128_s(R3^{piv}),$$
$$z_{14} = HE16_s((R3^{t0}, R3^{l0}), (R3^{t1}, R3^{l1})).$$

The server sends z_{11}, z_{12}, z_{14} to the client. (z_{14} is used in a later sub-protocol.)

Client comparison step. The client decrypts z_{11} and obtains in the clear $\mathsf{pivot}_{i^*} + R3^{piv}$. Subtracting from it the keyword kw, we have in the clear the nibbles of $\mathsf{pivot}_{i^*} - \mathsf{kw} + R3^{piv}$, in the reverse bit order as above. Denote the bit vector representing this quantity by $\vec{d} = \mathsf{pivot}_{i^*} - \mathsf{kw} + R3^{piv}$. (The client also gets in the clear the nibbles of $(t_0 + R3^{t0}, L_0 + R3^{l0}), (t_1 + R3^{t1}, L_1 + R3^{l1})$, these are not used in the comparison protocol but will be used in the oblivious-transfer protocol, as described in Appendix C.3 below.)

Note that for any nibble on which kw and pivot_{i^*} agree, the value of the corresponding four bits in $\vec{d}$ is the same as the corresponding bits of $R3^{piv}$. Moreover, due to the reverse bit ordering of the nibbles, then the carry bits of the operation $\mathsf{pivot}_{i^*} - \mathsf{kw} + R3^{piv}$ goes from more-significant to less-significant positions in $\mathsf{kw}, \mathsf{pivot}_{i^*}$. It follows that when kw and pivot_{i^*} disagree on some nibble, then the top bit in which that nibble of $\vec{d}$ differs from the corresponding nibble of $R3^{piv}$ is the same as the top bit in which $\mathsf{kw}, \mathsf{pivot}_{i^*}$ differ on this nibble. We thus conclude that the first bit where $\vec{d}, R3$ differ is also the top bit on which kw and pivot_{i^*} disagree.

The client uses the formula $a \oplus b = a + b - 2ab$ to XOR the bit vector $\vec{d}$ into the slots of the ciphertext z_{12} (that hold the bits of $R3$, encrypted wrt a mod-128 plaintext space). The result is a mod-128 ciphertext z'_{12} whose slots also hold 0/1 values, with the first slot holding a 1 corresponding to the top bit on which kw and pivot_{i^*} disagree.

Next, we use homomorphic rotations and additions to compute the running sums of the slots of z'_{12}, i.e., a ciphertext z''_{12} whose i'th slot holds the sum of all the slots $i' \le i$ in z'_{12}. Since there are only 120 slots in z'_{12}, each holding a 0/1 value, and we use a mod-128 plaintext space, it means that no wraparound can occur. Hence all the slots starting from the first non-zero slot in z'_{12} will hold a non-zero value in z''_{12}. That is, the values encrypted in the slots of z''_{12} are integers of the form $(0, 0, \ldots, 0, 1, \star, \ldots, \star)$ with all $\star$'s all nonzero, and the position of the 1 corresponding to the top bit on which $\mathsf{kw}, \mathsf{pivot}_{i^*}$ disagree.

The client and server then proceed to an equal-to-zero sub-protocol, converting the mod-128 ciphertext z''_{12} into a mod-16 ciphertext z'''_{12} (with the same number of slots), where a slot in z'''_{12} contains zero if the corresponding slot in z''_{12} does, and it contains 1 otherwise. In other words, encrypted in the slots of z'''_{12} is the vector $(0, 0, \ldots, 0, 1, 1, \ldots, 1)$ with the first 1 corresponding to the top bit on which $\mathsf{kw}, \mathsf{pivot}_{i^*}$ disagree. The equal-to-zero sub-protocol is described in Appendix C.2.1 below.

