Preserving Access Pattern Privacy in SGX-Assisted Encrypted Search

Shujie Cui, Sana Belguith, Ming Zhang, Muhammad Rizwan Asghar, and Giovanni Russello
Cyber Security Foundry
The University of Auckland
Auckland, New Zealand
Email: {scui379,sbel452}@aucklanduni.ac.nz, {ming.zhang, r.asghar,g.russello}@auckland.ac.nz

***Abstract*—Outsourcing sensitive data and operations to untrusted cloud providers is considered a challenging issue. To perform a search operation, even if both the data and the query are encrypted, attackers still can learn which data locations match the query and what results are returned to the user. This kind of leakage is referred to as data *access pattern*. Indeed, using access pattern leakage, attackers can easily infer the content of the data and the query. Oblivious RAM (ORAM), Fully Homomorphic Encryption (FHE), and secure Multi-Party Computation (MPC) offer a higher level of security but incur high computation and communication overheads.**

One promising practical approach to process the outsourced data efficiently and securely is leveraging trusted hardware like Intel SGX. Recently, several SGX-based solutions have been proposed in the literature. However, those solutions suffer from side channel attacks, high overheads of context switching, or limited SGX memory. In this paper, we present an SGX-assisted scheme for performing search over encrypted data. Our solution protects access pattern against side channel attacks while ensuring search efficiency. It can process large databases without requiring any long-term storage on SGX. We have implemented a prototype of the scheme and evaluated its performance using a dataset of 1 million records. The equality query can be completed in 9.55 milliseconds. Comparing with ORAM-based solutions, such as ObliDB, our scheme is more than $11\times$ faster.

I. Introduction

Cloud computing is a paradigm that offers on-demand storage and computing resources to individuals and enterprises. Due to its low cost and high scalability, cloud computing is growing rapidly. However, data stored and processed in the cloud is becoming an attractive target for adversaries because of the cloud platform's underlying infrastructure and shared model [8].

To protect outsourced data, Searchable Encryption (SE) schemes [9] have been proposed to enable the Cloud Server (CS) to perform encrypted search while ensuring security and privacy of outsourced data. However, due to access pattern leakage, outsourcing sensitive data and search functionality to untrusted cloud providers is still considered a challenging issue. Many recent works, such as [10]–[12], have shown that access pattern leakage can be leveraged to recover the content of the encrypted data and query. Although Oblivious RAM (ORAM) [13], Fully Homomorphic Encryption (FHE) [14], and Secure Multi-Party Computation (MPC) [15] provide a high level of protection to sensitive data, they incur high computation and communication overheads for systems requiring big data storage and intensive processing. There are works [16], [17] that introduce more practical ways to hide access pattern based on multiple cloud service providers.

Another promising practical approach to process outsourced data efficiently and securely is leveraging trusted hardware like Intel Software Guard Extension (SGX) [18]. SGX can isolate sensitive code and data in an encrypted memory region, called *enclave*. During execution, privacy and integrity of enclave memory are preserved with a set of hardware mechanisms. Recently, several SGX-based approaches for encrypted data access have been investigated in the literature. For instance, Fuhry *et al.* [1] present two constructions for sub-linear search over encrypted database by using SGX. Zheng *et al.* [4] introduce an SGX-assisted oblivious data analytics scheme that conceals data access pattern. Moreover, several works [3], [5], [6] explore deployment of ORAM on SGX. Unfortunately, these schemes suffer from several limitations. For instance, the constructions proposed in [1] leak access pattern. Although access pattern is concealed in [3], [5], [6], these schemes require long-term storage on SGX for managing the map between each data instance and its store location, which is in big size when the database has millions of distinct values. Moreover, in the scheme proposed by Zheng *et al.* [4], the entire database needs to be linearly scanned to answer a query. All these solutions are not very practical due to very limited memory resources in SGX. Another limitation is that there are side channel attacks when using SGX to ensure secure data access. Indeed, several recent works, such as [19]–[21], have shown that the Operating System (OS) can infer data access pattern in SGX by launching side channel attacks, such as page faults and timing attacks. State of the art solutions [3]–[6] do not give the solutions to resist the side channel attacks. In Table I, we compare existing SGX-based schemes.

Research Challenges. The main objective of this paper is to preserve access pattern privacy by using SGX while ensuring efficient search over encrypted data and without any long-term storage requirement in the enclave. As discussed below, there are two main challenges to achieve our objective: preserving access pattern privacy while providing efficient search and withstanding side channel attacks.

To support efficient search, a straightforward method is to build indexes and load only the required indexes into SGX.

TABLE I
COMPARISON OF RECENT SGX-BASED SCHEMES.

Scheme	Search complexity	Access pattern leakage	Side channel leakage	No long-term storage on SGX
Fuhry *et al.* [1] – Construction 1	$O(\log N)$	◐	○	✓
Fuhry *et al.* [1] – Construction 2	$O(\log N)$	○	○	✓
Gribov *et al.* [2]	$O(\log N)$	○	○	✓
Eskandarian *et al.* [3] – Linear	$O(N)$	●	○	✓
Eskandarian *et al.* [3] – Indexed	$O(\log^2 N)$	◐	○	✓
Zheng *et al.* [4]	$O(N)$	●	○	✓
Sasy *et al.* [5]	–	●	●	×
Costa *et al.* [6]	–	●	●	×
Ahmad *et al.* [7]	–	●	●	×
Our work	$O(N/P)$	●	●	✓

○, ◐, and ● mean the information is completely leaked, partially leaked, and not leaked, respectively. ✓ and × represent if long-term storage on SGX is not required or required, respectively. N represents the number of nodes in the tree or number of records in the database. P is page size in an SGX enclave. In [5], [6], no search operation is performed and this is denoted by –.

However, this method leaks the index access pattern directly to the CS. To fully hide the index access pattern, a typical strategy is to load all the indexes into SGX. Nonetheless, an SGX enclave only has around 90 MB memory for storing the code and data. For a large database, indexes will exhaust the enclave memory and virtual memory mechanism of the OS. Consequently, this will significantly affect the performance of SGX. Moreover, as shown in [1], which pages are accessed is still unprotected because of the page fault attack. Therefore, the first challenge is to fully guarantee the data access pattern privacy without exhausting the enclave memory when the database is large.

