Supporting Insertion in Encrypted Multi-Maps with Volume Hiding using a Trusted Execution Environment

Shunta Ishihara
University of Tsukuba
Tsukuba, Japan
ishihara@kde.cs.tsukuba.ac.jp

Chiemi Watanabe
Tsukuba University of Technology
Tsukuba, Japan
chiemi@a.tsukuba-tech.ac.jp

Toshiyuki Amagasa
University of Tsukuba
Tsukuba, Japan
amagasa@cs.tsukuba.ac.jp

ABSTRACT

There is a new type of threat to encrypted databases, called “volume leakage,” which causes the volume of data associated with some keys in a multi-map to be leaked. To counter this threat, several methods for encrypting multi-maps to secure the volume of data they contain have been proposed. One such method supports the insertion of data into encrypted multi-maps. However, it suffers from inefficiency caused by its protocol, which stipulates that all data be sent from the server to the client to perturb the insertion position. For this reason, it is not practical for handling large amounts of data. To address this problem, we propose an improved method for encrypted multi-maps that supports data insertion. The proposed method exploits the trusted execution environment (TEE) supported by modern processors, e.g., Intel SGX, for executing perturbation. TEE is a secure area of a main processor that guarantees the confidentiality and integrity of the code and data inside it. Thus, we can secure the volume and insertion position in an encrypted multi-map without conducting costly perturbation on the client-side. We also overcome the challenge of the capacity of the TEE being typically limited and too small to accommodate the multi-map itself. Specifically, we divide the hash table of the multi-map into partitions so that they can be loaded into the TEE and apply perturbation within each partition. We provide a security analysis of the proposed method, w.r.t. the volume hiding and the insertion position hiding. Further, we provide the results of experiments conducted to evaluate the feasibility of potential applications in terms of the processing time and the amount of noise.

KEYWORDS

encrypted multi-map, privacy, secure database, Intel SGX, cuckoo hashing, homomorphic encryption

1 INTRODUCTION

Cloud database services are becoming increasingly popular owing to their many advantages, such as reduced costs in terms of server management, deployment, and operation. Consequently, the number of cloud database services and the volume of data stored in such databases have been drastically increasing. In a cloud database service, the cloud service provider is responsible for database management, which may lead to a security issue because cloud service providers are not always trustworthy; i.e., they may be curious about the private data and could leak the data [1]. To address this issue, many researchers have proposed methods for encrypted databases to realize secure cloud database services [7, 11].

However, it has been pointed out that simply encrypting the data cannot ensure data privacy and could cause potential security threats. One such security threat is *volume leakage*. This security threat involves an adversary obtaining the size (or volume) of the query result, which in turn could reveal information related to the database and/or the query. For example, Kellaris et al. [16] and Grubbs et al. [10] showed that it is possible for an adversary to reconstruct the histogram of key values in the database only by using the volume. In addition, Grubbs et al. [10] reported that the distribution of queries can be inferred from the reconstructed histogram with high accuracy.

To solve this problem, Sarvar et al. [22] proposed a method for volume hiding in an encrypted multi-map. However, their proposed method assumes that the database is stable, and no updates are allowed, which limits its application domain. To allow data insertion in an encrypted multi-map with volume hiding, we previously proposed combining local differential privacy and randomized response to enable volume hiding to support data insertion [12]. However, that method suffers from slow insertion execution due to the protocol used, which stipulates that all data be sent to the client for perturbation of the insertion location. Consequently, it is not practical for large databases.

In the meantime, recent processors are being developed with a trusted execution environment (TEE) [8] to meet the growing demands for secure computation, e.g., Intel SGX [4]. A TEE is an isolated execution environment in a processor that offers protection over the code and data inside, ensuring confidentiality and integrity. In this paper, we propose an encrypted multi-map that supports data insertion using a TEE. The idea is to securely perturb the insertion position using TEE, thereby achieving volume hiding without sending the data from the server to the client. One of the challenges of handling large data using a TEE is that it typically offers limited memory space (e.g., 96MB in Intel SGX). To solve this problem, we divide the data into smaller blocks according to the available memory in the TEE and apply perturbation within the block where insertion occurs.

To verify the utility of the proposed method, we conducted a security analysis concerning hiding the volume and insertion position. We also experimentally evaluated its feasibility in potential applications in terms of the execution time of insertion and the amount of noise necessary to hide the volume and insertion position.

