Private Database Access with HE-over-ORAM Architecture

Craig Gentry[1], Shai Halevi[1], Charanjit Jutla[1](✉), and Mariana Raykova[2]

[1] IBM T.J. Watson Research Center, Yorktown Heights, USA
`csjutla.@us.ibm.com`
[2] SRI International, Menlo Park, USA

Abstract. Enabling private database queries is an important and challenging research problem with many real-world applications. The goal is such that the client obtains 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. The SWHE schemes we consider allow easy batching or "SIMD" operations, and also let us vary the plaintext space in use. These capabilities let us devise many sub-protocols that are interesting in their own right, for tasks such as encrypted comparisons, blinded permutations, and the really expensive ORAM eviction step.

We implemented our protocols on top of the `HElib` homomorphic encryption library. Our basic single-threaded implementation takes about 30 min 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

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.

This research was done as part of IARPA SPAR Program Contract No. D11PC20202.

Since it is often possible to deduce important information from the access pattern alone (see e.g., [17] for some examples), it is important to even hide 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. [4]. 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 the client is not the data-owner and 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 that 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 an encrypted server with need-based access for its clients[2].

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 [22] 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 [13, 22], 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. [25]. Gentry et al. later proposed a few optimizations for the underlying ORAM scheme [8], 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.

[1] This report only covers the implementation of the private query protocols themselves, we briefly comment on the related authorization issue in Appendix B of the full version [9].

[2] Such was the requirement of a recent IARPA SPAR program [16].

Very recently, Liu et al. [21] 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 [19] extended the secure-computation-over-ORAM architecture to handle any number $n \geq 2$ of parties. They use the SPDZ framework [7] (with protocols based on algebraic-black-box approach with preprocessing) and use both tree-ORAM and path-ORAM as the underlying ORAM schemes. (Path-ORAM was recently proposed by Stefanov et al. [26] and is a variant of tree-ORAM with better asymptotic efficiency. As we will see later, our work utilizes Path-ORAM. The work of Keller and Scholl is concurrent to ours.)

Along a different direction, many recent works have 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. [24], and recent works on searchable symmetric encryption due to Pappas et al. [23] Cash et al. [3], and Jarecki et al. [18].

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. [8], of using secure-computation 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. [26], 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 min. 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 Sect. 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 [19]. 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 [19] 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 [19] 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 [19], since they only report the online numbers and not the "expensive" offline computations that are done by the SPDZ framework.

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. In particular, the SWHE schemes we consider allow ciphertext packing and agility in the choice of the underlying plaintext space, which leads to surprisingly efficient sub-protocols for important tasks.

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., [5,20,27,28]. 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 Sect. 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 Sect. 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 Sect. 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 Sect. 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* (this is not to be confused with big-endian format, as we actually do integer operations on this reversed integer).

Blinded Permutation. This protocol, described in Sect. 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.)

Homomorphic Path-ORAM Eviction. While the homomorphic ORAM eviction protocol which is central to secure-computation-over-ORAM may not be considered interesting in its own right, the linear time protocol we design using the batching feature of SWHE (as opposed to a quadratic-time naive approach) displays the rich capabilities of SWHE based approach to secure computation. This particular protocol may be considered the highlight of this work, and we describe it in detail in Sect. 4.2.

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 its 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 [6]. 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.

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.[3]

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 2.93 GHz. 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 min, 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 Appendix E of the full version [9].

2 Background

2.1 The Path-ORAM Protocol

In the basic path-ORAM protocol [26], 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

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

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 [26] 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).

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 [25] 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 [8]. We ran 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.

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 $v \in \{0, \dots, N-1\}$, 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

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

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 $\mathsf{Eviction}$ 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 $i = 0$.

Access by Keyword. Gentry et al. described in [8] 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 [14,15]. 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.[4] We assume that we can instantiate the cryptosystem relative to an arbitrary plaintext space $R = Z_m$, and we use several different instances with different plaintext spaces. As mentioned in Sect. 2, contemporary

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

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.