Although SGX provides a trusted environment for data processing, it suffers from side channel leakage. Actually, the data is leaked when it is loaded to SGX [19]–[21]. Thus, the second challenge is to protect data access pattern against potential side channel attacks. Several countermeasures, such as data oblivious access, balanced code execution, and data shuffling have been proposed in [22]–[24]. However, these techniques are too generic to be used for SE schemes. For instance, to defend against the page fault attack, Shinde *et al.* [23] propose to balance the code execution by adding and accessing dummy data. Nonetheless, in a B+ tree, we should ensure the added dummy keys can be checked like real ones and do not affect correctness of search results; otherwise, the CS can infer the real index access pattern. Therefore, the techniques specific to SE schemes should be considered.

Our Contributions. In this paper, we present an SGX-assisted SE scheme addressing aforementioned research challenges. Basically, our solution uses the B+ tree structure to ensure search efficiency. To address the first challenge, our scheme loads and processes the tree indexes in batches. On the one hand, this method ensures that the index access pattern is protected. On the other hand, our scheme can also process a large database without exhausting the enclave memory. Moreover, it is also necessary to defend against the page fault attack. To mitigate other side channel attacks, the B+ tree is searched in a balanced way, independent of the query and access pattern. To analyse the performance, we evaluate our proposed scheme and compare it with ObliDB [3] on Big Data benchmark [25]. Our scheme outperforms ObliDB by at least $11\times$. We also compare our scheme with a baseline implementation without access pattern protection with sub linear search support. Our results show that our techniques to defend side channel attacks add less than $157\times$ overhead when the B+ tree contains 1 million keys. In addition, our approach does not require any long-term storage on the enclave. In summary, our contributions in this paper are as follows:

- Our scheme protects data access pattern from the CS by leveraging a trusted SGX.
- Our scheme prevents the CS from inferring access pattern by launching side channel attacks.
- In our scheme, a search operation can be performed efficiently.
- We implement a prototype of the system and test its performance on an SGX-based hardware.

Paper Organisation. The remainder of this paper is organised as follows. First, Section II reviews related work. Section III gives a brief overview of SGX functionalities and explains possible side channel attacks. An overview of our proposed solution is provided in Section IV. Then, we present our solution details in Section V before introducing its security analysis in Section VI. In Section VII, we report performance analysis. Finally, we conclude this paper in Section VIII.

II. RELATED WORK

Fuhry *et al.* [1] present two SGX-assisted constructions for search over encrypted data. To support sub-linear search, the B+ tree index is utilised in both constructions. In the first construction, the encrypted index tree is entirely loaded into the enclave for performing search, which is not scalable for large databases. Moreover, the untrusted server could observe data access inside the enclave with a page-level granularity by leveraging the page fault side channel attack [19]. In their second construction, only the nodes involved in the tree traversal are loaded into the enclave. In this case, the accessed nodes are leaked to the CS directly.

In [2], Gribov *et al.* also present a B+ tree-based SGX-assisted encrypted database, fully supporting SQL queries, called *StealthDB*. They also reduce the context switching overhead between the enclave and the untrusted server memory by $5\times - 10\times$ by using an exit-less communication mechanism [26]. Unfortunately, StealthDB still leaks the index access pattern since it only loads the matched nodes into the enclave for performing search.

Eskandarian and Zaharia [3] also proposed two basic methods for the data storage and access, named linear and indexed storage. For linear storage, SGX searches each record one by one, and then loads the matched records with ORAM primitives. This method can conceal access pattern effectively, but it incurs significantly high computation overheads. In the indexed storage, a B+ tree is searched first to narrow down the records to be scanned. However, access pattern over the B+ tree is leaked.

Zheng *et al.* [4] study how to leverage SGX to secure distributed analytical workloads, and propose a system called *Opaque*. By sorting and shuffling the data, Opaque could avoid the access pattern leakage. However, Opaque linearly scans and sorts the entire database to answer a query, which is inefficient for large databases. Moreover, they do not give specific solutions to address side channel attacks.

There are works [5]–[7] that present three different designs and implement the ORAM primitives running on SGX that can protect access pattern and defend side channel attacks. However, in those designs, the enclave has to store a map between each distinct value and its store location. Due to limited memory of SGX enclave, those implementations are not scalable to the databases with a large number of distinct values.

As summarised in Table I, none of above schemes can achieve our objective, *i.e.,* preserving access pattern privacy by using SGX while ensuring efficient search over encrypted data without any long-term storage on SGX.

III. Background

A. Intel SGX

In this section, we give a brief introduction of SGX functionalities relevant to our system. For more details on SGX, we refer the reader to [18], [27]. SGX is an extension of x86 instructions for creating and managing software components, called *enclave*. Physically, the enclave is located inside a hardware guarded area of memory called Enclave Page Cache (EPC). The EPC consists of 4KB page chunks, and only around 90MB EPC can be used by the application. The SGX hardware enforces additional protection on the enclave, such that it is isolated from the code running on the system including the OS and the hypervisor.

Apart from the isolated code and execution, SGX has another two main security properties: sealing and attestation. Sealing is the process of encrypting enclave secrets for persistent storage to disk [28]. Every SGX processor has a key called the Root Seal Key that is embedded during the manufacturing process. Once an enclave is created, a seal key – which is specific to the enclave – is derived from the root seal key. When the enclave is torn down, this seal key is used to encrypt data, and store the data in disk. SGX also supports remote attestation, enabling a remote party to verify if an enclave is created properly on a trusted SGX. It also provides integrity to the code and data loaded into the enclave. Furthermore, the remote attestation feature helps in establishing a secure channel between an external party and the enclave.

B. Side Channel Attacks on SGX

Intel SGX offers secure execution environment by cryptographically protecting code and data on an untrusted server. Unfortunately, it is vulnerable to side channel attacks. As discussed below, there are at least two possible side channel attacks that could be launched by the untrusted server to derive sensitive information.