Now the client subtract adjacent slots of z'''_{12} by shifting it one position to the right (with zero fill) and subtracting, thus getting $z^*_{12} = z'''_{12} - (z'''_{12} \gg 1)$. The slots of z^*_{12} hold either a unit vector $(0, 0, \ldots, 0, 1, 0, \ldots, 0)$ with the 1 corresponding to the top bit on which kw and pivot_{i^*} disagree, or the all-zero vector if $\mathsf{kw} = \mathsf{pivot}_{i^*}$.

The client next multiplies z^*_{12} by the bit-vector of kw (still in reverse bit ordering). This of course has no effect if z^*_{12} holds the all-zero vector, and otherwise it has the effect of multiplying the unit vector by the top bit of kw that differ from pivot_{i^*}. Namely, the result (denoted by $\tilde{z}_{12}$) encrypts the all-zero vector if either $\mathsf{kw} = \mathsf{pivot}_{i^*}$ or if for the top bit where they disagree we have 0 in kw (hence 1 in pivot_{i^*}). In other words, $\tilde{z}_{12}$ holds the all-zero vector iff $\mathsf{kw} \leq \mathsf{pivot}_{i^*}$, and otherwise it holds a unit vector.

Computing the total-sum of the slots of $\tilde{z}_{12}$, we finally get a ciphertext whose slots are either all-zero (if $\mathsf{kw} \leq \mathsf{pivot}_{i^*}$) or all-one (if $\mathsf{kw} > \mathsf{pivot}_{i^*}$). We denote this ciphertext by z_{15}. This completes the comparison protocol.

C.2.1 Equal-to-zero protocol

Client step 1. Given the mod-128 ciphertext z'''_{12}, encrypted under the server key, the client choose a random vector of masks $\vec{m}$ in Z_{128} and uses it to blind the entries of z'''_{12}. The client then encrypts $\vec{m}$ under its own key, by computing 7 ciphertexts with the i'th ciphertext holding in its slots the i'th bits in all the entries in $\vec{m}$. These ciphertext are encrypted relative to plaintext space modulo-16.

The client sends to the server the blinded z'''_{12} ciphertext as well as all the mask ciphertext m_i, $i = 0, 1, \ldots, 6$.

Server step. The server decrypts the blinded z'''_{12}, recovering in the clear the vector of blinded values $\vec{v}$, with $v[j] = z[j] + m[j] \bmod 128$ for every slot j. (Here $z[j]$ denotes the content of the j'th slot in z'''_{12} before blinding, and $m[i]$ is the i'th entry in $\vec{m}$.)

For $i = 0, 1, \ldots, 6$, the server then XORs the i'th bits of all the entries of $\vec{v}$ into the ciphertext m_i (as usual with $a \oplus b = a + b - 2ab$). Denote the resulting ciphertexts by m'_i, $i = 0, 1, \ldots, 6$. Note that for each slot j in ciphertext i, the result is zero if the i'th bit in $z[j] + r[j]$ equals to the i'th bit of $r[j]$, and is one otherwise. This means that when $z[j] \neq 0$, at least one of the m'_i's will have a one in the j'th slot.

Summing up all the m'_i's, we get a ciphertext m'' over plaintext space mod-16, whose j'th slot is zero if the j'th slot of z'''_{12} was zero (before masking), and otherwise the j'th slot of m'' contains an integer between 1 and 7.

The server now uses the homomorphic bit-extraction procedure from [11, 1] to compute the product of all the bits in each slot of m''. This procedure uses degree-7 homomorphism, and the result is a ciphertext m^*, encrypted under the client key with plaintext space mod-2, such that the j'th slot containing zero if j'th slot of z'''_{12} was zero (before masking), and one otherwise.

Note that this is the result that we want, except that the server is holding it encrypted under the client key whereas we need the client to hold it under the server key. To swap we again use our usual blind and decrypt procedure, where the client blinds m^* by a random mask, encrypts the mask under its own key, and send both ciphertexts to the client.

Client step 2. The client decrypts the blinded m^* and mask XORs result to the encrypted ciphertext under the server key, thereby getting the result of the protocol.

