INSPIRE: IN-Storage Private Information REtrieval via Protocol and Architecture Co-design

Jilan Lin*
jilan@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Ling Liang*
lingliang@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Zheng Qu
zhengqu@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Ishtiyaque Ahmad
ishtiyaque@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Liu Liu
liu_liu@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Fengbin Tu
fengbintu@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Trinabh Gupta
trinabh@ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Yufei Ding
yufeiding@cs.ucsb.edu
UC Santa Barbara
Santa Barbara, California, USA

Yuan Xie
yuanxie@gmail.com
UC Santa Barbara
Santa Barbara, California, USA

ABSTRACT

Private Information Retrieval (PIR) plays a vital role in secure, database-centric applications. However, existing PIR protocols explore a massive working space containing hundreds of GiBs of query and database data. As a consequence, PIR performance is severely bounded by storage communication, making it far from practical for real-world deployment.

In this work, we describe INSPIRE, an accelerator for IN-Storage Private Information REtrieval. INSPIRE follows a protocol and architecture co-design approach. We first design the INSPIRE protocol with a multi-stage filtering mechanism, which achieves a constant PIR query size. For a 1-billion-entry database of size 288GiB, INSPIRE's protocol reduces the query size from 27GiB to 3.6MiB. Further, we propose the INSPIRE hardware, a heterogeneous in-storage architecture, which integrates our protocol across the SSD hierarchy. Together with the INSPIRE protocol, the INSPIRE hardware reduces the query time from 28.4min to 36s, relative to the the state-of-the-art FastPIR scheme.

CCS CONCEPTS

• **Hardware → Memory and dense storage**; • **Security and privacy → Data anonymization and sanitization**.

KEYWORDS

In-storage computing, private information retrieval (PIR)

*Both authors contributed equally to this research.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06.
https://doi.org/10.1145/3470496.3527433

ACM Reference Format:
Jilan Lin, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. 2022. INSPIRE: IN-Storage Private Information REtrieval via Protocol and Architecture Co-design. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3470496.3527433

1 INTRODUCTION

With more data being moved to the cloud, database systems have grown quickly to become the backbone of many daily applications [35, 35, 55]. As a result, the demand for user privacy has turned into an increasingly concerning issue: *when accessing the database, can we prevent the server from knowing where parts of the database the user is accessing?* In 1995, Chor et al. introduced Private Information Retrieval (PIR) to address this problem [13]. Subsequent research has extensively studied the broad applications of PIR protocols, including anonymous communication [1, 5, 32, 44], content sharing [23, 39, 60], and business services [21, 24].

The key insight behind PIR protocols is the *all-for-one* concept. This means that to retrieve **one** record from a database obliviously, the server should necessarily must computation over **all the records** in the database. This is necessary because otherwise the server would learn which record the user is not interested in [8].

Recent breakthrough in fully homomorphic encryption (FHE) [20] has greatly expedited the development of PIR, particularly, single-server PIR that employs one server. (There is another variant of PIR that requires multiple non-colluding servers.) Fig. 1 illustrates the workflow of the state-of-the-art FastPIR protocol that is based on FHE. FastPIR uses a one-hot query vector to perform the all-for-one computation. Specifically, it encrypts the one-hot vector into a ciphertext using FHE. The server then performs *homomorphic multiplications* (namely vector convolutions) between the query ciphertext and the entire database. In addition, FastPIR uses a reduction step to shrink the answer size, using a series of *homomorphic rotations* and *aggregations*. Finally, the user derives the result using the FHE decryption function.

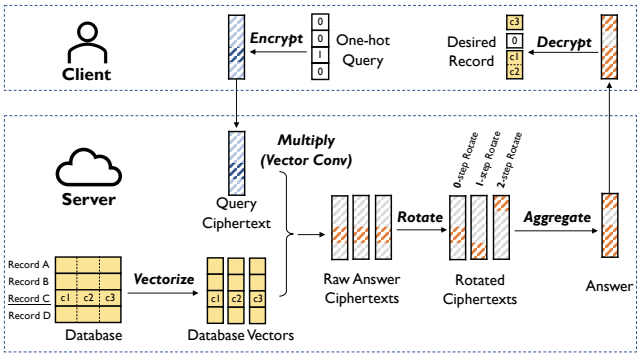

Figure 1: Workflow illustration of the state-of-the-art PIR protocol (called FastPIR [1]). To fetch the record c from a 4-entry database (containing records A, B, C, and D), the client encodes a one-hot vector into a ciphertext. The server performs a reduction computation over the entire database using homomorphic operations. Thus, record c is retrieved obliviously, and the server does not know which record is retrieved.

Despite the rigorous privacy guarantee of PIR, the all-for-one computation brings significant overhead, especially in terms of communication. The communication overhead lies in two aspects: first, the performance is bounded at the storage I/O when PIR conducts homomorphic multiplications over the entire database. Since a database usually consumes hundreds of GiBs of storage, performing computation on data from storage devices (such as SSD) is slow. In particular, we find that FastPIR takes 28.4min to fetch a 288-byte record from a 288GiB database, where 50% of the time is spent waiting for SSD accesses (Section 5.2). Second, the PIR query ciphertext is too large for real-world deployment, as the query size grows linearly with the size of the database. For the 288GiB database above, the query generated by FastPIR will be 27GiB. Although researchers have optimized PIR protocols significantly over the years, the issue of an exceedingly large amount of communication has not been well addressed.

To break the bandwidth wall in the traditional I/O and meet the intensive storage access demands, in-storage processing (ISP) appears to be a promising solution. The ISP technique directly puts the computation logic near or inside the storage device, such that the application benefits from shorter access latency and higher internal bandwidth. Prior work has broadly engaged ISP architecture with various applications, including deep learning [31], recommendations [62], and graph analytics [34]. However, applying in-storage processing to PIR is non-trivial, as naively attaching an FHE accelerator to a storage device results in sub-optimal performance.

First, the ISP technique cannot address the query size issue. In particular, when processing a batch of queries, the query data can be even larger than the database size. Even though the ISP architecture can provide $4-8\times$ higher bandwidth than external I/O, it hardly meets the growing throughput demand of large query data.

Second, the heterogeneous computation pattern in PIR requires dedicated hardware and dataflow design. Directly attaching a monolithic FHE accelerator to the SSD device cannot fully leverage the internal parallelism from multiple flash devices, because the accelerator can only gain limited bandwidth from the attached DRAM buffer, which is not enough to satisfy the memory-bound problem.

This paper describes INSPIRE which leverages protocol and architecture co-design to accelerate IN-Storage Private Information REtrieval. At the protocol level, INSPIRE's key insight is to use the classical idea of recursion [53], but while making a better trade-off between query size and computation overhead. The key trick is to partition the database hierarchically and process smaller queries in each hierarchy. Further, INSPIRE amortizes the computation overhead with a multi-stage query process. At the lower stage, it uses a block query to process the data from the entire database, which avoids the heavy computation needed for rotations. At the higher stage, it performs FastPIR-like column reductions, but while optimizing the rotation flow to reduce the large memory consumption in FastPIR. As a result, the INSPIRE protocol reduces the 27GiB query size in FastPIR to 3.6MiB for the 288GiB database.

On the hardware side, the key insight of INSPIRE's architecture is to integrate the hierarchical query processing with the micro-architecture hierarchy inside the SSD device, which can fully utilize the internal bandwidth parallelism from multiple storage channels. Specifically, we design INSPIRE as a heterogeneous architecture. As a first step, we design a block collector to perform block-level reduction and equip each flash channel with a block collector. Therefore, the block collectors leverage the channel-level parallelism for higher internal bandwidth. Next, we extend the original embedded controller in SSD to support homomorphic computation, such that the results from different channels are sent to this module to perform the more complicated answer aggregation. Through a customized dataflow that processes all data in a streaming manner, the INSPIRE architecture achieves 22.9× speedup compared with the vanilla CPU baseline.

We summarize the key contributions of this paper as follows:

- The INSPIRE protocol, which adopts a hierarchical database partitioning and multi-stage answer reduction. The protocol significantly reduces the query size and avoids the heavy rotation overhead.
- The INSPIRE accelerator based on the ISP architecture. INSPIRE explores a heterogeneous architecture, leveraging both large internal bandwidth and customized accelerator to accelerate PIR's record retrievals.
- An implementation of INSPIRE's protocol on top of SEAL [52], an open-source FHE library, that demonstrates 2.22× performance speedup relative to FastPIR. We also build a cycle-accurate simulator for the INSPIRE architecture based on MQSim [56], a simulator for SSDs, which demonstrates a 22.9× speedup over CPU and 1.93× speedup over the state-of-the-art FHE accelerator.

2 PRELIMINARIES

In this section we introduce preliminaries for further discussion, including fully homomorphic encryption (FHE), private information retrieval (PIR), the state-of-the-art FastPIR protocol [1], and in-storage processing (ISP).

2.1 Fully Homomorphic Encryption (FHE)

Modern PIR protocols rely on Fully Homomorphic Encryption (FHE) to conceal the query information. FHE is a type of encryption scheme that allows generic operations on encrypted data (ciphertext). In the most popular FHE schemes, such as BFV [9, 18], BGV [10] and CKKS [12], the raw data that is encrypted is a vector, and the ciphertext is a polynomial (represented as a vector of polynomial coefficients). Therefore, FHE programs follow a vector programming model, as most FHE operations involve element-wise computation between vectors.