Page Fault Attack. An SGX program is executed in user mode, and it needs the underlying OS to manage virtual memory pages. Specifically, when launching an SGX process, the OS creates the page tables that map the virtual addresses to physical memory entries. However, when the virtual pages can not be mapped to the physical memory, the CPU incurs a page fault and the faulting address will be reported to the OS. By manipulating the page table mappings, as shown in [19], a malicious OS can observe the page access pattern in SGX.

Timing Attack. Timing attack allows attackers to learn sensitive information by analysing the time taken to execute data-dependent operations. Every logical operation in a computer takes time to execute, and this time could differ based on the input, which is also the case for the operations in SGX. With precise measurements of the execution time for each operation, an attacker can backtrack the input data.

IV. Solution Overview

In this section, we first define the threat model. Then, we describe the leakage and give a security definition of our system. Finally, we briefly explain our proposed approach.

A. Threat Model and Assumptions

Our system includes three entities: the user, SGX-enabled Cloud Server (CS) and an SGX enclave within the CS. The user uploads encrypted databases to the CS and later sends encrypted queries. We assume the user is trusted.

The CS is responsible for storing encrypted data and loading data into the enclave for performing search. Similar to existing work (see Section II), we assume adversaries could attack and fully control the OS running on the CS, and they are curious about the data residing on the CS. For simplicity, in the rest of the paper, we regard the CS as an adversary, which honestly follows the specified protocol but is curious to know the data. Since we employ encryption, the CS is unable to access the data directly. Moreover, the CS can not control and access the code and data within the enclave. Nonetheless, it can interrupt the enclave as desired, by modifying the OS and SGX SDK, in order to get side channel information. Therefore, we assume the CS is able to exploit side channel attacks including the page fault attack [19] and timing attack [29] to infer the code

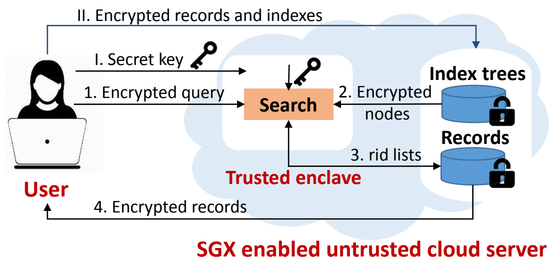

Fig. 1. Proposed approach: The user shares the secret key with the trusted SGX enclave (Step I). The encrypted data and indexes are stored in the untrusted cloud server (Step II). The query is decrypted and processed by the enclave (Steps 1, 2, and 3). After searching, the matched records together with a set of random records will be returned to the user (Step 4).

paths and data access patterns inside the enclave. There are also other types of side channel attacks, such as cache attack [30], [31], power monitoring [32], electromagnetic [33], and acoustic [34], which are out of the scope of this work.

SGX enclave is responsible for processing user queries and returning search results to users without leaking any content and access pattern to the CS. SGX enclave is also considered to be trusted. In particular, both integrity and confidentiality of the code and data inside the enclave are protected with inherent cryptographic mechanisms. We also assume that SGX provides methods for establishing a secure channel with the users for protecting the communication between them.

B. Leakage and Security Definition

In this work, we do not make any effort to hide the tree structure and database size. Basically, for each B+ tree (there is a B+ tree for every field in the database), the number of its levels, the number of nodes at each level, and the size of each encrypted node are leaked to the CS. In this section, we define the leakage as index size, *i.e.,* $Size(\mathbf{I})$. Moreover, from the search history, the CS knows the number of queries has been issued, which tree is searched for each query, and the structure of each encrypted query. This leakage from queries is defined as $\mathbf{L}(Q)$. For the dataset $\mathbf{D}$, the number of records and the size of each encrypted record are also leaked, which are defined as $Size(\mathbf{D})$. Based on the leakage, the security of access pattern is defined as follows:

Definition 1 (Access Pattern Security): Let $\overrightarrow{H} := ((Q_1, R_1), \ldots, (Q_T, R_T))$ be the search history at time T, where Q_t denotes the query, and R_t is its search result at time t $(1 \leq t \leq T)$. Let $A(\overrightarrow{H})$ be access history of the data stored on the CS, where search history is $\overrightarrow{H}$. We say that access pattern is protected from the CS if for any two search histories $\overrightarrow{H}_0$ and $\overrightarrow{H}_1$ of the same length and leakage (as defined above) and their access pattern $A(\overrightarrow{H}_0)$ and $A(\overrightarrow{H}_1)$, respectively, are computationally indistinguishable by the CS.

C. Architecture Overview

The overview of our proposed architecture is illustrated in Fig. 1. It consists of a trusted code base inside the enclave for processing queries, the data storage on the CS, and the encryption and decryption operations on the user side.

Initially, the user generates a secret key sk to encrypt the outsourced data and queries. The secret key sk is shared with the enclave via a secure channel (Step I). To ensure search efficiency, the user first builds B+ tree indexes for the dataset and then uploads both the encrypted dataset and index trees to the CS (Step II). When issuing a query, the user encrypts it using randomised encryption and sk (Step 1). Since sk is unknown to the CS, the content of the query and whether the same query has been sent before (*i.e., search pattern*) are protected from the CS. With the secret key sk, the enclave decrypts the query and loads the associated index tree from the CS for performing search (Step 2). To hide the tree access pattern, the nodes are loaded and accessed in an oblivious manner that is independent of the query and could resist against side channel attacks including page fault and timing as explained in Section V. Every key stored in the leaf node is attached with a pointer pointing to a list of identifiers (rids) of the records matching the key. To hide the record access pattern, the enclave sends the matched rids to the CS also in an oblivious manner that could resist against side channel attacks (Step 3). With the matched rids, the CS sends the matched records to the user (Step 4). Finally, the user obtains the result in plaintext by decrypting them using sk.

In this work, we only give the details of processing basic equality queries. More complicated SQL queries, such as range query, GROUP BY, JOIN, COUNT, and SUM will be our future work.

V. Solution Detail

In this section, we first introduce our solution used to protect access pattern. Then, we show how the data is represented and encrypted. Finally, we describe how the queries are processed in our system.

A. Access Pattern Protection

In this work, the main objective is to hide access pattern over the B+ tree from the CS. Protecting the index access pattern means protecting which node(s) at each level is(are) accessed from the CS, excluding the root node.