In this work, our target is healthcare data management in a cloud database as the use case. It is common in healthcare applications to share medical records about patients among multiple medical institutions, and leaking the number of patients may cause unintended privacy disclosure. Similarly, when inserting a new patient record, it is necessary to hide the insertion position because it may cause unintended disclosure of privacy, such as the type of disease.

The main contributions of this study are as follows:

- **We propose a method that allows data insertion into an encrypted multi-map while ensuring that the multi-map remains deferentially private [9]** We use dummy insertion with dummy keys and entries to perturb the insertion position. More precisely, for each insertion request, we generate dummy keys based on the randomized response, allowing us to achieve local differential privacy.
- **We propose a scheme that exploits the TEE to perturb data securely.** To deal with large amounts of data that cannot be directly loaded into the TEE, we partition the database (multi-map) into smaller blocks so that we can load each block into the TEE and perturb the insertion location within each block.
- **We evaluate the security of our proposed method.** We use the randomized response mechanism of local differential privacy (LDP) [15] to hide the volume of data, and evaluate its security by finding the privacy budget ϵ of differential privacy. Furthermore, we update multiple insertion positions via the mendacity operation based on cuckoo hashing [21] and evaluate the security of this operation in a TEE.
- **We provide experimental results that verify the feasibility of the proposed scheme using a TEE, Intel SGX.** We compared the performance of the proposed scheme and non-secure baselines in terms of the insertion execution time and the memory occupation time. We also compared the required noise when inserting data under different privacy mechanisms, i.e., DP and LDP. Based on the experimental results, we show our method is useful for the specified use cases. The experimental results show that the method is useful.

The remainder of this paper is organized as follows: Section 2 briefly overviews some preliminaries for this work. In Section 3, we give an overview of existing methods. In Section 4, we discuss the proposed method for insertion on the server and the improvement of volume hiding and also evaluate its security capability. In Section 6, we report on the experiments conducted to evaluate the insertion operation of our method for a large database. Finally, Section 7 concludes this paper and outlines possible future work.

2 PRELIMINARIES

2.1 Cuckoo hashing and its properties

In both this study and our preceding work [12, 22], we utilize cuckoo hashing [21]. Therefore, we will briefly explain it and its properties. Cuckoo hashing utilizes two arrays and two hash functions and is known to be robust. Let us consider inserting a key using cuckoo hashing. If the hash value collides in one of the arrays, the algorithm moves the already stored entry to the other array using the other hash function and inserts a new entry into the vacancy. In this paper, We call the above property of cuckoo hashing the *mendacity operation.* Figure 1 shows the use of two hash functions (h_1 and h_2) and two arrays (T_1 and T_2) in inserting the entry x.

First, the algorithm checks whether the position $h_1(x)$ in T_1 is empty. If the position is empty, it inserts x into T_1 at $h_1(x)$ and completes the operation; otherwise, (as shown in Figure 1), it removes the existing value y and inserts the value x instead. For the removed value y, it checks the vacancy status of $h_2(y)$ in T_2, and inserts y in T_2 at $h_2(y)$ if it is empty; otherwise, it recursively processes the same operation.

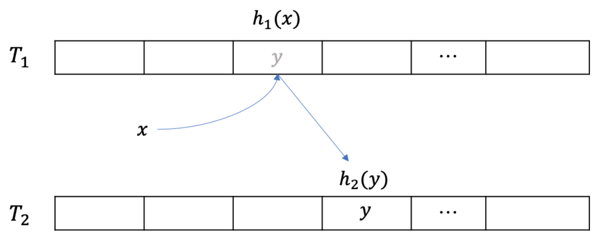


Figure 1: The “mendacity operation” in cuckoo hashing

By creating the hash table in this manner, it is guaranteed that the value x is always in either $T_1[h_1(x)]$ or $T_2[h_2(x)]$, ensuring that it can be searched for with a time complexity of $O(1)$.

There is a possibility that the mendacity operation may take a very long time, or not be finished if we are unlucky. To address this issue, Kirsch, Mitzenmacher and Wieder proposed *stash-based cuckoo hashing* [18]. Stash-based cuckoo hashing avoids the above problem by using the stash. Any value that is not inserted into the table while iterating over the threshold level is inserted into the stash.