Algorithm 1: FHE Primitives – Client

```
Function VecEncrypt(V, pk):
    /* Encrypt a vector V = [v1, v2, v3] into ciphertext C with
       public key pk.                                          */
return C

Function VecDecrypt(C, sk):
    /* Decrypt a ciphertext C into a plain vector
       V = [v1, v2, v3] using the secret key sk.               */
return V
```

Algorithm 1 and Algorithm 2 present the FHE primitives needed for PIR, for a small example vector containing three elements. At the client side, the function `VecEncrypt` encrypts a raw vector $V = [v_1, v_2, v_3]$ into a **ciphertext** C, where C is a polynomial of degree 3 (the same length as V). During encryption, the vector V is usually termed a **message**. It is first encoded into a polynomial, called a **plaintext**, and then encrypted to form the ciphertext. Conversely, the function `VecDecrypt` decrypts a ciphertext to a plain message. The decryption requires a secret key sk that is only known to the client.

The server-side uses three types of FHE operations: `Hom_Mul`, `Hom_Add`, and `Hom_Rot`. The `Hom_Add` and `Hom_Mul` take two ciphertexts as input and return the encryption of element-wise addition/multiplication. Note that these two functions can also take a plaintext W as inputs. `Hom_Rot` is a special operation that rotates the elements in the plain vector according to $step$. The sign of $step$ denotes the direction of rotation. For example, with $V = [v_1, v_2, v_3]$, rotating V one step to the left ($step = -1$) will result in an encryption of $[v_2, v_3, v_1]$. For different values of steps, FHE requires different rotation keys rk; these keys are generated by the client.

FHE Computation Complexity: Adding two polynomials (ciphertexts) is simple, which causes $O(M)$ time complexity with the polynomial degree of M. `Hom_Mul` needs more complicated computation, as multiplying two polynomials requires convolution. Number Theory Transfer (NTT) is widely used to accelerate this computation [51]. NTT is a variant of Discrete Fourier Transfer (DFT), which can transfer the convolution (in the time domain) to the element-wise multiplication (in the frequency domain). The computation complexity of `Hom_Mul` is then $O(NlogN)$ for NTT and inverse NTT. When multiplying two ciphertexts, the resulting ciphertext usually needs to be relinearlized with a sophisticated key switching process [51], during which NTT is also the dominant operation. Finally, `Hom_Rot` requires key switching and data reordering/shuffling. We will discuss this in more detail in Section 4.4.

Algorithm 2: FHE Primitives – Server

```
/* C1=VecEncrypt([v1, v2, v3]), C2=VecEncrypt([w1, w2, w3])   */
Function Hom_Add(C1, C2):
    /* Cout=VecEncrypt([v1 + w1, v2 + w2, v3 + w3])           */
return Cout

Function Hom_Mul(C1, C2):
    /* Cout=VecEncrypt([v1 × w1, v2 × w2, v3 × w3])           */
return Cout

/* C1=VecEncrypt([v1, v2, v3]), W2=[w1, w2, w3]               */
Function Hom_Add(C1, W2):
    /* Cout=VecEncrypt([v1 + w1, v2 + w2, v3 + w3])           */
return Cout

Function Hom_Mul(C1, W2):
    /* Cout=VecEncrypt([v1 × w1, v2 × w2, v3 × w3])           */
return Cout

Function Hom_Rot(C1, rk, step = −1):
    /* Rotate C1 one step to the left:
       cout=VecEncrypt([v2, v3, v1])                          */
    /* Distinct rotation key rk is needed for different steps
       (the third input parameter).                           */
return Cout
```

As a remark, we note that in PIR the database content is public and encoded into plaintexts after the NTT computation. Therefore, PIR performs more plaintext-ciphertext multiplications (the ciphertext is the PIR query).

2.2 Private Information Retrieval (PIR)

Chor et al. [13] introduced private information retrieval (PIR) to address the following problem: for a database with N records, how can a client retrieve the k-th record without leaking k to the server?

There are two lines of PIR protocols: information theoretic PIR (IT-PIR) [7, 13–15] and computational PIR (CPIR) [1, 4, 11, 17, 40]. IT-PIR protocols replicate the database across multiple non-colluding servers. The client sends different queries to these servers and derives the answer (the desired database record) by combining the responses. IT-PIR protocols achieve information-theoretic security against adversarial attacks [13]. On the other hand, CPIR protocols put the database onto a single server, while guaranteeing security against computationally-bounded adversaries. In this work, we focus on the single-server CPIR because it is more practical than deploying non-colluding servers.

2.3 The state-of-the-art: FastPIR

Fig. 1 presents the dataflow overview of the FastPIR protocol [1], which contains the following steps for an example database with 4 records $[a, b, c, d]$, where the client wants to fetch the 3^{rd} record c. ① The client generates a query q which is an FHE ciphertext that encrypts a one-hot vector of length 4, where only the 3^{rd} slot in the vector is 1 and others are 0. ② The client then sends the query ciphertext to the server. ③ The server partitions the database into multiple vectors (columns) to facilitate the vector program in FHE. As a result, each record in the database is partitioned into 3 slices. ④ The server performs 3 `Hom_Mul` operations between the

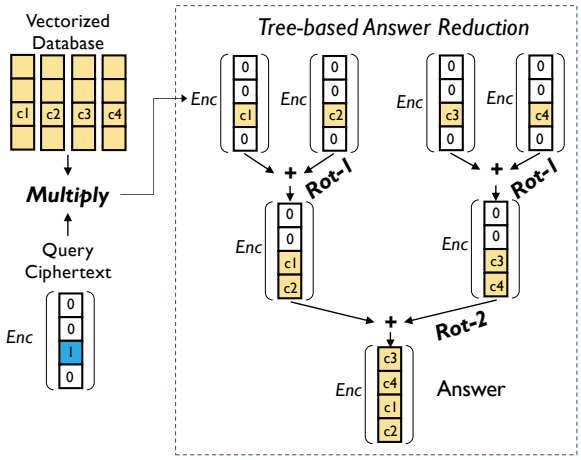

Figure 2: The tree-based answer reduction in FastPIR protocol to reduce the answer size using homomorphic rotations and additions.

query ciphertext and database vectors, resulting in three ciphertexts. ⑤ Through the `Hom_Rot` operation, the server shifts the record slices into different slots in the ciphertexts, which are then added together to achieve a single compact answer. ⑥ The server returns the answer to the client. ⑦ The client derives the desired record with `VecDecrypt`. Note that the record vector could be shuffled in any order, but the client knows the beginning of the record based on the index (three in this discussion).

A key aspect of FastPIR is a tree-based rotation scheme to perform the rotation operations efficiently. Fig. 2 shows a more detailed example of this scheme for records with 4 slices each. During the homomorphic multiplication step, a ciphertext is generated for each database column ($[0, 0, c1, 0]$, $[0, 0, c2, 0]$, $[0, 0, c3, 0]$, and $[0, 0, c4, 0]$). These ciphertexts are the leaves of a tree. Then the scheme aggregates the first two leaves (ciphertexts) having the same parent using a rotation with 1 step followed by an add operation. That is, the scheme rotates the ciphertext of $[0, 0, c2, 0]$ into $[0, 0, 0, c2]$, and then aggregates it with the first ciphertext to produce a ciphertext of $[0, 0, c1, c2]$. FastPIR follows this rotation-and-addition recursively to generate the root of the tree, which is a ciphertext of $[c3, c4, c1, c2]$ and contains all the slices of the desired record.

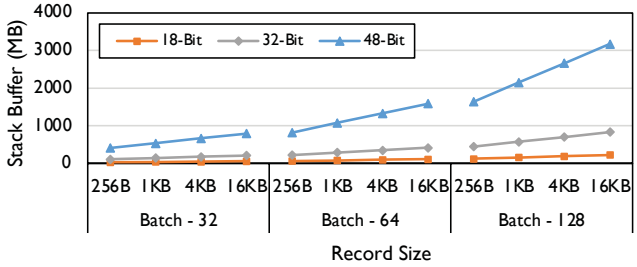

Figure 3: The size of stack used for the recursive tree-based rotation in FastPIR. We slice the records in the database into slices of 18/32/48-bits. We also vary the size of the records and the size of the number of queries being processed simultaneously (batch size).

2.4 Inefficiencies in FastPIR

Even though FastPIR achieves the best performance among existing PIR protocols, the scalability of FastPIR is still poor due to three inefficiencies:

(a) Large Query: As shown in Fig. 1, the query length in FastPIR grows linearly with the number of records. This results in an unacceptably long query for a large database. Considering the message box used in anonymous communication systems [5], the query can be as large as 27GiB for 1B users, which results in a significant query load time.

(b) Large Recursion Stack: The tree-based recursion in FastPIR is not hardware-friendly for a large tree due to the large buffer needed for the recursion stack. Fig. 3 illustrates the growth of stack size with the batch size (the number of PIR queries being processed simultaneously) and the number of database records. The database records are partitioned into slices of the width of 18, 32, and 48 bits for each column. We find that the stack size is significantly enlarged, e.g., over a GiB, for higher values of slice and record size.