In our approach, to fully hide the index access pattern without exhausting the enclave memory, SGX reserves only a single EPC page and loads the index tree in batches, rather than loading the entire tree in one go. Specifically, SGX loads and processes the tree nodes level by level. When the nodes at one level can not be loaded in one page, they will be grouped in fragments and loaded fragment by fragment, where each fragment contains the maximum number of nodes that can be loaded in one page. Thus, our system can efficiently process the dataset with any size. Moreover, which node is accessed inside each loaded page is unknown to the CS.

Note that SGX could also reserve more pages and load more nodes per batch. In this work, we focus on the case of loading one page each time.

1) Solutions against Page Fault Attacks: Using a single page to hold data blocks is not sufficient to resist the page fault attack. Indeed, when the CS launches page fault attacks, it can still learn if the page is accessed or not since only when the page is accessed the fault address will be reported to the OS. Our solution is to ensure every loaded page is accessed by SGX. In particular, when the matched nodes are not included in the EPC page, some random nodes will be accessed. In this case, even if the page fault exception occurs, the CS is unable to know if it is caused by accessing the matched nodes or random ones.

2) Solutions against Timing Attacks: To resist timing attacks, the time of searching the B+ tree should be independent of the query and access pattern.

In the traditional B+ tree, only one node at each level should be searched for equality queries. To hide access pattern at a page level, we just need to ensure a random node is accessed when the matched node is not contained in the EPC page. However, there are still two issues making the index access pattern vulnerable to timing attacks. First, in a traditional B+ tree, the nodes may contain a different number of keys, and the time of processing a fragment depends on how many keys are checked in the accessed node. In turn, based on the processing time, the CS can infer the number of checked keys, and then infer which node is accessed. For instance, assume checking one key takes T time. If the loaded fragment is processed in nT time, the searched node must contain no less than n keys. According to the size of each encrypted node, the CS could infer which node is accessed with a high probability. This issue can be solved by hiding the real size of each node. However, in a traditional B+ tree, the number of keys checked in the accessed node also varies with the query. That is, different queries are processed at different times, and the equivalent queries must be processed at the same time. On the contrary, when all the pages are processed at the same time, there is a high probability that the two queries are equivalent and their tree access patterns are the same. To solve the two issues, each fragment should be processed in constant time for all equality queries. To this end, we first propose to ensure all the nodes in the tree contains $b-1$ keys by adding dummy keys, where b is the branching factor for building the tree. Second, we propose to check all the keys in the accessed node, rather than to stop the search once the matching key is found. As a consequence, each fragment is processed in $(b-1)T$ time no matter which node is accessed. The details of generating dummy keys and checking the node are given in the Sections V-B and V-D, respectively.

B. Data Representation

To support sub linear search, given the database, the user first builds a standard B+ tree based on a defined branching factor. Formally, we define a B+ tree as $tree = \{b, L, N, \textbf{cnt}, \textbf{nodes}\}$. Here, b is the branching factor, indicating each node can have up to b children nodes and $b-1$ keys. L is the number of levels of the tree, and the root node is in level $l = 1$. $\textbf{cnt} = \{cnt_1, \ldots, cnt_L\}$ is an array of integers of length L. cnt_l represents the number of nodes in level l, where $l \in [1, L]$. The total number of nodes in the tree is $N = \Sigma_{l=1}^{l=L} cnt_l$. $\textbf{nodes} = (node_0, \ldots, node_{N-1})$ is the array

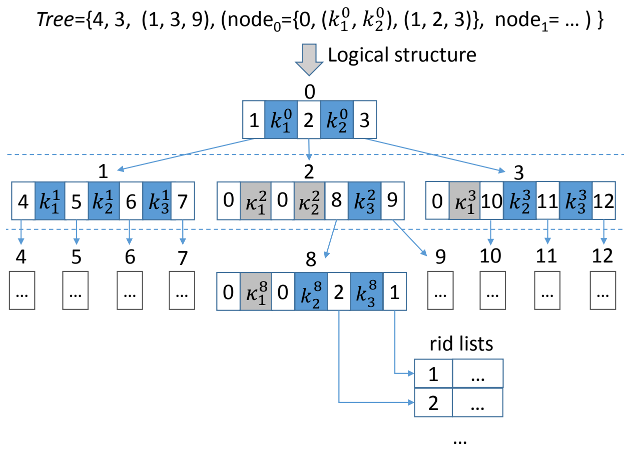
Tree={4, 3, (1, 3, 9), (node0={0, (k_1^0, k_2^0), (1, 2, 3)}, node1= ...) }
Logical structure
0
1 k_1^0 2 k_2^0 3
1
4 k_1^1 5 k_2^1 6 k_3^1 7
2
0 κ_1^2 0 κ_2^2 8 k_3^2 9
3
0 κ_1^3 10 k_2^3 11 k_3^3 12
4
5
6
7
8
0 κ_1^8 0 k_2^8 2 k_3^8 1
9
10
11
12
rid lists
1 ...
2 ...
...

Fig. 2. A B+ tree index example with $branch = 4$, 3 levels, and 13 nodes: From root to leave nodes and from left to right, all the nodes are assigned a sequence of numbers in order as the id. Each non-root node stores 3 keys and 4 values, including both real and dummy keys and values. The dummy keys and values must be stored in the most left. The real value is the id of its children node. The dummy values are set to 0. The real value in each leaf-node is the id of a list of identifiers of the records matching the key.

storing the N nodes in the tree. The order of storing nodes is from root to leave nodes and from left to right.

The non-leaf node is defined as $node_{id} = \{id, (k_1, \ldots, k_{b-1}), (cid_1, \ldots, cid_b))\}$. id is unique and used to identify the node, which also represents the node store location in **nodes**. In the B+ tree, each node contains an array of keys $(k_1, \ldots, k_{b-1})$ for searching, and an array of its child nodes ids $(cid_1, \ldots, cid_b)$ for reaching its child nodes.

It is known that the root node is always the first node to be searched whatever the query is. To defend against the timing attack, we just need to search all its keys for all queries. Thus, we do not need to pad the root node with dummy keys. However, as already mentioned, we should ensure all the non-root nodes has $b-1$ keys by adding dummy keys.