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 $x'_{n-1} \cdots x'_0$ 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 a new 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.

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 $\ell \leq n$ 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 Sect. 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 $x \geq y$ 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 $x \neq y$ 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 $c''_i \leftarrow \boxplus_{i' \geq i} c'_i = HE_S(s_i)$, 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 $s_i \neq 0$ for all $i \leq i^*$ (since each of the latter s_i's is a sum of $\leq n$ bits, not all of them zero).

EQ.3. The client and server apply the equal-to-zero protocol from Sect. 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 $i \leq i^*$.

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 note that if we encrypt the integer x in *reverse bit order* then a much simpler comparison protocol can be obtained. Due to space limitations, this protocol is described only in the full version (see [9]).

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 basic idea of the protocol is the following: The server blinds the encrypted permutation p by permuting it with a new random permutation q; the net effect of this is that when the client receives this (packed) ciphertext and decrypts it, then it comprises of the permutation $p \circ q^{-1}$. The server also blinds the array a with a random vector r. It further encrypts r under its own HE key to obtain R. Next, it permutes both the blinded a and R using q, and sends these two ciphertexts along with the blinded permutation. As mentioned earlier, the client obtains $p \circ q^{-1}$, and applies this permutation to the other two ciphertexts, (homomorphically) blinds them both using a same fresh random array, and sends them back to the client. The client now only needs to decrypt the permuted blinding array r, and subtract it (homomorphically) from the permuted (and still encrypted) a.

The protocol is described in Fig. 1 in the full version [9].

4 Protocols for Private Queries

Below we describe at a high level the main protocols in our implementation. More detailed description is available in the full version [9]. At a high-level, every database access proceeds 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 and 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).

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 Appendix D.1 of the full version [9].

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 Sect. 3.2. The low-level details are described in Appendix D.2 of the full version [9].

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 D.3 of the full version [9].

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 D.4 of the full version [9].

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.

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 Sect. 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 w.r.t. 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 Sect. 3.1 h times to each $\delta[i]$, each time computing whether the bottom j bits of $\delta[i]$ are zero (for $j = 0 \ldots h-1$). 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 $C_j[0 \ldots 119]$. 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 $E_j = \boxplus_i C_j[i]$ (each E_j can be computed in $\log(2hZ + S)$ steps by appropriate shifts and additions). 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 $j = 1 \ldots h$ and $k = 1 \ldots Z$ 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

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

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 Sect. 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, so that the protocol runs in linear time instead of quadratic time. Specifically, for each value of j' we can compute a ciphertext such that in the j-th slot is the encryption of the value $\sum_{t=j}^{j'} e'_t$. 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. However, a naive protocol which pushes entries as far down as possible (limited by $C_j[i]$), iteratively and starting from level h upto 0, would have quadratic complexity. Thus, our sub-protocol is already a major improvement.

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 Sect. 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 Sect. 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 [2], 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 B of the full version [9], 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 Sect. 2.1) and intermediate nodes (the parameter Z). We tested two eviction strategies, the one from [26] that uses the read path also as eviction path, and the one from [8] 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 [26] 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 [8]. 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 Sect. 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.

The largest circuit depth that we need to handle in our protocols is $\lceil \log 112 \rceil = 7$ (in Sub-protocol 2 from Sect. 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 $GF(2^{53})$.

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 min per access (1904 s). Table 1 summarizes the breakout of this time into the different sub-protocols from Sect. 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.

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

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

As seen in Table 1, the most expensive are Sub-protocols 2 and 3 in the eviction phase. In particular, computing the bits $\sigma_{j,k}$ from the e'_j's as in Eq. (1) takes 669 s (35 % of the total).

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 min per query.