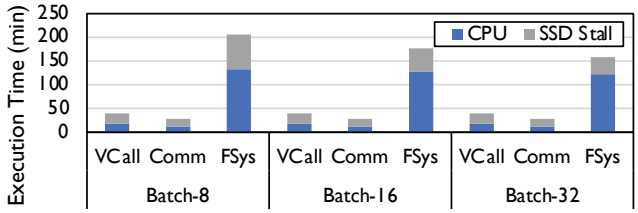

Figure 4: The execution time of FastPIR running on three workloads: Vcall (Voice Calling), Comm (Communication), and FSys (File System). The execution time is normalized by the batch size. The breakdown of CPU time and SSD stall is shown.

(c) Long SSD Stall: The PIR processing has to access and manipulate the entire database stored in SSD storage, which causes a severe performance bottleneck. Fig. 4 shows the breakdown of FastPIR's execution time by the CPU and SSD. We find that for the batch size of 8, 16, and 32, an average of 55.4%, 57.3%, and 28.6% of the execution is stalled on SSD. This indicates that SSD accessing is a major bottleneck in large-scale PIR processing.

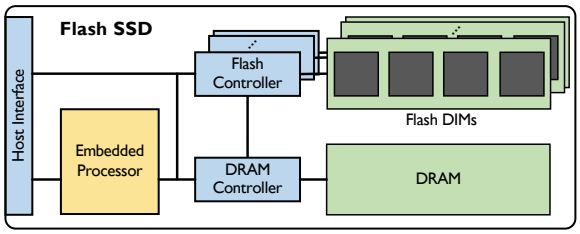

Figure 5: The architecture of flash SSD.

2.5 In-Storage Processing

The general architecture of modern SSD is shown in Fig. 5. The SSD storage usually contains a host interface, an embedded processor, DRAM, and flash memory DIMs. The host interface implements

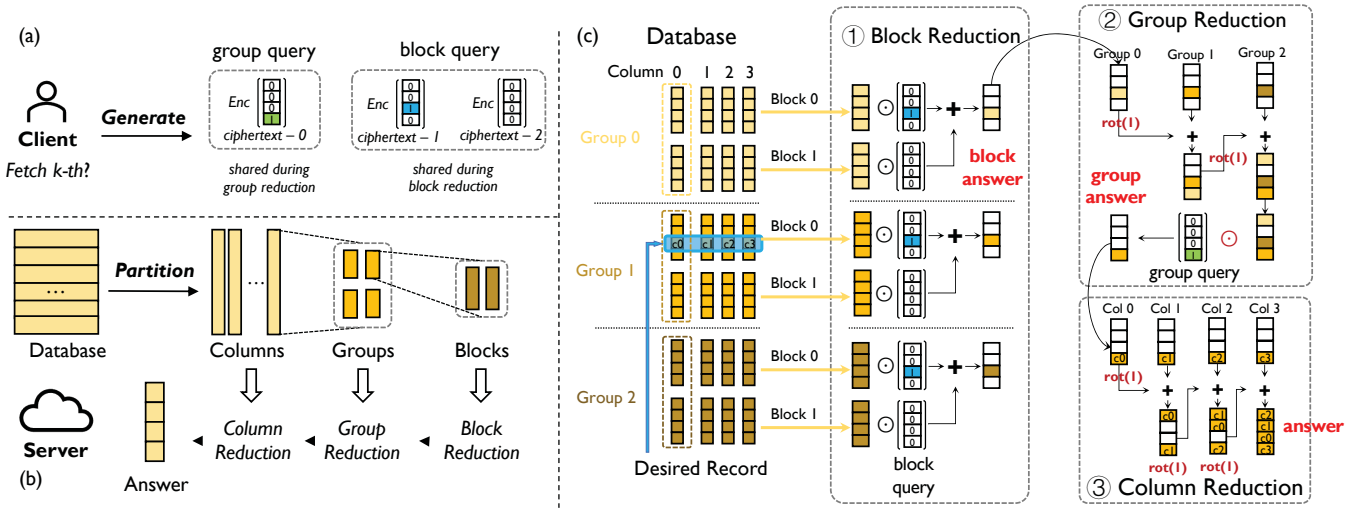

Figure 6: An illustration of the INSPIRE protocol. (a) On the client side, instead of generating a long query, the client generates one group query and one block query that consist of multiple ciphertexts for the desired record. (b) The database is hierarchically partitioned into columns, groups, and blocks. The server performs a multi-stage reduction process to derive the answer. (c) A detailed illustration of block reduction, group reduction, and column reduction. The block reduction traverses all the blocks in the database. The group reduction and column reduction shrink the answer with homomorphic rotation and addition.

the interface protocol, such as SATA or PCI express. When access requests come from the host interface, the embedded processor executes the Flash Translation Layer (FTL) to derive the physical address of the request. The embedded processor can be a RISC processor (such as ARM) that has limited computation capability. The processor further schedules these requests to flash controllers, which control multiple flash channels. The flash controller issues specific commands to the corresponding flash DIM to access the data. The DRAM can be used as a buffer to transfer the data from flash channels to the host.

SSD storage has been widely used for applications that require large memory capacity, such as industry-level neural network training [31] and large-scale graph processing [34]. However, accessing the SSD storage is much slower than memory, and the bandwidth of I/O is very limited. The latency of accessing flash SSD is about 50 μs, while the I/O bandwidth is up to 1.0 GiB/s per PCIe 3.0 lane. Therefore, in-storage processing emerges as a promising solution to overcome this performance gap, which leverages higher (4-8×) bandwidth and low latency internal to the SSD storage [59].

3 INSPIRE PROTOCOL

In this section, we introduce the INSPIRE protocol. We first introduce our design approach, followed by the protocol overview and optimizations.

3.1 Design Approach

Although the query in PIR can be extremely large, it contains many encryptions of the same data. As shown in Fig. 1, FastPIR encodes every unwanted record as a zero in the plaintext. However, naively reusing encryptions of these zeros will leak the query information to the server. Our INSPIRE protocol leverages the classical design of recursion [53]. The key idea is to hierarchically partition the database and reduce the query size by sharing the query in each hierarchy. Meanwhile, we keep the ciphertexts within the query independent from each other. Thus, the server cannot tell the difference between them, and the INSPIRE protocol can ensure the security guarantee of PIR.

Specifically, as shown in Fig. 6(a) and (b), we partition the database into *columns*, *groups*, and *blocks*. We design two types of queries: block query and group query. The query consists of different ciphertexts, encrypting the location of the desired record in the block/group. The ciphertexts within the query are not sharable, and thus the index information remains private to the server. By sharing the query at different data hierarchy, the server follows a multi-stage reduction process. At the lowest level, the protocol performs block reduction over the entire database, while using multiple ciphertexts to avoid the heavy rotation operations. At the middle level, the group reduction aggregates block answers with an identical group query for each column. At the top level, the protocol performs a standard FastPIR-like column reduction to derive the final answer.

Second, we further optimize the rotation-heavy computation pattern during the group reduction and column reduction. Instead of utilizing the tree based rotation, INSPIRE uses a streaming approach to facilitate the architecture design.

3.2 Protocol Overview

The INSPIRE protocol is composed by four functions at the client and server side: `DB_Partition`, `Query_Generate`, `Ans_Generate`, and `Ans_Decrypt`. These functions are described in detail in Alg. 3 and Alg. 4. First, `DB_Partition` is the initialization stage that partitions the database. The partitioning parameters, including block size and group size, are decided at the client-side using the `Param_Generate` function. Second, the client uses `Query_Generate`

Algorithm 3: INSPIRE Protocol – Client

```
/* Generate partitioning parameters                                  */
Function Param_Generate():
    // l_B – block length, w_B – block width
    // n_B – number of blocks in a group
    // l_G – group length, n_G – number of groups in a column
return l_B, w_B, n_B, l_G, n_G

/* Generate query to fetch k-th record                               */
Function Query_Generate(k):
    q_B = Vector < Vector (l_B) > (n_B) ;          // Block Query
    q_G = Vector(l_B) ;                            // Group Query
    // Find which block has the k-th record
    q_B_k = k % l_G;
    for i = 0 : n_B do
        for j = 0 : l_B do
            q_B[i][j] = (q_B_k == i * l_B + j) ? 1 : 0;
        end
        q_B[i] = VecEncrypt(q_B[i])
    end
    // Find which group has the k-th record
    block_idx = k % l_B ;                  // Index in block answer
    rot_offset = n_G − k/l_G; ;              // Number of rotations
    q_G_k = (block_idx + rot_offset) % l_B;
    for i = 0 : l_B do
        q_G[i] = (q_G_k == i) ? 1 : 0;
    end
    q_G = VecEncrypt(q_G)
return q_B, q_G

/* Decrypt the answer returned from server                           */
Function Answer_Decrypt(ans):
    msg = VecDecrypt(ans); ;                   // Decrypt answer
    /* Reorder the data in the msg vector                            */
return msg
```