The $b-1$ keys in each non-root node consist of both real and dummy keys. Specifically, assume the node contains γ real keys. To ensure the node can be processed in a balanced way (the details are given in Section V-D), the real keys are stored in the most right of the node and in increasing order, *i.e.,* $(k_1, \ldots, k_{b-1-\gamma})$ and $(k_{b-\gamma}, \ldots, k_{b-1})$ are dummy and real keys respectively, and $k_1 < \ldots < k_{b-1}$. The γ real keys separate the key domain into $\gamma + 1$ subtrees which are reachable by $\gamma+1$ children node ids, *i.e.,* $(cid_{b-\gamma}, \ldots, cid_b)$. Moreover, the real keys in $node_{cid_i}$ are in $(k_{i-1}, k_i]$. In other words, there is a sub domain for each node. To ensure the dummy keys never match the query, we assign them the values out of the node's sub domain. Specifically, we set $node_{cid_i}.dummy\ keys < k_{i-1}$. In addition, the dummy keys are not used to generate subtrees, so $(cid_1, \ldots, cid_{b-\gamma-1})$ are also dummy and set to 0.

The leaf node has the same structure as the non-leaf node. However, the leaf node does not have children nodes; instead, its cid_i points to a list of $rids$ of the records whose values are equal to k_i. Likewise, if the leaf node has less than $b-1$ real keys, a number of dummy keys will be generated and stored in the most left of the key array. The record identifiers lists are encrypted and stored separately. Moreover, to hide size information, the user pads the lists to the same size before encrypting.

Fig. 2 illustrates the logical structure of a sample tree with $b = 4$, $L = 4$, and $N = 13$, where $cnt_1 = 1, cnt_2 = 3$, and $cnt_3 = 9$. The node with less than 3 keys is padded with dummy keys that are inserted to the most left of the nodes. For instance, the first two keys κ_1^2 and κ_2^2 in $node_2$ and the first key κ_1^3 in $node_3$ are the dummy keys. Moreover, the sub domain for $node_2$ is $(k_1^0, k_2^0]$, so we should set $\kappa_2^2, \kappa_1^2 < k_1^0$. The sub domain for $node_3$ is $(k_2^0, MAX]$, so $\kappa_1^3 < k_2^0$, where MAX represents the maximum value for keys.

C. Data Encryption

Algorithm 1 Encryption$(tree, P)$

```
1: Pads each node into s-bit if required
2: for $l = 1$ to $l = tree.L$ do
3:     Segment the nodes at level $l$ into fragments $\mathbf{F}^l = \{\mathbf{f}_1^l, \ldots, \mathbf{f}_{n_l}^l\}$, where $n_l = \lceil \frac{tree.cnt_l}{P} \rceil$
4:     $\pi_{sk\|l}(\mathbf{F}^l)$
5:     for $i = 1$ to $i = n_l$ do
6:         $Enc_{sk}(\mathbf{f}_i^l)$
7: for each $cid$ in each leaf node do
8:     $Enc_{sk}(list_{cid})$, pad the list first if required
```

After building the tree, the user encrypts both the dataset and the index tree before sending them to the CS. For encryption, given a security parameter λ, the user generates the secret key sk. Basically, each record is encrypted with sk using a randomised encryption algorithm such as AES-GCM. Each encrypted record is identified with a unique rid in plaintext.

In this work, we do not aim to hide the tree structure from the CS. That is, the CS could learn the values of b, L, N and **cnt**. The user only encrypts the nodes in **nodes**. The detail of the tree encryption is shown in Algorithm 1.

Specifically, the tree encryption is performed in three phases. In the first phase, all the nodes are padded to the same size. In addition to adding dummy keys, the elements inside each node, including id, cid and key, should also be padded to the same size. Assume all the nodes are s bits. It is known that the enclave page size is 4 KB, meaning each page can hold at most $P = \lfloor \frac{4KB}{s} \rfloor$ nodes.

In phase 2, the user fragments the nodes at each level P by P (Line 3), and then permutes the fragments with a pseudo-random permutation $\pi : \{0,1\}^{\lambda'} \times \{0,1\}^{n_L} \rightarrow \{0,1\}^{n_L}$ (Line 4) in order to hide the order among the fragments, where n_L represents the number of fragments at the last level and $\lambda' = \lambda + \log_2 n_L$. Each fragment has P nodes excluding the last fragment, such the nodes in each level will be loaded fragment by fragment. Afterwards, each fragment is encrypted with a semantically secure block encryption $Enc : \{0,1\}^{\lambda} \times \{0,1\}^{32768} \rightarrow \{0,1\}^{32768}$ ($32768 = 4KB$ is the data size in bit that can be hold in one EPC page), *e.g.,* AES-128 in GCM mode (Line 6).

In the last phase, for each cid in each leaf node, the user pads and encrypts its rid list with Enc and sk (Line 8). By padding the lists into the same size, the data distribution can be hidden from the CS.

D. Searching Tree

Algorithm 2 EqualityQuery$(EQ, tree, P)$

```
1: $Q \leftarrow Dec_{sk}(EQ)$
2: $mid \leftarrow 0$
3: $Nodes[P] \leftarrow 0$ {Allocate an EPC for loading tree nodes}
4: for $l = 1$ to $l = tree.L$ do
5:     for $i = 1$ to $i = \lceil \frac{tree.cnt_l}{P} \rceil$ do
6:         Load $Enc_{sk}(\mathbf{f}_i^l)$ to $Nodes$
7:         SearchPage$(Q, Nodes, n, l, i, tree.b, tree.L, mid)$, where $n$ is the number of nodes in this fragment
```

When issuing a query Q, the user encrypts it by computing $EQ \leftarrow Enc_{sk}(Q)$, such the query is protected from the CS. Moreover, Enc is semantically secure and search pattern is also hidden from the CS.

The search operation is performed on SGX, and the details are shown in Algorithm 2. Once received EQ, SGX first decrypts EQ with sk (Line 1), such it can learn the type of the query, the interested field(s) and keyword(s).