In terms of the time to process the separate trees, the read-and-update phase takes roughly 11 s 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 s, and the last (height-22) tree takes 130 s. 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. In the full version [9] we list a few promising avenues.

References

1. Alperin-Sheriff, J., Peikert, C.: Practical bootstrapping in quasilinear time. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 1–20. Springer, Heidelberg (2013)
2. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: ITCS, pp. 309–325 (2012)
3. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013)
4. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. J. ACM **45**(6), 965–981 (1998)
5. Damgård, I.B., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 285–304. Springer, Heidelberg (2006)
6. Damgård, I.B., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 247–264. Springer, Heidelberg (2003)
7. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012)
8. Gentry, C., Goldman, K.A., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing ORAM and using it efficiently for secure computation. In: De Cristofaro, E., Wright, M. (eds.) PETS 2013. LNCS, vol. 7981, pp. 1–18. Springer, Heidelberg (2013)
9. Gentry, C., Halevi, S., Jutla, C., Raykova, M.: Private database access with he-over-oram architecture. Cryptology ePrint Archive, Report 2014/345 (2014). http://eprint.iacr.org/2014/345
10. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 465–482. Springer, Heidelberg (2012)

11. Gentry, C., Halevi, S., Smart, N.P.: Better bootstrapping in fully homomorphic encryption. In: Fischlin, M., Buchmann, J., Manulis, M. (eds.) PKC 2012. LNCS, vol. 7293, pp. 1–16. Springer, Heidelberg (2012)
12. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996)
13. Dov Gordon, S., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: CCS (2012)
14. Halevi, S., Shoup, V.: Algorithms in HElib. Cryptology ePrint Archive, Report 2014/106 (2014). http://eprint.iacr.org/
15. Halevi, S., Shoup, V.: HElib - An Implementation of homomorphic encryption (2014). https://github.com/shaih/HElib/
16. IARPA. Security and privacy assurance research (spar) program (2011). http://www.iarpa.gov/index.php/research-programs/spar/baa
17. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012. The Internet Society (2012)
18. Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.-C., Steiner, M.: Outsourced symmetric private information retrieval. In: ACM Conference on Computer and Communications Security - ACM-CCS 2013, pp. 875–888. ACM (2013)
19. Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. Cryptology ePrint Archive, Report 2014/137 (2014). http://eprint.iacr.org/
20. Lipmaa, H., Toft, T.: Secure equality and greater-than tests with sublinear online complexity. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013. LNCS, vol. 7966, pp. 645–656. Springer, Heidelberg (2013)
21. Liu, C., Huang, Y., Shi, E., Hicks, M., Katz, J.: Automating efficient ram-model secure computation. In: Proceedings of 35th IEEE Symposium on Security and Privacy (2014)
22. Ostrovsky, R., Shoup, V.: Private information storage. In: STOC (1997)
23. Pappas, V., Raykova, M., Vo, B., Bellovin, S.M., Malkin, T.: Private search in the real world. In: Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC 2011, pp. 83–92 (2011)
24. Popa, R.A., Redfield, C.M.S., Zeldovich, N., Balakrishnan, H.: CryptDB: processing queries on an encrypted database. Commun. ACM **55**(9), 103–111 (2012)
25. Shi, E., Hubert Chan, T.-H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 197–214. Springer, Heidelberg (2011)
26. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C.W., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: Sadeghi, A.-R., Gligor, V.D., Yung, M. (eds.) ACM Conference on Computer and Communications Security, pp. 299–310. ACM (2013)
27. Toft, T.: Sub-linear, secure comparison with two non-colluding parties. In: Catalano, D., Fazio, N., Gennaro, R., Nicolosi, A. (eds.) PKC 2011. LNCS, vol. 6571, pp. 174–191. Springer, Heidelberg (2011)
28. Yu, C.-H.: Sign modules in secure arithmetic circuits. Cryptology ePrint Archive, Report 2011/539 (2011). http://eprint.iacr.org/