Algorithm 4: INSPIRE Protocol – Server

```
/* Partition a database with certain params                          */
Function DB_Partition(db, l_B, w_B, n_B):
    L, W = db.length, db.width;
    n_C = W / w_B;
    n_G = L/(l_B × n_B);
    /* Three indices are needed to locate a block: column id,
       group id, block id                                            */
return n_G, n_C

/* Generate answer at the server side                                */
Function Answer_Generate(q_B, q_G):
    Initialize ans, ans_block, ans_group;
    for c = 0 : n_C do
        for g = 0 : n_G do
            for b = 0 : n_B do
                // ❶ Block Reduction
                db_block = db[(g*n_B+b)l_B : (g*n_B+b+1)l_B, c];
                db_block =Hom_Mul(db_block, q_B[b]);
                ans_block = Hom_Add(ans_block, db_block);
            end
            // ❷ Group Reduction
            ans_group = Hom_Add(ans_group, ans_block);
            ans_group = Hom_Rot(ans_group, 1);
        end
        // ❸ Column Reduction
        ans_group = Hom_Mul(ans_group, q_G);
        ans = Hom_Add(ans, ans_group);
        ans = Hom_Rot(ans, 1);
    end
return ans
```

with the assigned index to generate the query ciphertexts and send them to the server. Third, when the server receives a query, it performs the data retrieval using `Answer_Generate` and sends back the result to the client. Finally, the client uses `Ans_Decrypt` to decrypt and re-arrange the record in the received answer.

`DB_Partition`: INSPIRE adopts a hierarchical database partitioning scheme by arranging the database into blocks, groups, and columns. Specifically, on the server side, the database can be viewed as a 2D matrix with the shape of $N \times n_C$. Each row in the database stands for a record, and each record can be further divided into n_C pieces along the column direction. Suppose we have the ciphertext with length l_B. Then, for each column in the database, every continuous l_B elements are considered as a block. There are N/l_B blocks for each column. Further, several blocks are considered as a group along the row direction. We denote the number of blocks in a group as n_B and the number of groups in a column as n_G, where $l_B \times n_B \times n_G = N$. As the example shown in Fig. 6(c), the database is reshaped as a 24×4 matrix. We use the blue label to indicate the record the client is interested in. Here, 4 elements are encrypted together as a plaintext, and this plaintext is considered as a block. Each column in the database contains 6 blocks. Based on our leveled database partition, each column in the database is divided into 3 groups and each group contains 2 blocks.

`Query_Generate`: The query in INSPIRE also follows the hierarchical scheme and is composed of a block query and a group query. The block query includes n_B ciphertexts, where one of them encrypts a one-hot vector and others independently encrypt all-zero vectors. The group query includes a single ciphertext that indicates which group holds the desired record. The block query traverses the entire database with relatively simple operations, and INSPIRE streams all the blocks in the database with minimal hardware. The query size of INSPIRE is much smaller than FastPIR because both the block query and group query are shared, unlike FastPIR which has a separate ciphertext for every block in a column.

`Ans_Generate`: During the data retrieval, the server takes the encrypted query from the client and generates an encrypted answer. INSPIRE adopts a three-stage processing method to compute the answer on the server: block reduction, group reduction, and column reduction. We explain the process flow of these stages in detail next.

3.3 Multi-Stage Answer Generation

Block Reduction: Each group performs the block reduction with the shared block query. During the reduction, each block in the database performs `Hom_Mul` with the corresponding ciphertext in the block query. Then the resulting ciphertexts within a group are

aggregated through Hom_Add to get the block answer. As shown in Fig. 6(c), each group has two blocks, and these two blocks are multiplied by the two ciphertexts in the block query. Therefore, the shared block query has to traverse the entire database, but the resulting homomorphic operations are relatively simple with only one multiplication and addition for each block.

Group Reduction: Through block reduction, we already derive the desired record in Group 1. But we still have unnecessary data in Group 0 and Group 2. Thus, group reduction aggregates and eliminates this data. As shown in Fig. 6(c), the protocol has three block results shifted by different steps via Hom_Rot operations. Then, it adds them together using Hom_Add. The group query then multiplies with this aggregated answer, which produces the group answer. The group answer keeps only a piece c_i of the desired record in the i-th column.

Column Reduction: The reduction process in the column reduction phase is very similar to the group reduction. We shift the ciphertexts with different steps, and the final answer is the aggregation of the group answers. In this example, we have the same number of columns as ciphertext length ($l_B = n_C$), and the record C exactly fills the result ciphertext. In case we have more columns than ciphertext length, we can simply use longer ciphertext or multiple ciphertexts.

Optimized Rotation Flow: As shown in Fig. 6(c), our rotation flow is different than the tree based reduction in FastPIR. Instead, we use an answer ciphertext and perform in-place computation. When we need to aggregate the answer ciphertext with the next block/group result, we rotate the answer ciphertext by 1 and directly add the new ciphertext to it. We call this the *RNA* (rotation and add) scheme. It has two benefits: first, it eliminates the large recursion stack used for the tree traversal, and we only need a small buffer to store the answer ciphertext on-chip. Second, all the rotation operations are performed with $step = 1$. Therefore, we use only one rotation key and avoid a large buffer for storing different keys.

3.4 Complexity Analysis

Query Complexity: Our INSPIRE query consists of a block query and a group query. The block query has the size of $n_B \times l_B \times M$, where n_B is the number of blocks per group, l_B is the block length, and M is the size of the polynomial coefficient. The group query has the size of $l_B \times M$. Thus, the total query size is $(n_B+1) \times l_B \times M$. Compared with FastPIR whose query size is $N \times M$, we reduce the query size by a factor approximately equal to the number of groups n_G.

Table 1: Computation Complexity of different reduction stages in INSPIRE, with comparison to FastPIR

	Hom_Add	Hom_Mul	Hom_Rot
Block Reduction	$n_C n_G(n_B-1)$	$n_C n_G n_B$	0
Group Reduction	$n_C(n_G-1)$	n_C	$n_C(n_G-1)$
Column Reduction	n_C-1	0	n_C-1
INSPIRE	$n_C n_G n_B-1$	$n_C n_G n_B+n_C$	$n_C n_G-1$
FastPIR	$n_C n_G n_B-1$	$n_C n_G n_B$	n_C-1

Computation Complexity: In Tab. 1, we show the number of Hom_Mul, Hom_Rot, and Hom_Add operations in different reduction phases. The total operations in FastPIR are also shown as a comparison. We find that our INSPIRE has the same number of homomorphic additions and slightly increases n_C homomorphic multiplications. On the other hand, we increase homomorphic rotations by n_G times. However, although the total rotations are increased, the rotation operation in INSPIRE is cheaper than FastPIR. Importantly, all rotations in INSPIRE are processed in a streaming fashion, and thus we avoid the expensive recursion stack of FastPIR (Fig. 3).

4 INSPIRE ARCHITECTURE

Although our INSPIRE protocol significantly reduces the size of the query, we still expect a large amount of accesses to database blocks. In order to overcome the bandwidth bottleneck at the storage I/O, we further design the INSPIRE architecture, an in-storage processing accelerator. In this section, we first present the design overview of INSPIRE architecture. Then, we present the implementation details and dataflow design in the different hardware units.

4.1 Architecture Overview

The overall architecture of INSPIRE is shown in Fig. 7, which is based on the original flash SSD. During the data retrieving, both the block query and blocks of database records are loaded sequentially from the Flash DIMs. The block reduction is performed at the block collector, leveraging the channel-level parallelism to generate block answers. Further at the answer accelerator, block answers are aggregated together to generate the group answer with the group query. And finally, we use column reduction to derive the final answer and send it back through the host interface. Therefore, INSPIRE adopts a heterogeneous architecture. This subsection details how INSPIRE architecture handles the memory-bounded block reduction and computation-bounded group/column reduction.

The Block Collector is located beside each flash controller. Because block reduction needs to traverse the entire database, we locate it near memory to leverage the internal memory bandwidth. All the groups are interleaved in different flash channels, such that each block collector processes blocks in a group without inter-channel communication. In the Block Collector, the block query and block answer are stored in the global buffer (GLB), and database blocks are streamed in from the flash DIM. The required homomorphic operations for block reduction can be realized through a number theory transform (NTT) unit and a MAC array. We will detail the NTT units in the following subsections. After aggregating block answers in a group, the Block Collector sends the group answer to the DRAM.

The Answer Accelerator is located beside the SSD controller to execute the computation-bounded group/column reduction stages, which contain complex homomorphic rotation. During the group reduction stage, the block answers from different flash channels are loaded from DRAM to the GLB in Answer Accelerator. Then, the *RNA* (Hom_Rot and Hom_Add) based combination is achieved through the permutation unit, NTT unit, and MAC array. After the group answer is acquired from group reduction, all the group answers in the same column are combined into the final answer with *RNA*, which follows the same computation scheme as the *RNA* in group reduction.