In Algorithm 2, mid is used to cache the id(s) of the node(s) should be searched at the next level, or the identifier(s) of the matched rids list(s) after searching each fragment. When Q is an equality query, mid is set to be an integer of the same length as node id, and initialised as id of the root node.

SGX reserves a fixed EPC page $Nodes$ (Line 3) and loads the required index tree(s) in batches for performing search (Line 4 - Line 7). Indeed, SGX loads and processes the permuted fragments at each level one by one. Since all the nodes at each level will be loaded, it is unnecessary to recover the order of the fragments before loading. In the following, we show how the loaded fragment is processed in detail for equality queries.

Algorithm 3 EqualitySearch$(Q, Nodes, len, l, i, b, L, mid)$

```
1: $Nodes \leftarrow Decrypt_{sk}(Nodes)$
2: $r_1 \xleftarrow{\$} \{0, len-1\}$
3: $r_2 \leftarrow mid - Nodes[0].id$
4: if $Nodes[0].id \le mid \le Nodes[len-1].id$ then
5:     $node \leftarrow Nodes[r_2]$
6:     $flag \leftarrow 1$
7: else
8:     $node \leftarrow Nodes[r_1]$
9:     $flag \leftarrow 0$
10: $tid \leftarrow node.cid_b$
11: for $j = b-1$ to $j = 1$ do
12:     if $(Q.K \le node.k_j$ and $l < L)$ or $(Q.K = node.k_j$ and $l = L)$ then
13:         $shift \leftarrow 1$
14:     else
15:         $shift \leftarrow 0$
16:     $tid \leftarrow tid - shift$
17: $mid \leftarrow mid * (1 - flag) + flag * tid$
```

For an equality query, only one node will be accessed in each loaded fragment, *i.e.,* the matched one or a random one. The challenge is how to ensure that the CS is unable to learn whether the accessed node is the matched one or a random one via side channels.

The detail of processing each loaded fragment is described in Algorithm 3, *EqualitySearch*. After searching the nodes at the upper level, the id of the matched node (*i.e.,* mid) is cached in SGX. SGX first decrypts the loaded fragment (Line 1). Second, it checks if the matched node $node_{mid}$ is contained in the current fragment (Line 4). To hide if the accessed node is the matched one or a random one, the two cases are processed in a balanced way (Line 2 - Line 9). Specifically, SGX pre-computes a random location r_1 and the possible location of the matched node r_2. If the matched node is included in this fragment, the r_2-th node in the page will be accessed. Otherwise, the r_1-th node will be accessed. Moreover, a $flag$ is used to mark if the searched node is the matched one or a random one. Specifically, $flag = 1$ when the searched node is the matched one, and $flag = 0$ otherwise.

Once the node to be searched is determined, the next step is to check which key inside the node matches the query. The enclave traverses all the keys inside the node (Line 10 - Line 16). To resist timing attacks, the node is also processed in a balanced way. That is, all the keys in the node are checked in the same way no matter whether it is a dummy or real and whether it matches the query or not. Specifically, $Q.K$ is first assumed to be greater than the last key $node.key_{b-1}$ (Line 10). In other words, the last child node is assumed to be the matched one at the next level. From the most right to the left, SGX checks if each key $node.k_j$ is greater than or equal to the query. If yes, tid decrements, meaning the child node cid_{j+1} is not the matched one. Otherwise, tid is unchanged, indicating cid_{j+1} is the matched child node.

Nevertheless, the dummy keys in the searched node could also match the query since they are assigned with real values. To ensure the correctness of the search result, we should ensure the dummy keys can not change the value of mid. As shown in Line 17, after searching a node, mid is updated based on two values: $flag$ and tid. When the searched node is randomly picked, $flag = 0$ and mid is not changed after searching no matter what tid is. On the contrary, when $flag = 1$, *i.e.,* the searched node is the matched one, mid will be updated with tid. We should ensure the dummy keys can not change the value of tid, *i.e.,* $shift = 0$ and $Q.K$ is greater than all the dummy keys stored in matched nodes. Indeed, $Q.K$ should be greater than or equal to the first real key in the matched node. Recall that the dummy keys are stored in the most left of the node. By assigning the values less than the first real key, they will never change tid since they are less than $Q.K$.

Note that, if the searched node is the matched leaf node, the result mid is the identifier of the matched rids list.

E. Returning Search Results

After searching the tree index, SGX gets the identifier(s) of the matched rids list. The next step is to return the matched records to the user. The straightforward way is to send mid to the CS. However, the CS could learn the record access pattern, and then infer the search pattern and tree access pattern.

In our system, the matched records are returned in an oblivious manner. Specifically, SGX first loads the matched list(s) together with a set of random lists, and which of them are the matched one (s) is unknown to the CS. Second, to resist side channel attacks, SGX decrypts all the lists to get the rids of the matched records. Third, the matched rids and a set of random rids are sent to the CS, and the CS sends the records identified with these rids to the user. Fourth, SGX re-encrypts the matched rids with sk to make it different from the one stored in the CS, and sends it to the user, using which the user can identify which records are the matched ones. Finally, the user decrypts the matched records and get the data in plaintext.

VI. Security Analysis

In this section, we prove that our scheme achieves the data access pattern security against the CS.

Thereom 1: The proposed system protects access pattern from the CS.

Proof. Let $\overrightarrow{H}$ be the query history of size T.

Tree Access. In our system, for any query, SGX loads the whole tree into the enclave in batches. That is, the index access pattern for each query is always the whole tree from the CS perspective. Moreover, which node is accessed in the EPC page is invisible to the CS. Access pattern at the page granularity is protected from the CS, since the page is searched after each loading. In particular, when there is a page fault, the CS OS can always get the fault report, no matter which node inside the page is searched. However, the CS has no idea if the accessed node is the matched or random one. By padding all the non-root nodes to the same size and checking the keys in a balanced way, each page of the nodes can be processed in a constant time for equality queries and such the index access pattern is protected against the timing attack.

Records Access. While returning records to users, the CS sees $A(\overrightarrow{H})$, which is a sequence $(R'_1, \ldots, R'_T)$, where R'_t consists of the matched records and a set of random records. All the records stored in the CS are encrypted using randomised encryption. Therefore, the records in each R'_t are indistinguishable from a set of random bit strings by the definition of randomised encryption. Note that although R'_t is revealed to the CS, it can not differentiate matched records from the random ones, and it can infer the precise access pattern with $\frac{1}{2}^{|R'_t|}$ probability.