C.3 Oblivious Transfer Protocol

Once the client has a ciphertext $z_{15} = HE16_s(b)$, encrypting the bit comparing the keyword to the pivot. (That is $b = 1$ if $\mathsf{kw} > \mathsf{pivot}$ and $b = 0$ otherwise.) We use this bit to select which of the two pairs (t_0, L_0) (t_1, L_1) should be returned to the server. In addition, in this sub-protocol we also need to prepare the output in the format needed for processing the next tree. Specifically, the server should get the value t_b encrypted under AES-CTR with plaintext space mod-2. Note that this require that we convert t_b from its current plaintext space mod-16 to a mod-2 plaintext space.

Client step 1. Recall from the comparison protocol that the client has a ciphertext

$$z_{14} = HE16_s((R3^{t0}, R3^{l0}), (R3^{t1}, R3^{l1})),$$

and that it also has in the clear the values $(t_0 - R3^{t0}, L_0 - R3^{l0}), (t_1 - R3^{t1}, L_1 - R3^{l1})$. The client chooses a random s and subtracts it from the two cleartext values $t_b + R3^{t_b}$, thus getting the cleartext values

$$(t_0 - R3^{t0} - s, L_0 - R3^{l0}), (t_1 - R3^{t1} - s, L_1 - R3^{l1}).$$

It homomorphically adds these plaintext values to the first two slots in z_{14}, thus obtaining $z'_{14} = HE16_s((t_0 - s, L_0), (t_1 - s, L_1))$. Using the encrypted bit, the client computes

$$z_{16} = z_{15} \boxtimes z'_{14} \;\boxplus\; (1 - z_{15}) \boxtimes (z'_{14} \ll 1),$$

and the first slot of z_{16} contains the pair $(t_b - s, L_b)$, with b the value of the comparison bit, and we will blind it so that only the value of this first slot will be visible to the server. This is almost what we need, except that we would need to convert the t_b part from a mod-16 representation to a mod-2 representation before we can use it for the next tree.

For this purpose, we use a "brute force" 1-out-of-16 oblivious transfer: For each nibble $s[j]$ of s we prepare 16 encrypted values under a mod-2 client key, with the i'th value being the bits of $i + s[j]$ mod 16. The client sends to the server z_{16}, and for each nibble of t_n it also sends the 16 mod-2 encrypted values. (These values can be packed in the different slots of a single mod-2 ciphertext.)

Server step 1. The server decrypts z_{16} and recovers L_b in the clear, and also all the nibbles of $t_b - s$. Denoting the j'th nibble by $v_j = t_b[j] - s[j]$ mod 16, the server uses v_j to choose one of the 16 encrypted values that it gets from the client for this nibble. Note that the v_j'th encrypted value is the bits of $v_j + s[j]$ mod 16, and by construction we have that $v_j + s[j] = t_b[j]$. Namely, the server now has the value $t_b[j]$, encrypted under the client key with a mod-2 plaintext space.

It now only remains to convert this from HE to AES ciphertext, which we do with the usual blind-and-decrypt protocol. Namely the server chooses a random r, blinds the ciphertexts and sends to the client back the encryption of $t_b[j] \oplus r[j]$.

Final OT steps. The client recovers all the $t_b[j] \oplus r[j]$'s, concatenate them together to get $t_b \oplus r$, then encrypt it under AES (mod-2) and send back to the server. The server, who knows r, can XOR it into the AES-CTR ciphertext to get AES-CTR encryption of t_b.

At this point, the server already holds L_b in the clear and an AES-CTR encryption of t_b, so we are ready to process the next tree. However, we still need to update the read path in the current tree, as described next.