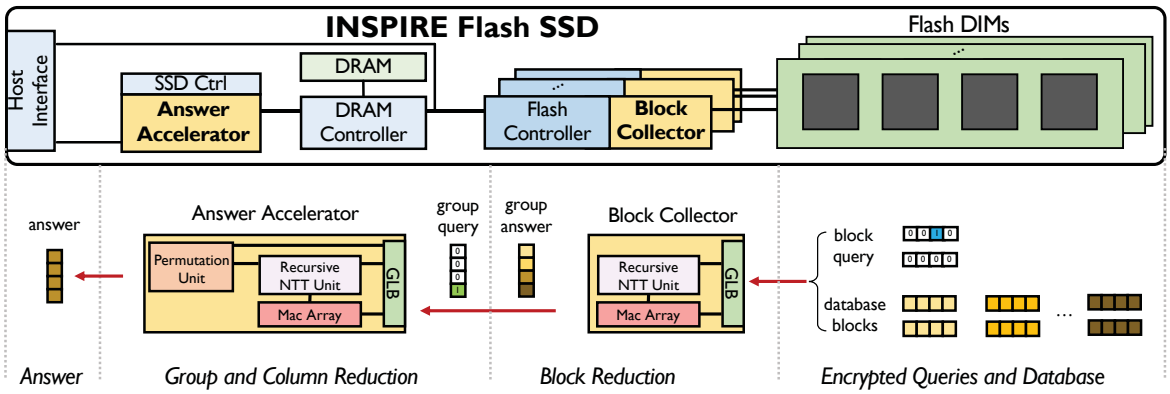

Figure 7: Architecture of INSPIRE. Queries and database are stored on Flash DIMs. INSPIRE adopts a heterogeneous architecture to execute block reduction and group/column reduction on block collectors and answer accelerator respectively. The encrypted answer is finally exported to host through the host interface.

4.2 MAC Array

The MAC array in both the Block Collectors and the Answer Accelerator is a SIMD unit consisting of a group of MACs, and the MAC performs modular addition and multiplication. Different from other applications, the data (such as polynomial coefficients) are bounded by a modulus p, i.e., an integer modulo p. p can be a large prime number or a product of multiple prime numbers [51, 52]. Therefore, modular adder and multiplier are needed when we perform computations such as $(a \cdot b) \bmod p$ and $(a + b) \bmod p$.

We follow the design in F1 [51] for modular arithmetic. In particular, F1 adopted the Montgomery multiplier [45] for fast modular multiplications. It also reduced the total number of stages in the multiplier by choosing an appropriate modulus p.

4.3 Recursive NTT Unit

NTT operations are the dominant operations in `Hom_Mul` and `Hom_Rot` [51]. The NTT computation is the same as Discrete Fourier Transform (DFT) but over a polynomial ring. As shown in Fig. 8(a), we design the recursive NTT unit to compute the NTT transform of an input vector X with arbitrary length. The recursive NTT unit is composed of a fixed input NTT unit, a twiddle unit, and a transpose unit. Specifically, the fixed NTT unit is implemented as a customized butterfly hardware which takes a length-N input and computes the length-N NTT result. The twiddle unit scales the results from the NTT unit using MACs. The transpose unit transposes the results through a crossbar. We put register buffers at each level of the butterfly hierarchy such that the NTT unit is pipelined for streaming the input. As shown in Figure 7, both the Block Collector and the Answer Accelerator contain an NTT unit, but with different sizes to facilitate unique throughput requirements.

Break down Long-Sequence NTT: Since the ciphertext polynomial is usually quite large, with length from 4K to 16K, it is infeasible to directly implement an NTT unit at this scale. Thus, we need to efficiently map a long polynomial onto smaller NTT units.

INSPIRE recursively adopts a 2D NTT algorithm to achieve such mapping [41]. The key idea of 2D NTT is that we can break down a length-$L = m \times n$ NTT into smaller length NTTs, by performing length-m NTT n times and performing length-n NTT m times. More specifically, the 2D NTT has the following steps: ❶ Reshape the length-L vector into an $m \times n$ matrix. ❷ Perform m-input NTT for every column, and multiply the result with a known parameter (called twiddle factor). ❸ Perform n-input NTT unit for every row. ❹ Finally, reshape the matrix back to length-L vector.

Fig. 8(b) shows an example of this algorithm. Suppose we need to do a 32-input NTT on a vector X. First, X is reshaped into a 4×8 matrix. Then, we feed each column into a 4-input NTT unit and

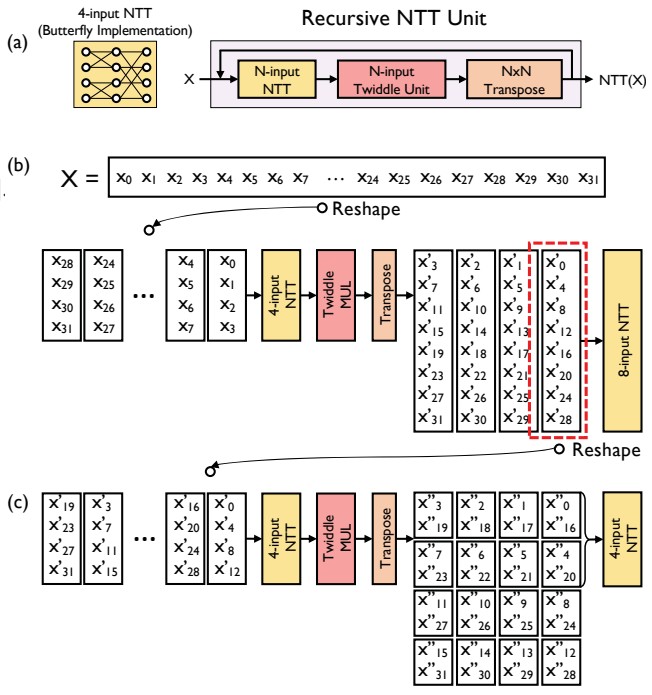

Figure 8: (a) The architecture of block collector that implements butterfly computation. (b) The dataflow of breaking down long-sequence NTT to 2D NTT. (c) We further conduct the 2D NTT recursively.

have them go through the twiddle unit. Finally, we transpose the matrix and perform 8-input NTT for each column (with length 8).

Recursive Processing: The naive 2D NTT still suffers from inflexibility, as the length of the input ciphertext may vary in different applications. To address this problem, INSPIRE further supports a recursive NTT scheme to perform NTT computation with arbitrary vector length. The key idea is to apply the 2D reshaping recursively for each dimension until the NTT size can fit into our hardware NTT unit. We reshape the length-L NTT as a N^d tensor (instead of a matrix), where N is the length of our hardware butterfly. Thus, we apply the 2D algorithm recursively for a long sequence NTT computation. We show an example of our recursive NTT in Fig. 8(c). For the 8-input NTT desired in Fig. 8(b), we keep reshaping this vector into a 4×2 matrix. Then we can apply another 2D NTT to compute the result with the same 4-input NTT unit. Note that we can still use the 4-input NTT unit to compute a 2-input NTT, since the data in the butterfly is decouplable. Our NTT hardware can be programmed to compute two 2-input NTTs simultaneously.

4.4 Permutation Unit

Permutation Operation: The permutation unit is essentially a switch to reorder the data in the `Hom_Rot` operation. The permutation step in Hom_Rot is to map all the polynomial coefficients to different positions. For example, we can permute the polynomial $2x^3+5x^2+7x+1$ into $5x^3+7x^2+x+2$. A more formal way to describe the permutation is as follows: Suppose the length of the ciphertext vector is N, and we use i and i' to indicate the old and new index of a coefficient before and after permutation. To rotate the plain vector by r steps, the permutation is performed as $i' = (i \times k^r)\ mod\ N$, where k is co-prime with $2N$.

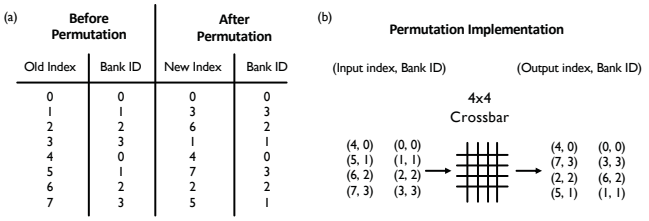

Figure 9: Data mapping of the permutation in Hom_Rot. Here, $N = 4096$, $B = 8$, $k = 3$.

It is guaranteed that each coefficient will go to a unique position. Thus, a crossbar switch is enough to route all the data in one cycle (because there is no destination conflicts). However, since the ciphertext can be as large as 4K-16K (Sec. 4.3), it is infeasible to have such a large crossbar.

Key Insight: Our key finding is that such destination conflicts do not exist for every continuous power of 2 coefficients. Fig. 9 shows an example of this interesting property. Assume that we have 8 coefficients stored in 4 (=2^2) banks. After permutation, the data in position (0,1,2,3,4,5,6,7) now goes to position (0,3,6,1,4,7,2,3). For the 4 coefficients located in bank (0,1,2,3), the new bank ID is (0,3,2,1). There is no bank conflict to relocate these 4 data elements. Therefore, we use a small crossbar with a size of 4 to process a long permutation. To permute a 4096-length polynomial, we can directly permute (relocate) the data 0-3, 4-7, ..., 4092-4095 in a sequential order, where each permutation is done in one cycle.

5 EVALUATION

In this section, we evaluate the performance of the INSPIRE protocol and architecture. We first introduce our evaluation methodology. Then, we show the performance gain from INSPIRE. Then, we evaluate the scalability and sensitivity of our design. Finally, we present the area and power overhead of INSPIRE.