2.2 Trusted Execution Environment (TEE)

Recent processors support a trusted execution environment (TEE) [8] to meet the growing demands of secure computation. A TEE is an isolated execution environment in a processor that offers protection for the code and data inside, ensuring confidentiality and integrity.

Specifically, 6th generation (and higher) Intel CPUs support a TEE called SGX [4]. SGX allows us to create a small (e.g., 96 MB) protected memory area called the *enclave* that is isolated from the rest of the system. Hence, we can run programs that are protected from the OS (which is controlled by a third party) and numerous applications/system-level attacks.

The existing SGX is vulnerable to side-channel attacks; e.g., cache-lines [19], branch execution [25], and page-table access [26]. However, the T-SGX [23] and Sanctum [5] systems have evolved to overcome such attacks, and it is believed that future versions of SGX will be resilient to those attacks. In this paper, we do not consider side-channel attacks.

3 EXISTING WORKS

In this section, we introduce several related works as following. In subsection 3.1, we introduce various privacy leakages. In subsection 3.2, we introduce several works of updating for encrypted databases. In subsection 3.3, we introduce the work of Sarvar et al. [22] on which our work is based. In subsection 3.4, we introduce our previous work. In subsection 3.5, we discuss the problems of our previous work.

3.1 Various privacy leakages

Many existing encrypted databases focus on keeping only the content of the data secret. However, it has been reported that the existing encrypted databases are not secure enough. Liu et al. [20] and Islam et al. [13] proposed an access pattern attack method. The access pattern attack is an attack technique that identifies identical queries using data search logs and access frequency. It is prevented by the use of Oblivious RAM (ORAM). ORAM is a technique to hide the access pattern by changing the storage location of the encrypted data each time they are accessed. By using ORAM, an attacker cannot know which data has been accessed, the frequency of searches, or even the relationships between the data. Kellaris et al. [16] and Gurbbs et al. [10] proposed the attack method using the volume of search results. They proposed an attack method that can reconstruct the distribution of data in a database and queries by observing the response size (volume) to a sequence of search queries. There are several types of volume attacks, such as attacks on encrypted multimaps and attacks on range queries. In this paper, we deal with attacks on encrypted multimaps, and in the following, we introduce a method for hiding volume leakage in encrypted multimaps. The naive method of volume hiding is to use padding so that the number of data in all keys is equal. However, it has a large overhead in terms of both storage and communication costs. Kamara et al. [14] and Sarvar et al. [22] proposed a volume hiding method that reduces storage and communication costs compared to naive methods. In particular, since this paper is based on Sarvar et al.'s work [22], we introduce their method in detail in the subsection 3.3.

3.2 Updating for EDBs

In this subsection, we introduce several works of updating for encrypted databases. Chenghong et al. [24] proposed a framework for an encrypted database that prevents privacy from being leaked depending on the time of updating. The privacy leakage due to update time is caused, for example, by event data updates from smart sensors in the building (security cameras, smart light bulbs, Wi-Fi access points, etc.). Even if each event data is encrypted to protect the privacy of the people in the building, the IoT provider can know the private information about the activities in the building from the the event occurrence time without decrypting them. To prevent such privacy leakage, it is necessary to decouple the relationship between event occurrence time and update time. The easiest way to prevent privacy is to not upload the data, in which case data analysts will not be able to utilize the data. The next possible solution is to update the data every unit of time, regardless of the occurrence of events. However, when events occur infrequently, most updates are dummies, and the provider has the problem of wasting resources for unnecessary computations. Chenghong et al. proposed a framework that provides a guarantee of differential privacy in a single update for the update time problem, and can be arbitrarily customized by the user by changing parameters for the three trade-off issues of privacy, data accuracy, and processing performance.

Natacha et al. [6] proposed a key-value store that achieves ORAM-based access pattern confidentiality while providing ACID transactions, which are required in many applications. Existing ORAMs cannot support transactions for the following two reasons. The first is that ORAM is not fault-tolerant. The Second is that ORAM has limited or no support for concurrency. They delay transaction commits until the end of a fixed-size epoch and buffer their execution in a reliable proxy to enhance consistency and durability on a per-epoch basis. Since only the last value of the key changed during an epoch is written back to the ORAM, the number of ORAM operations required to commit a transaction can be reduced, thus reducing the cost of CPU and bandwidth amortization without increasing contention.