C.4 Update Protocol

Next we need to update the path in the current tree, marking the entry that was extracted as "empty", and copying its content to an available empty slot in the root (while choosing a new random leaf value for that entry). In more detail, the leaf value L which is encrypted mod-2 under AES-CTR, should be replaced by $d_c \oplus d_s$ (recall that d_c, d_s are the 24-bit input values of the client and server, respectively). Moreover, the value L_b that was returned to the server above should be replaced by some new random value L'_b, and the client and server should get output values d'_c, d'_s that satisfy $d'_c \oplus d'_s = L'_b$. (These values will be used as inputs for the read & update phase of the next tree.)

C.4.1 Part I, marking the extracted entry as empty

To mark the extracted entry as empty, all we need to do is modify the pair (t, L) for this entry (which is encrypted under AES-CTR mod-2) and XOR 1 into the LSB of t.

Recall that in the extraction step 2, the server computed the ciphertext $\mathsf{tt} = HE2_c(\vec{t} \lll x)$, as well as an index j^* of the relevant entry in this shifted path. The server XORs 1 into the LSB of slot j^* in tt, then blinds the result by a random $\vec{R6}$ and returns to the client, together with a mod-2 encryption of $\vec{R6}$ under the server key (which we denote by $z_{24} = HE2_s(\vec{R6})$. The client decrypts the HE ciphertext, XOR it into z_{24} to get back an encryption of $\vec{t} \lll x$ under the server key, then rotates it back by x positions and blind it by a random $\vec{s}$ to get $z_{32} = HE2_s(\vec{t} \oplus \vec{S5})$.

The client encrypt $\vec{S5}$ under its own AES key to get $z_{31} = AES2_c(\vec{S5})$ and sends z_{31}, z_{32} to the server. The server decrypts z_{32} and XORs the result into z_{31}, thus getting the ciphertext $z_{37} = AES2_c(\vec{t})$ with the extracted entry marked as empty.

C.4.2 Part II, moving the extracted entry to the root

Next we need to move the content of the extracted entry to the root, and store it there with new leaf values L' and L'_b.

Server step 1. Recall that at the end of the extraction step 2, the server has two ciphertexts that encrypt the content of the extracted entry, $\mathsf{p}_{i^*} = HE16_c(\vec{p}[i^*])$ and $\mathsf{t}_{i^*} = HE2_c(\vec{t}[i^*])$. Below we denote $\vec{p}[i^*] = (t_0, L_0,\ t_1, L_1,\ \mathsf{pivot})$ and $\vec{t}[i^*] = (T, L)$.

First, the server chooses two random 24-bit values (u, v), records u for future use, and use the pair (u, v) it to blind t_{i^*}, getting $z_{25} = HE2_c(T \oplus u, L \oplus v)$.

Next, the server chooses fresh randomness $R8 = (u_0, v_0,\ u_1, v_1,\ r)$ and subtract it from the first slot of p_{i^*}, thus getting $z_{26} = HE16_c(t_0 - u_0, L_0 - v_0,\ t_1 - u_1, L_1 - v_1,\ \mathsf{pivot} - r)$. It also prepares two ciphertexts under it sown key, $z_{27} = HE16_s(u_0, 0, u_1, v_1, r)$ and $z_{28} = HE16_s(u_0, v_0, u_1, 0, r)$.

In addition, the server chooses a random 24-bit quantity d'_s, break it into six 4-bit nibbles and prepares four ciphertexts, $D_0, \ldots, D_3$, with D_0 encrypting the LSBs of the six nibbles (in six slots), D_1 the next bit, D_2 the third bit, and D_3 the MSBs of all the nibbles, all under a mod-16 plaintext space. The server sends $z_{25}, z_{26}, z_{27}, z_{28}$ and the D_i's to the client.

Client step. (We remark that this step requires some intra-slot data movement, since we need to update either the leaf value L_0 or L_1, which are both in the same slot.)