Table 2: INSPIRE Architecture Configurations

SSD Device			
Read/Program/Erase Latency		75/750/3800 *ns*	
Channel-Chip-Die-Plane		16-4-2-2	
Blocks/Plane	2048	Pages/Block	512
Channel Width	1B	Channel Rate	1033MT/s
Page Size	8KiB	Capacity	2TiB
Host Interface	PCIe 3.0 ×4	Flash Protocol	NVDDR3
Answer Accelerator (AA) and Block Collector (BC)			
Tech Node	28nm	Frequency	400MHz
Xbar Switch (AA)	4×4	Operand	32b
NTT input size (AA)	32	NTT input size (BC)	8
Total mMuls (AA)	160	Total mMuls (BC)	24
Total mAdds (AA)	224	Total mAdds (BC)	32
Transpose unit (AA)	32×32	Transpose unit (BC)	8×8
GLB (AA)	2MiB	GLB (BC)	1.25MiB

5.1 Methodology

Software Implementation: We implemented the INSPIRE protocol using the Microsoft SEAL library [52]. We choose the BFV encryption scheme [9, 18] and selected the BFV parameters to provide the highest security level according to the Homomorphic Encryption Standard [3]. We stress that our protocol also works with other FHE schemes such as BGV [10] and CKKS [12]. Our implementation uses multiple threads for answer generation using OpenMP, to fully leverage the data-level parallelism.

Hardware Implementation: We implemented the INSPIRE architecture logic in RTL and synthesized it with Design Compiler and 28 *nm* technology node to derive the hardware parameters, including timing, power, and area. We built a cycle-accurate simulator on top of MQSim [56], an NVMe/SATA SSD simulator, to model the performance of software-hardware co-optimized INSPIRE.

Configurations: We configured the SSD device similar to our CPU baseline, as shown in Table 2. The hierarchy in the SSD is organized as channel-chip-die-plane-block-page. With a page size of 8KiB, the total capacity of SSD is 2TiB. The page read and program latency for LSB/CSB/MSB are 75 and 750 *ns*, respectively. The block erase latency is 3800 *ns*. Each flash channel is equipped with the NVDDR3 protocol, providing a channel width of 1B and a transfer rate of 1033MT/s. The host communicates with the SSD using PCIe 3.0 ×4, with an ideal bandwidth of 4GiB/s.

For the INSPIRE architecture configurations, we set the input size of NTT units to 32 and 8 for the answer accelerator (AA) and each block collector (BC), respectively. An X input NTT unit is realized through $X(log_2X - 1)/2$ modular multiplications and $Xlog_2X$ modular additions (mAdds). The answer accelerator has

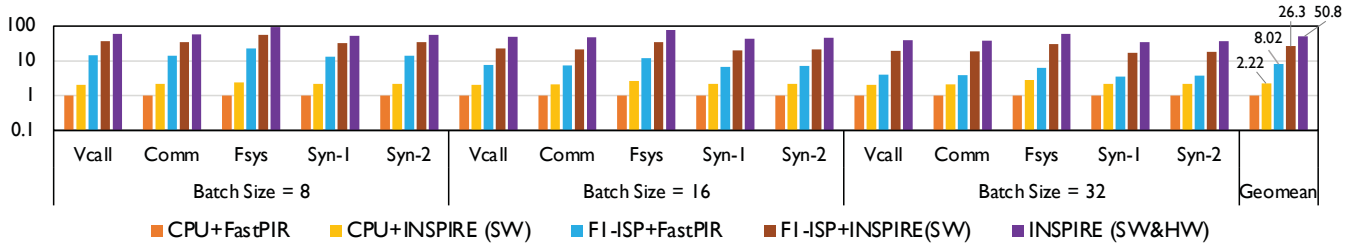

Figure 10: The overall performance of INSPIRE compared against FastPIR and F1. We evaluate our protocol and FastPIR on both CPU and F1-ISP platforms. The results are normalized to the FastPIR on CPU baseline. Five workloads and three batch sizes are used.

Table 3: Database Workloads

	Record Length	Num. Records	Database Size
Voice Calling (VCall)	96B	2^{32}	384GB
Communication (Comm)	288B	2^{30}	288GB
File System (FSys)	10MB	2^{17}	1.25TB
Synthetic DB 1 (Syn-1)	1KB	2^{29}	512GB
Synthetic DB 2 (Syn-2)	18KB	2^{26}	1.13TB

160 mMuls and 224 mAdds in total (in the twiddle unit and MAC array). Each block collector involves 24 mMuls and 32 mAdds. The transpose unit sizes for the answer accelerator and block collector are 32×32 and 8×8, respectively. The answer accelerator needs an additional 4×4 Xbar switch to realize the homomorphic rotation. The global buffers for the answer accelerator and block collector are 512KiB and 64KiB, respectively. For homomorphic parameters, we set the element in plain vector to 18bits and the coefficient in the ciphertext to 109bits. After the RNS decomposition [19], each coefficient in ciphertext is composed of 4 32-bit numbers. Moreover, the mMul is realized through optimized Montgomery multiplier [42, 51] which simplifies the complex modular multiplication.

FHE Parameters: We set the polynomial degree as 4096 (l_B = 4096 in Algorithm 4). We use the security level of $\lambda = 128$ bits and 109-bit ciphertext coefficient, which follows the same setting as in FastPIR. The plain message to be encrypted is set to 18-bits. The INSPIRE protocol has a multiplication depth of 2 in total.

Workloads: Table 3 summarizes the characteristics of the database workloads we evaluate. We run the PIR protocol for three applications: voice calling [1], anonymous communication [4, 5], and file system [40]. The main difference between these workloads is the record type, which determines the record length. We scale the number of records to evaluate the storage-based database. In addition, we have two synthetic databases, Syn-1 and Syn-2, to enhance the workload diversity.

Baselines: We use FastPIR [1] as our software baseline, since it demonstrates the best performance among existing PIR schemes including XPIR [40] and SealPIR [4]. For a fair comparison, both FastPIR and INSPIRE use the BFV encryption scheme with the same security level. We also optimize and parallelize the source code of FastPIR with OpenMP.

We use both CPU and FHE accelerator as our hardware baseline. The CPU platform is a 64-thread Ryzen Threadripper 3970X @ 2.2GHz (3.7GHz with turbo boost). The storage is a 2TiB Intel 660p NVMe SSD interfaced with PCIe 3.0 ×4. The measured bandwidth of the SSD is 1.8GiB/s. We also include F1 [51], the state-of-the-art FHE accelerator as an additional baseline. We implemented F1 with the ISP architecture, where we attach F1 to the DRAM buffer inside the SSD. The DDR4 DRAM has an ideal bandwidth of 12.8GiB/s.

5.2 Performance

In this section, we present key results of INSPIRE, including the overall performance, network bandwidth reduction, noise growth, and query size.

Overall Performance: Fig. 10 shows the overall performance of INSPIRE, alongside a comparison with FastPIR and F1. The results are shown across three batch sizes: 8, 16, and 32. As an intuitive example, FastPIR spends 28.4min on average to process a Comm query (Fig. 4). Our INSPIRE protocol only takes 13.3min for the same query, and our INSPIRE architecture further reduces the time to 36s.

At the software level, the INSPIRE protocol achieves 2.22× performance speedups against FastPIR on CPU, with 3.28× speedup on in-storage F1 (F1-ISP). The performance gain of INSPIRE protocol mainly comes from the reduced memory access for queries. Also, INSPIRE simplifies the dataflow in rotations, which avoids the memory overhead caused by the large recursion stack. Moreover, the INSPIRE protocol demonstrates better performance in the ISP architecture of F1-ISP. The key reason is that the small query and rotation buffer required by INSPIRE are more friendly to accelerator architectures, which usually have limited on-chip resources. For FastPIR, it is much more time-consuming to wait for tiled queries from the host.

At the hardware level, the INSPIRE architecture shows 22.9× speedup against the CPU baseline, with 1.93× speedup compared with F1-ISP. The performance gain of INSPIRE mainly lies in two aspects: first, we leverage much higher aggregated bandwidth to process the memory-bound workloads. While F1-ISP utilizes the bandwidth from the DRAM buffer, the heterogeneous architecture of INSPIRE better leverages the channel-level parallelism via the block collectors in the flash channels. Second, the INSPIRE architecture is tightly coupled with and specialized for the protocol. Different reduction stages are pipelined across block collectors and the query accelerator. Thus, we avoid the sophisticated data mapping and communication in an F1-like architecture.