Another issue is if the CS can infer search pattern from $A(\overrightarrow{H})$ when $Q_i = Q_j (1 \leq i, j \leq T)$, $R'_i \cap R'_j \neq \emptyset$. However, when $Q_i \neq Q_j (1 \leq i, j \leq T)$, it is also possible that $R'_i \cap R'_j \neq \emptyset$, where the intersection are unmatched records. Thus, only when $R'_i \cap R'_j = \emptyset$, the CS can learn $Q_i \neq Q_j$. On the contrary, it can not determine if they are equal when $R'_i \cap R'_j \neq \emptyset$.

VII. Performance analysis

In this section, we first analyse the B+ tree search complexity of our scheme. Then, we demonstrate the performance of our scheme by using the TPC-H [35] and Big data benchmarks [25].

TABLE II
COMPARISON WITH OBLIDB AND THE BASELINE.

Tables	**Keys in B+ Tree**	**Branching Factor**	**#Pages per Batch**	**Query**	**Result Size**	**Our Work**	**ObliDB Indexed**	**Speedup**	**Baseline**	**Overhead**
ORDERS	1,048,575	32	16	Q_0	1	9.55ms	–	–	0.061ms	156.6×
CFPB	1023	4	1	Q_1	1	0.352ms	3.9ms	11×	0.029ms	12.1×
CFPB	1023	4	1	Q_2	177	0.352ms	695ms	1974.4×	0.029ms	12.1×

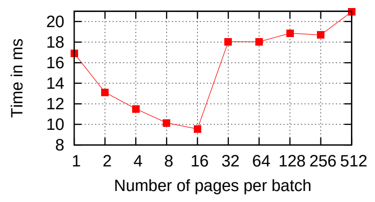

Fig. 3. Tree search time with 1 million keys.

A. Complexity Analysis

For a traditional B+ tree, the search complexity is $O(\log_b N)$, where N represents the number of nodes and b is a branching factor of the tree. However, in our system, in order to resist side channel attacks, the B+ tree is accessed in a balanced way, where multiple nodes at each level should be searched for each query. Specifically, within every P nodes, at least one of them should be accessed. Thus, the search complexity is $O(\frac{N}{P})$ in our system.

B. Implementation

We implemented and evaluated the performance of our system on an Intel NUC 7i3BNH, with a 4-core Intel i3-7100U 2.4GHZ processor with SGX enabled and 8GB RAM. The prototype of the proposed system is implemented in C/C++. The cryptography primitives used on the user side, such as the records and indexes encryption, are implemented based on Libgcrypt 1.8.2 library [36]. The trusted code on SGX is implemented based on SGX SDK 2.0. Specifically, we use 'sgx_rijndael128GCM_decrypt' to recover the plaintext of loaded nodes in SGX. The performance is tested on real SGX hardware. All the times presented in the following are averaged over 100 runs.

C. TPC-H Benchmarking

We first evaluated the performance of our scheme with TPC-H [35] dataset. The table used in our experiment is the 'ORDER' table, which consists of 1.5 million records and 9 fields. We built a B+ tree index for 'O_ORDERKEY' field, which consists of 1 million keys (when building the tree $48,575$ dummy keys are added).

We first evaluated the B+ tree search time with different batch sizes, *i.e.,* the number of fragments loaded per batch, which is also the number of EPC pages reserved in the enclave. When building the B+ tree, the branching factor was fixed to 32. We tested the B+ tree searching time for equality queries by changing the batch size from 1 to 512. The result is shown in Fig. 3.

From Fig. 3, we see the searching time goes down when loading more pages each time, and reaches the lowest point when loading 16 pages per batch, which is about 9.55 milliseconds (ms). However, when loading more than 16 pages, the searching time goes up again. That is because there is less context switching between the untrusted and trusted code when loading more nodes in each batch. However, when loading more than 16 pages, the enclave memory is exhausted, and part of the data has to be swapped in and out between the enclave and the untrusted memory.

Moreover, we also implemented the second construction (*i.e.,* Construction 2) of HardIDX [1] as a baseline, where only the matched nodes in the B+ tree are loaded into SGX for searching. Comparing with the baseline, our scheme takes many strategies to protect access pattern from the CS. In Table II, we test a simple query $\mathbf{Q}_0$: 'select * from ORDERS where O_ORDERKEY$=1506$' and compare the search time of our system with the baseline case, and show the overhead added by these strategies. When loading 16 pages per batch, our scheme incurs 156.6× performance overheads than the baseline.

D. Big Data Benchmarking

ObliDB [3] is an ORAM-based solution that can also protect access pattern. In [3], Eskandarian *et al.* have shown that their indexed solution is much more efficient than Opaque [4]. In the following, we will show our scheme is much more efficient than ObliDB indexed solution.

For the comparison, as done in ObliDB, we evaluated our scheme with Big Data Benchmark [25]. Specifically, we tested two queries, $\mathbf{Q}_1$: 'select * from CFPB where Date_recieved $=$ 20130817' and $\mathbf{Q}_2$: select * from CFPB where Date_recieved $=$ 20130514', with different result sizes on 'CFPB' table. 'CFPB' contains 107000 rows, and we built a B+ tree with 1023 keys (including 543 real keys and 460 dummy keys) for the searched field.

We downloaded the code of ObliDB from [37] and tested its performance on our own machine. Moreover, we also tested the baseline implementation, where only the matched nodes are loaded into SGX for search, with the same queries and datasets. For testing our scheme, the branching factor was fixed to 4, only 1 page was loaded per batch. Moreover, once the matched $rids$ list is found, we loaded 2× random matched records to hide the real access pattern.