3.3 Sarvar's work

Sarvar et al. [22] proposed a method that preserves privacy in multi-maps while achieving volume hiding. In their method, they exploit differential privacy (DP) to hide the volume of data and reduce the communication cost.

Specifically, they originally added dummy entries when applying DP, which, however, increases storage costs. To address this problem, they reduce the server's storage size using cuckoo hashing so that they do not always add dummy entries when inserting a new entry. Let us consider inserting multiple values with the same key k. Instead of inserting values with the same key, they generate (surrogate) keys, i.e., $k + 1, k + 2, \ldots$, and insert the key–value pairs at different hash addresses in the tables according to the generated keys, thereby avoiding the insertion of multiple values with the same key. Consequently, they do not need to insert dummy entries to achieve DP.

However they need to maintain the number of stored values for each key. To achieve this, they introduce a *count table* to maintain the volume of each key. Because it maintains the volume information, they apply DP only on the count table, reducing the extra storage cost of dummy entries.

When searching the multi-map, given a search key k, they first interrogate the count table to get the volume of k, and generate search keys, $k + 1$, $k + 2$, etc. Then, they interrogate the hash table to get the results. Note that the results may contain false positives owing to conflicts in the hash tables, which could be regarded as dummy results for protecting the volume of query results.

Figure 2 illustrates an example of Sarvar's method. In this example, there are two values associated with the key *key1*. In this case, the hash address for the first value (x_1) with the key *key1* is calculated as $h_1(key1 + 1)$ or $h_2(key1 + 1)$, followed by $h_1(key1 + 2)$ or $h_2(key1 + 2)$, etc. The volume of each key is maintained in the count table, where the values are protected by applying DP. When retrieving values associated with a (user-specified) query key $key1$, the system first gets its volume (4) and generates search keys ($key1 + 1$, $key1 + 2$, $key1 + 3$, and $key1 + 4$). Then, it retrieves the values by accessing the hash addresses computed by the generated keys and the hash functions (h_1 and h_2). Note that the results may contain false positives caused by hash collisions (e.g., $h_2(key1 + 2)$ retrieves $(key2, x_2)$), which can be seen as dummy entries that can be filtered out on the client-side.

One major limitation of Sarvar's method is that they do not support any updates on the multi-map.

3.4 Our previous method

To address the limitation of Sarvar's method (i.e., not supporting updates), we previously proposed an extension that enables it to support data insertion [12] while preventing volume leakage. Specifically, our method supports data insertion while maintaining the multi-map as being deferentially private by using randomized response based on local differential privacy (LDP). It hides the position of past insertions using the mendacity operation, a feature of cuckoo hashing. To protect related operations from inside attacks (e.g., by the server administrator), we perform the operation to hide the insertion positions on the client.

More specifically, when inserting a key–value pair, we apply a randomized response, which is a mechanism for achieving LDP, to maintain the differential privacy of the multi-map. The key is

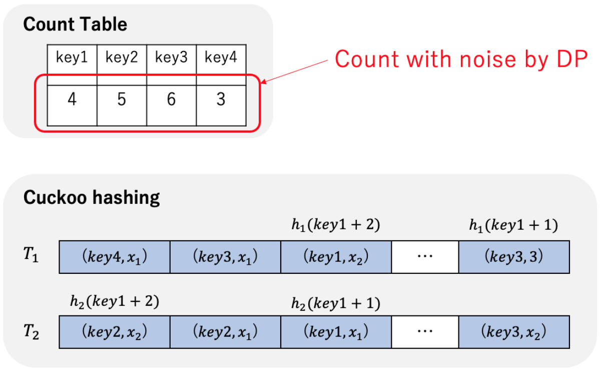


Figure 2: Example illustrating Sarvar's method based on cuckoo hashing.

chosen based on the following formula:

$$key = \begin{cases} key_{true} & \text{with probability } p \\ key_{dummy} & \text{with probability } (1-p) \end{cases} \qquad (1)$$

where key_{true} and key_{dummy} are the keys of the non-dummy and dummy data, respectively. We iterate the randomized response until key_{true} is selected.

We do not use the generated keys for insertion but rather for updating the *count table*. We also use it as done in Sarvar's method [22] to maintain the number of values (volume) associated with each key. Note that the volume count includes both non-dummy and dummy data to guarantee differential privacy. This process is performed on ciphertexts using additive homomorphic encryption.