Network Bandwidth: The network bandwidth directly shows the communication efficiency with compact queries. Fig. 11 shows

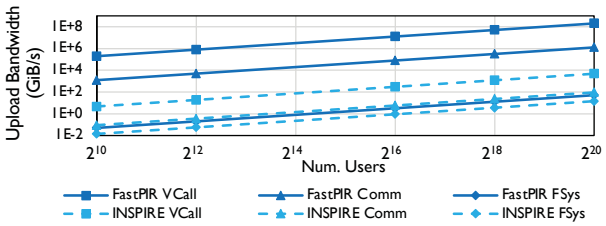

Figure 11: The upload traffis of INSPIRE and FastPIR, which depends on the query size and query frequency.

the upload traffic of the INSPIRE protocol, with a comparison to FastPIR. The x-axis shows the number of users scaling from 2^{10} to 2^{20}. We find that INSPIRE reduces the upload bandwidth by 41943×, 13706×, and 3.56× for VCall, Comm, and FSys, respectively. The significant bandwidth reduction for VCall and Comm is because these two workloads have a very large number of records, and our hierarchical query substantially decreases the total query size. Also, we find that VCall requires much higher (52–333×) upload bandwidth than the other two applications. The reason is that users in VCall have to frequently query the database to fetch the newest message, leading to more simultaneous uploads. Finally, since PIR protocol does not share query data across users, the upload traffic increases linearly as the number of users grow for both FastPIR and INSPIRE.

5.3 Sensitivity Study

In this section, we study the scalability of INSPIRE, along with the performance sensitivity to different security levels.

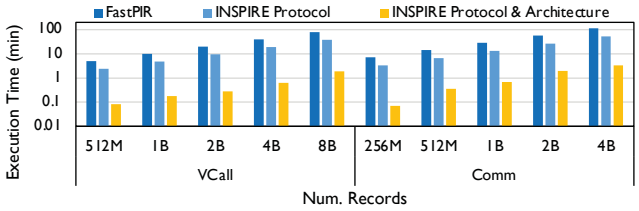

Figure 12: The scalability of FastPIR and INSPIRE when the number of records (i.e., the size of database) increases. Two workloads, VCall and Comm, are used for evaluation. The batch size is set to 32.

Scalability: We analyze the scalability of FastPIR and INSPIRE with the variants of VCall and Comm workloads. As shown in Fig. 12, the number of records in the database scales from 512M to 8B for VCall, and 256M to 4B for Comm. A large batch size of 32 is used, and the absolute execution latency is presented. Compared to FastPIR, we find that the INSPIRE protocol demonstrates 2.08× better performance. The INSPIRE architecture offers an additional performance gain of 49.8×. Further, the scaling of INSPIRE is approximately linear. This is because PIR applications are memory-bound. When the size of the database grows, the accessed data and the number of compute operations increase accordingly. Also, we observe that FastPIR spends tens of minutes processing a query. In comparison, our INSPIRE architecture considerably reduces the processing time to the second-level. This makes it possible to deploy PIR in real database systems.

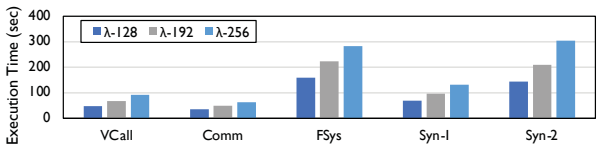

Figure 13: The performance of INSPIRE as a function of different security levels, where $\lambda = 256$ denotes the highest security level and $\lambda = 128$ denotes the lowest security level.

Sensitivity to Security Level: We study how different security parameters impact the performance of INSPIRE. As shown in Fig. 13, we choose three security levels that are provided by SEAL: $\lambda = 128 - bit$, $\lambda = 192 - bit$, and $\lambda = 256 - bit$. The $\lambda = 256 - bit$ gives the strongest security guarantee [3]. We find that by applying a higher security level of $\lambda = 192 - bit$ and $\lambda = 256 - bit$, the performance will be downgraded by 1.40× and 1.89×, respectively. This is because with larger λ, the coefficient size in the plaintext is reduced, and thus the message that can be encrypted is smaller. Therefore, the database has to be partitioned at a finer granularity and thus takes more time to process.

5.4 Area and Power Overhead

Table 4 presents the area and power overhead of the INSPIRE architecture, broken down into each hardware component. We find that computation logics, including the NTT engine and MAC array, take 14.44% of the area and 19.08% of the power. The buffer and routing units, including the transpose unit, GLB, and the crossbar switch occupy 85.56% and 80.92% of the total area and power, respectively. Most of the area and power consumption is taken by the GLB which is used to store the block query in block collector and the temporal answers in the answer accelerator. Note that in the INSPIRE architecture there are 8 block collectors, and the results in Tab. 4 accumulate the resource consumption for all block collectors.

Table 4: Area and Power Estimation.

	Area (mm^2)	Power (mW)		Area (mm^2)	Power (mW)
NTT Engine	2.745	1183.59	**Transpose Unit**	0.122	115.87
mMAC Array	2.954	1393.74	**Global Buffer**	31.592	12385.56
Xbar Switch	0.001	0.40	**Control&Others**	0.055	52.25
Block Collector	33.722	13509.07	**Answer Accelerator**	5.838	2438.29
Total Area 39.56mm^2; Total Power 15.947W					

6 DISCUSSION

Compatibility with non-private queries. While INSPIRE utilizes specialized hardware to process the PIR queries, normal SSD accessing commands (such as read and write) can still be executed. Therefore, traditional non-private queries are also compatible with INSPIRE. In real-world applications, a small portion of queries may contain sensitive information and thus require to be processed privately. For these queries, INSPIRE offers a significantly improved performance than existing PIR schemes.

Endurance of SSD. SSD devices have finite endurance, meaning that the total number of writes to the device is fixed [25]. Traditional ISP architectures usually introduce more writes to SSD devices and thus could suffer from the endurance issue [27]. In comparison, the database in INSPIRE remains static in the protocol, and we avoid massive SSD writes by reducing the query size and buffering the query on-chip.

7 RELATED WORK

PIR Protocol Design. A long line of research papers have reduced the communication overhead in PIR schemes. Stern proposed recursion to reorganize the database [53], and this scheme has been broadly used in XPIR [40] and other works [29, 37]. SealPIR also reduces the answer size by directly splitting the database into columns and combining the answer from different columns [4].

However, directly applying recursion introduces s significant overhead, which comes from rapid noise growth and massive reduction operations. INSPIRE provides a scalable multi-stage processing solution to reduce query size and atomize computation overhead. INSPIRE also optimizes rotation operations during the answer reduction stage. Finally, INSPIRE engages a hardware-friendly dataflow, where different reduction stages are pipelined to facilitate the architecture design.

In-Storage Processing. Previous studies leverage in-storage processing architecture to accelerate applications that explore large memory capacity and have memory-bound characteristics [16, 26, 31, 34, 50, 58, 62]. Different from prior work that tends to put the accelerator near the storage, the INSPIRE architecture takes a more intrusive approach that fully customizes the SSD system. INSPIRE directly integrates lightweight processors (block collector) at each flash channel and thus enjoys much higher aggregated bandwidth.

Oblivious RAM (ORAM). Both PIR and ORAM are two cryptographic primitives to hide the access pattern while accessing cloud data. ORAM assumes a setting where a client owns the data, outsources it, and then later queries it. In contrast, PIR works in a setting where the data is public, and the query has to be hidden. Thus, these schemes target different scenarios: private data versus public data.

Homomorphic Encryption Accelerators. Fully homomorphic encryption (FHE) is a core primitive for private computing. Many works using FHE focus on deep learning applications that are dominated by multiplications [43, 47–49, 51, 54, 57, 61]. These works design efficient hardware for specific operators in FHE, such as NTT transform and polynomial multiplication. INSPIRE distinguishes itself from these works with two unique features. First, the PIR application exhibits memory-bound characteristics caused by the need to traverse the entire database. INSPIRE leverages the in-storage processing architecture for higher throughput. Second, INSPIRE pays particular attention to a heterogeneous architecture that is customized for PIR's dataflow.

In-Memory Processing. In-memory processing technique is another approach to solve the memory-bound problem, which instead puts the computation logic inside the main memory [2, 6, 22, 28, 30, 33, 36, 38, 46, 63]. Compared with in-storage processing, it explores higher bandwidth provided by DRAM/HBM devices. However, we design the INSPIRE architecture based on the processing-in-storage technique due to the smaller capacity concern of memory. As a database usually scales to the order of TB~PB, SSD storage with a larger capacity is more suitable for PIR applications.

8 CONCLUSION

This work follows a software-hardware co-design approach to address the performance bottleneck in existing PIR schemes. We first present the INSPIRE protocol that leverages hierarchical database partitioning and multi-stage answer reduction to reduce the communication overhead in PIR. Based on the protocol, we present the INSPIRE architecture, an in-storage processing architecture that utilizes the large internal bandwidth and a specialized accelerator to boost the performance of query processing. The INSPIRE protocol achieves 2.22× performance speedup compared to the state-of-the-art FastPIR scheme. Meanwhile, the INSPIRE architecture further brings 22.9× performance speedup over CPU and 1.93× speedup over F1, the state-of-the-art FHE accelerator.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their deep insights and our shepherd Dr. Mithuna Thottethodi for their constructive comments that helped us improve the paper. This work is supported by the National Science Foundation under Grant No. 2124039. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2021. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Annual International Symposium on Computer Architecture*. 105–117.

[3] Martin R Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin E Lauter, et al. 2019. Homomorphic Encryption Standard. *IACR Cryptol. ePrint Arch.* 2019 (2019), 939.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. 2018. PIR with compressed queries and amortized query processing. In *IEEE symposium on security and privacy (SP)*. 962–979.