The client decrypts z_{25} getting $(t \oplus u, L \oplus v)$. It discards $L \oplus v$, replacing it with the 24-bit input value d_c (which was chosen at random while processing the previous tree), and setting $z_{33} = AES2_c(t \oplus u, d_c)$. The client then chooses a random 24-bit quantity d'_c and XOR it into the bits encrypted in the D_i's (using $a \oplus b = a + b - 2ab$), thus getting an encryption under the server key of the bits of $L'_b = d'_c \oplus d'_s$, relative to a mod-16 plaintext space. The client computes $D = D_0 + 2D_1 + 4D_2 + 8D_3$, and now D contains the six nibbles of L'_b in six of its slots.

Using another round-trip of blind-and-decrypt, the client uses the server to pack all these nibbles in a single slot, thus getting a ciphertext that encrypts L'_b in its first slot, $el = HE16_s(L'_b)$. Note that L'_b is a 6-nibble quantity that is contained in a slot that can hold upto 53 nibbles. We ensure that all the other nibbles that are encrypted in this slot are zero, which means that this slot (which is large enough to hold a degree-52 polynomial) contains a degree-5 polynomial.

The client prepares two constants, c_6 and c_{18}, that hold in their slots 0 the polynomials X^6 and X^{18}, respectively. Computing the two ciphertexts $el_6 = c_6 \boxplus el$ and $el_{18} = c_{18} \boxplus el$, we have

$$el_0 = HE16_s(0, L'_b,\ 0, 0,\ 0) \text{ and } el_1 = HE16_s(0, 0,\ 0, L'_b,\ 0).$$

The client then decrypts z_{26} and gets $(t_0 - u_0, L_0 - v_0,\ t_1 - u_1, L_1 - v_1,\ \textsf{pivot} - r)$ in the clear. It then prepares two constants, a_0 which has in its first slot $(t_0 - u_0, 0,\ t_1 - u_1, L_1 - v_1,\ \textsf{pivot} - r)$ and a_1 which has in its first slot $(t_0 - u_0, L_0 - v_0,\ t_1 - u_1, 0,\ \textsf{pivot} - r)$. It then sets

$$w_0 \leftarrow el_0 \boxplus a_0 \boxplus z_{27} = HE16_s(t_0, L'_b,\ t_1, L_1,\ \textsf{pivot}) \text{ and}$$
$$w_1 \leftarrow el_1 \boxplus a_1 \boxplus z_{28} = HE16_s(t_0, L_0,\ t_1, L'_b,\ \textsf{pivot}).$$

Recalling that we have a ciphertext that encrypts the result of the comparison, $z_{15} = HE16_s(b)$, the client sets

$$w_b = z_{15} \boxtimes w_1 \ \boxplus\ (1 - z_{15}) \boxtimes w_0 = \begin{cases} HE16_s(t_0, L'_b,\ t_1, L_1,\ \textsf{pivot}) & \text{if } b = 0 \\ HE16_s(t_0, L_0,\ t_1, L'_b,\ \textsf{pivot}) & \text{if } b = 1 \end{cases}$$

That is, w_b contains in its first slot the updated entry that should be encrypted under the mod-16 AES key of the client. The client chooses a random s and sets $z_{34} = AES16_c(s)$. It also subtract s from the first slot of w_b, and call the blinded result z_{35}. The client sends z_{33}, z_{34}, z_{35} to the server.

Server Step 2. The server recalls the 24-bit value u that it chose earlier and its 24-bit input value d_s, and it XORs (u, d_s) into z_{33} to get $z_{38} = AES2_c(t, L')$ with $L' = d_c \oplus d_s$.

Next the server decrypts z_{35} and adds the resulting ciphertext in slot 0 (mod 16) into z_{34} to get $z_{39} = AES16_c(t_0, L'_0,\ t_1, L'_1, \textsf{pivot})$ (where one of L'_0, L'_1 is updated with the value $L'_b = d'_c \oplus d'_s$ and the other is left unchanged from its original value).

The server finally replaces the last entries in z_1, z_{37} by z_{39}, z_{38}, respectively, thereby inserting the content of (the mod-16 portion of) the updated extracted entry in the last entry of the root. This concludes the read/update phase.