Figure 3 shows an example of updating the count table by inserting *key1*. Note that *key3* and *key4* are selected as randomized responses. As can be seen, the volume of the dummy key is also updated at the same time so that we can prevent inferring of the update on key_1 from the difference between the volumes before and after the insertion. As the randomized response guarantees LDP, privacy is maintained even after data insertion.

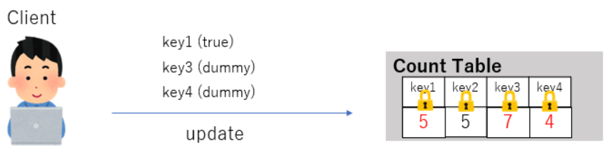


Figure 3: Updating volume when inserting a value with *key*1.

Next, we discuss how to find the privacy budget ϵ of LDP. From the derivation of ϵ in the randomized response, it is found using $ln(\frac{p}{1-p})$ because

$$\frac{Q(key_{insert} \mid key_{true})}{Q(key_{insert} \mid key_{dummy})} = \frac{p}{1-p}$$

where $Q(key_{insert} \mid key_{true})$ and $Q(key_{i}nsert \mid key_{d}ummy)$ are the probabilities that the key being inserted is selected and that it is selected from the dummy, respectively. In addition, to ensure ϵ is non-negative, $p >= 0.5$ is required. Note here that dummy keys may not be selected by the randomized response.

Furthermore, when inserting a new value into the multi-map, we must hide the insertion position so that the server administrator cannot infer the updated data from the index in the tables. We use the *mendacity operation* in cuckoo hashing to hide the insertion position. The mendacity operation is a recursive process that moves existing values to the other side of the hash tables until the values are all stored in the hash tables. As a result, one insertion may cause multiple updates in different positions in the tables, which is useful for hiding the insertion position. If an insertion causes only a few mendacity operations, we insert dummy entries to hide the insertion position.

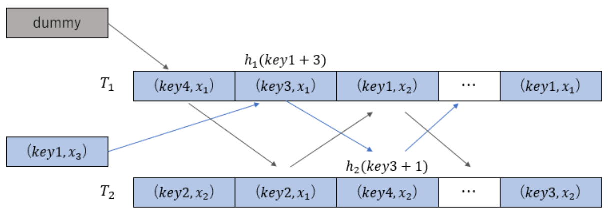


Figure 4: Inserting a data record whose key is $key1$.

Figure 4 shows the process flow for updating the table when $(key1, x_3)$ is inserted. First, the algorithm attempts to store the value at position $h_1(key1+3)$ in table T_1. Then, based on the mendacity operation, the originally stored data are moved to the other table, which results in $(key3, x_1)$ at $h_1(key1+3)$ being removed and stored at $h_2(key3+1)$ in table T_2.

In addition, by adding dummy data at position $h_1(key1+1)$, we intentionally increase the number of updates. We randomly choose the position of the dummy entries. Note that if there are data that we cannot store in the hash tables according to the mendacity operation, we store them in the stash on the client.

3.5 Drawbacks

Figure 5 gives an overview of our previous method, with an example of inserting a new value into *key1*, depicting the mendacity operation for perturbing the updating positions. If the operations are performed on the server, private information may be leaked from the operation history to inside attackers, such as system administrators. For this reason, in our previous method, we send the whole index back to the client to undergo mendacity operations to protect privacy. This incurs a high cost in terms of network traffic, particularly when the multi-map is large, and makes the process slow. This may cause several problems in real applications. One such problem is that the processing performance heavily depends on the client's performance, and in some cases, the client may not complete the process owing to insufficient resources, such as memory. Another problem arises when processing concurrent requests from two or more clients. When updating the multi-map, the map needs to be sent to the client and also locked until the update is completed, which significantly degrades the performance under concurrent access.

4 PROPOSED METHOD

We propose a new insertion method for encrypted multi-maps with volume hiding. To eliminate the drawbacks of our previous method, we do not rely on the client for perturbation of the insertion position. Instead, we exploit the trusted execution environment (TEE) [8] provided by the main processor to perform