[5] Sebastian Angel and Srinath Setty. 2016. Unobservable communication over fully untrusted infrastructure. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 551–569.

[6] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems. In *IEEE/ACM international symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[7] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and J-F Raymond. 2002. Breaking the O (n/sup 1/(2k-1)/) barrier for information-theoretic private information retrieval. In *IEEE Symposium on Foundations of Computer Science*. IEEE, 261–270.

[8] Amos Beimel, Yuval Ishai, and Tal Malkin. 2000. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Annual International Cryptology Conference*. Springer, 55–73.

[9] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual Cryptology Conference*. Springer, 868–886.

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[11] Yan-Cheng Chang. 2004. Single database private information retrieval with logarithmic communication. In *Australasian Conference on Information Security and Privacy*. Springer, 50–61.

[12] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[13] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private information retrieval. In *IEEE Annual Foundations of Computer Science*. 41–50.

[14] Daniel Demmler, Amir Herzberg, and Thomas Schneider. 2014. RAID-PIR: Practical multi-server pir. In *ACM Workshop on Cloud Computing Security*. 45–56.

[15] Casey Devet, Ian Goldberg, and Nadia Heninger. 2012. Optimally robust private information retrieval. In *USENIX Security Symposium (SEC)*. 269–283.

[16] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. 2013. Query processing on smart SSDs: Opportunities and challenges. In *ACM SIGMOD International Conference on Management of Data*. 1221–1230.

[17] Changyu Dong and Liqun Chen. 2014. A fast single server private information retrieval protocol with low communication cost. In *European symposium on research in computer security*. Springer, 380–399.

[18] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* (2012).

[19] Harvey L Garner. 1959. The residue number system. In *Papers presented at the the March 3-5, 1959, western joint computer conference*. 146–153.

[20] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Stanford university.

[21] Matthew Green, Watson Ladd, and Ian Miers. 2016. A protocol for privately reporting ad impressions at scale. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1591–1601.

[22] Peng Gu, Xinfeng Xie, Shuangchen Li, Dimin Niu, Hongzhong Zheng, Krishna T Malladi, and Yuan Xie. 2020. DLUX: a LUT-based Near-Bank Accelerator for Data Center Deep Learning Training Workloads. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020).

[23] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. 2016. Scalable and private media consumption with Popcorn. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 91–107.

[24] Ryan Henry, Femi Olumofin, and Ian Goldberg. 2011. Practical PIR for electronic commerce. In *ACM SIGSAC Conference on Computer and communications security (CCS)*. 677–690.

[25] Jonmichael Hands 2021. Endurance of NVMe, SAS, and SATA SSDs. snia.org/sites/default/files/SSSI/NVMe_SAS_SATA_Endurance_White_Paper.pdf.

[26] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. 2013. Enabling cost-effective data processing with smart SSD. In *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*. IEEE, 1–12.

[27] Roman Kaplan, Leonid Yavits, and Ran Ginosar. 2018. Prins: Processing-in-storage acceleration of machine learning. *IEEE Transactions on Nanotechnology* 17, 5 (2018), 889–896.

[28] Chad D Kersey, Hyesoon Kim, and Sudhakar Yalamanchili. 2017. Lightweight SIMT core designs for intelligent 3D stacked DRAM. In *Proceedings of the International Symposium on Memory Systems*. 49–59.

[29] Aggelos Kiayias, Nikos Leonardos, Helger Lipmaa, Kateryna Pavlyk, and Qiang Tang. 2015. Optimal Rate Private Information Retrieval from Homomorphic Encryption. *Privacy Enhancing Technologies* 2015, 2 (2015), 222–243.

[30] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. 2017. Toward standardized near-data processing with unrestricted data placement for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[31] Shine Kim, Yunho Jin, Gina Sohn, Jonghyun Bae, Tae Jun Ham, and Jae W Lee. 2021. Behemoth: A Flash-centric Training Accelerator for Extreme-scale DNNs. In *USENIX Conference on File and Storage Technologies (FAST)*. 371–385.

[32] Albert Hyukjae Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. 2015. Riffle: An efficient communication system with strong anonymity. (2015).

[33] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 2021. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *IEEE International Solid- State Circuits Conference (ISSCC)*, Vol. 64. 350–352. https://doi.org/10.1109/ISSCC42613.2021.9365862

[34] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H Peter Hofstee, Gi-Joon Nam, Mark R Nutter, and Damir Jamsek. 2017. Extrav: boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1706–1717.

[35] Feifei Li. 2019. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2263–2272.

[36] Jilan Lin, Shuangchen Li, Yufei Ding, and Yuan Xie. 2021. Overcoming the Memory Hierarchy Inefficiencies in Graph Processing Applications. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[37] Helger Lipmaa. 2005. An oblivious transfer protocol with log-squared communication. In *International Conference on Information Security*. Springer, 314–328.

[38] Liu Liu, Jilan Lin, Zheng Qu, Yufei Ding, and Yuan Xie. 2021. ENMC: Extreme Near-Memory Classification via Approximate Screening. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1309–1322.

[39] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. 2014. Efficient Private File Retrieval by Combining ORAM and PIR.. In *NDSS*. Citeseer.

[40] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies* 2016 (2016), 155–174.

[41] Coskun Mermer, Donglok Kim, and Yongmin Kim. 2003. Efficient 2D FFT implementation on mediaprocessors. *Parallel Comput.* 29, 6 (2003), 691–709.

[42] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2019. Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, 253–260.

[43] Vincent Migliore, Maria Mendez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine, and Guy Gogniat. 2016. Hardware/software co-design of an accelerator for FV homomorphic encryption scheme using Karatsuba algorithm. *IEEE Trans. Comput.* 67, 3 (2016), 335–347.

[44] Prateek Mittal, Femi G Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. 2011. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *USENIX Security Symposium (SEC)*.

[45] Peter L Montgomery. 1985. Modular multiplication without trial division. *Mathematics of computation* 44, 170 (1985), 519–521.

[46] Lifeng Nai, Ramyad Hadidi, He Xiao, Hyojong Kim, Jaewoong Sim, and Hyesoon Kim. 2018. CoolPIM: Thermal-aware source throttling for efficient PIM instruction offloading. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 680–689.

[47] Erdinç Öztürk, Yarkın Doröz, Erkay Savaş, and Berk Sunar. 2016. A custom accelerator for homomorphic encryption applications. *IEEE Trans. Comput.* 66, 1 (2016), 3–16.

[48] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent Lee, Gu-Yeon Wei, Hsien-Hsin S Lee, and David Brooks. 2020. Cheetah: Optimizations and methods for PrivacyPreserving inference via homomorphic encryption. *arXiv e-prints* (2020), arXiv–2006.

[49] Dayane Reis, Jonathan Takeshita, Taeho Jung, Michael Niemier, and Xiaobo Sharon Hu. 2020. Computing-in-Memory for Performance and Energy-Efficient Homomorphic Encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 11 (2020), 2300–2313.

[50] Sahand Salamat, Armin Haj Aboutalebi, Behnam Khaleghi, Joo Hwan Lee, Yang Seok Ki, and Tajana Rosing. 2021. NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 262–272.

[51] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 238–252.

[52] SEAL 2020. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[53] Julien P Stern. 1998. A new and efficient all-or-nothing disclosure of secrets protocol. In *International conference on the theory and application of cryptology and information security*. Springer, 357–371.

[54] Yang Su, Bailong Yang, Chen Yang, and Luogeng Tian. 2020. FPGA-based hardware accelerator for leveled Ring-LWE fully homomorphic encryption. *IEEE Access* 8 (2020), 168008–168025.

[55] Muhammad Talha, Mishal Sohail, and Hajar Hajji. 2020. Analysis of research on amazon AWS cloud computing seller data security. *International Journal of Research in Engineering and Innovation* 4, 3 (2020), 131–136.

[56] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. 2018. Mqsim: A framework for enabling realistic studies of modern multi-queue SSD devices. In *USENIX Conference on File and Storage Technologies (FAST)*. 49–66.

[57] Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede. 2020. Heaws: An accelerator for homomorphic encryption on the Amazon AWS FPGA. *IEEE Trans. Comput.* 69, 8 (2020), 1185–1196.

[58] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. 2016. SSD in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*. 1–7.

[59] Jianguo Wang, Dongchul Park, Yannis Papakonstantinou, and Steven Swanson. 2016. Ssd in-storage computing for search engines. *IEEE Trans. Comput.* (2016).

[60] Shiyuan Wang, Divyakant Agrawal, and Amr El Abbadi. 2010. Generalizing PIR for practical private retrieval of public data. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 1–16.

[61] Wei Wang, Xinming Huang, Niall Emmart, and Charles Weems. 2013. VLSI design of a large-number multiplier for fully homomorphic encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 9 (2013), 1879–1887.

[62] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: near data processing for solid

state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 717–729.

[63] Amir Yazdanbakhsh, Choungki Song, Jacob Sacks, Pejman Lotfi-Kamran, Hadi Esmaeilzadeh, and Nam Sung Kim. 2018. In-dram near-data approximate acceleration for GPUs. In *International Conference on Parallel Architectures and Compilation Techniques*. 1–14.