The test results for our scheme, ObliDB and the baseline are illustrated in Table II. In our scheme, to resist timing attacks, the same type of queries are processed in constant time. From Table II, we can see that, both Q_1 and Q_2 were executed in 0.352 ms, which includes the B+ tree search time and the *rids* list processing time. In contrast, the ObliDB indexed solution is implemented based on ORAM primitives, which is affected significantly by the result size. For the point queries over 'CFPB' dataset, when result sizes are 1 and 177, our scheme is $11\times$ and $1974.4\times$ faster than ObliDB (indexed solution), respectively. Comparing with the baseline, our solution is $12.1\times$ slower.

VIII. Conclusions and Future Work

In this paper, we introduce an approach that supports search over encrypted data and preserves privacy of access pattern using SGX. B+ tree indexes are built in order to support sub linear search. By loading the tree nodes page by page and accessing the nodes in a balanced manner, access pattern is also protected against page fault and timing channel attacks. Moreover, our solution can process large databases efficiently without requiring long-term storage on SGX. As for future work, we aim to support more complicated SQL queries, such as GROUP BY, JOIN, COUNT, and SUM, and show the performance. Moreover, we will investigate sophisticated solutions to resist cache attacks. In addition, we will consider dynamic databases and support more complex operations.

Acknowledgements

This research is supported by STRATUS (Security Technologies Returning Accountability, Trust and User-Centric Services in the Cloud), a project funded by the Ministry of Business, Innovation and Employment (MBIE), New Zealand.

References

[1] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A. Sadeghi, "HardIDX: Practical and secure index with SGX," in *DBSec 2017*. Springer, 2017, pp. 386–408.

[2] A. Gribov, D. Vinayagamurthy, and S. Gorbunov, "StealthDB: A Scalable Encrypted Database with Full SQL Query Support," *ArXiv e-prints*, November 2017.

[3] S. Eskandarian and M. Zaharia, "An oblivious general-purpose SQL database for the cloud," *CoRR*, vol. abs/1710.00458, 2017.

[4] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *NSDI 2017*, A. Akella and J. Howell, Eds. USENIX Association, 2017, pp. 283–298.

[5] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace: Oblivious memory primitives from Intel SGX," *IACR Cryptology ePrint Archive*, vol. 2017, p. 549, 2017.

[6] M. Costa, L. Esswood, O. Ohrimenko, F. Schuster, and S. Wagh, "The pyramid scheme: Oblivious RAM for trusted processors," *CoRR*, vol. abs/1712.07882, 2017.

[7] A. Ahmad, K. Kim, A. Kumar, M. I. Sarfaraz, and B. Lee, "OBLIVIATE: A data oblivious file system for intel sgx," 2018.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[9] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *S&P 2000*. IEEE Computer Society, 2000, pp. 44–55.

[10] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *SIGSAC 2015*, I. Ray, N. Li, and C. Kruegel, Eds. ACM, 2015, pp. 668–679.

[11] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *NDSS 2012*. The Internet Society, 2012.

[12] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *USENIX Security 2016*. USENIX Association, 2016, pp. 707–720.

[13] R. Ostrovsky, "Efficient computation on Oblivious RAMs," in *STOC 1990*. ACM, 1990, pp. 514–523.

[14] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *STOC 2009*. ACM, 2009, pp. 169–178.

[15] Y. Ishai, E. Kushilevitz, S. Lu, and R. Ostrovsky, "Private large-scale databases with distributed searchable symmetric encryption," in *CT-RSA 2016*, ser. Lecture Notes in Computer Science, vol. 9610. Springer, 2016, pp. 90–107.

[16] S. Cui, M. R. Asghar, S. D. Galbraith, and G. Russello, "P-McDb: Privacy-preserving search using multi-cloud encrypted databases," in *CLOUD 2017*, G. C. Fox, Ed. IEEE Computer Society, 2017, pp. 334–341.

[17] S. Cui, M. R. Asghar, and G. Russello, "ObliviousDB: Practical and efficient searchable encryption with controllable leakage," 2016, https://www.cs.auckland.ac.nz/~asghar/papers/eprint16-ObliviousDB.pdf.

[18] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[19] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *SP 2015*. IEEE Computer Society, 2015, pp. 640–656.

[20] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *WOOT 2017*, W. Enck and C. Mulliner, Eds. USENIX Association, 2017.

[21] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *USENIX 2017*. USENIX Association, 2017, pp. 557–574.

[22] S. Chandra, V. Karande, Z. Lin, L. Khan, M. Kantarcioglu, and B. M. Thuraisingham, "Securing data analytics on SGX with randomization," in *ESORICS 2017*. Springer, 2017, pp. 352–369.

[23] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *AsiaCCS 2016*. ACM, 2016, pp. 317–328.

[24] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A. Sadeghi, "DR.SGX: hardening SGX enclaves against cache attacks with data location randomization," *CoRR*, vol. abs/1709.09917, 2017.

[25] AMPLAB, University of Califorian, "Big data benchmark," https://amplab.cs.berkeley.edu/benchmark/, last accessed: March 2, 2018.

[26] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: ExitLess OS services for SGX enclaves," in *EuroSys 2017*. ACM, 2017, pp. 238–253.

[27] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *HASP 2013*, 2013, p. 10.

[28] "Introduction to Intel SGX sealing," https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing, last accessed: March 4, 2018.

[29] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *USENIX 2003*. USENIX Association, 2003.

[30] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA 2006*. Springer, 2006, pp. 1–20.

[31] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," in *CHES 2017*, ser. Lecture Notes in Computer Science, vol. 10529. Springer, 2017, pp. 69–90.

[32] J. Ambrose and A. Ignjatovic, *Power Analysis Side Channel Attacks: The Processor Design-level Context*. Omniscriptum Gmbh & Company Kg., 2010.

[33] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *CHES 2001*, ser. Lecture Notes in Computer Science, vol. 2162, no. Generators. Springer, 2001, pp. 251–261.

[34] D. Genkin, A. Shamir, and E. Tromer, "Acoustic cryptanalysis," *J. Cryptology*, vol. 30, no. 2, pp. 392–443, 2017.

[35] "TPC-H," http://www.tpc.org/tpch/, last accessed: March 2, 2018.

[36] "GnuPG," https://www.gnupg.org/index.html, last accessed: March 1, 2018.

[37] "An oblivious general-purpose SQL database for the cloud," https://github.com/SabaEskandarian/ObliDB, last accessed: March 4, 2018.