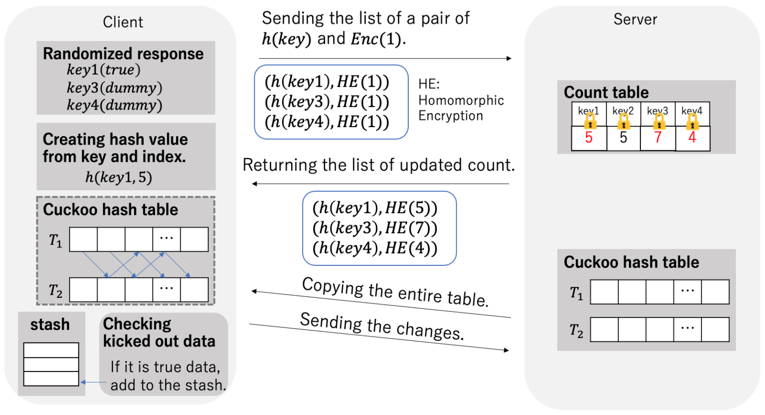


Figure 5: Process flow of our previous method.

multi-map perturbation securely. Specifically, we use Intel SGX as the TEE.

Figure 6 shows an example of inserting $(key1, x_1)$. The main difference from the previous method is that there is an enclave, a TEE provided by Intel SGX, on the server, and we update the hash tables using it. Thus, we update and perturb each hash table without sending the entire hash table to the client. The values in the hash table are encrypted and stored in the server's storage. The hash tables are partitioned into smaller blocks (< 96 MB) so that they can be loaded into the enclave, because the size of the enclave is limited up to 96 MB. When inserting a new dataset, it is loaded into the enclave along with a hash block. We present the details in Section 4.1.

After copying the relevant hash block to the enclave, the mendacity operation (Section 3.4) is applied. Because the process in the enclave is not visible to adversaries, they do not know which part of the hash table has been updated. The entries updated in the enclave are encrypted and written back to the main memory. Note that we can protect the process of perturbing hash entries using the enclave. However, it is possible to observe the updated locations of the hash entries by comparing the table before and after the insertion. In particular, if there is no hash collision, the mendacity operation is not applied, leading to leakage of the updated location. Therefore, we perform the mendacity operation by inserting dummy data when no hash collision occurs to perturb the updated location. In addition, the dummy entries are helpful because the adversary cannot guess how many values have been inserted from the update locations.

After the completion of data insertion, the data removed from the cuckoo hash table are sent back to the client. Even when the removed data are dummy data, they are sent back to the client, and the client can filter the dummies out. This prevents the adversaries from correctly guessing whether the removed data are dummy data or not by observing whether they are sent to the client.

The proposed method basically solves the problems of previous methods by using the SGX. In summary, we make the following two extensions to our previous method.

- Hash table partitioning for handling large amounts of data.
- Hiding the access location against attacks that use the difference in the search results before and after data insertion.

We present the details of the proposed method below.

4.1 Hash table partitioning

We use the enclave provided by Intel SGX to securely execute the mendacity operation on the server side while hiding the updated position from inside attackers. One of the major challenges is that the available memory space in an enclave is limited (96 MB) and is too small to load the hash tables entirely. To cope with this problem, we partition the cuckoo hash tables into smaller encrypted blocks (< 96 MB) and perform data insertions within each block in the enclave. Another factor is the characteristic of Intel SGX whereby the processing inside of an enclave is fast, whereas calling a function in the enclave from outside (or vice versa) is slow (2x to 2000x slower than regular function calls) [2, 17]. Further, the argument size for a function call crossing the border of the enclave is limited to 8 MB for one function call. Therefore, if the block size is set to 96 MB, we need to further divide the block into smaller (< 8 MB) pieces and send from/to the enclave via multiple function calls, which is time-consuming.

Figure 7 shows how a new data record being inserted is assigned to a block and data insertion performed on the enclave. As can be seen, the data being inserted are allocated to each block according to the hash address calculated in the enclave. Next, we copy the destination block into the enclave and also pass the data being inserted to the enclave, followed by the execution of data insertion. In the enclave, the entire block is not decrypted; only the points where the mendacity operation occurs are decrypted. Subsequently, the updated block is returned to the main memory, and the removed data are sent to the client.

4.2 Hiding the access locations in search

To perform a search, as in our previous method, we first obtain the volume of results associated with the search key from the count table and calculate the hash addresses for accessing the hash tables. If we assume that the hash table will not be updated, it is sufficiently safe to add noise to the count table based on DP to hide the number of results. However, when insertions occur, an attacker may infer what data are inserted by comparing the ciphertext before and after the insertion operation. Suppose that several values are associated with the same key key_1. An attacker can infer the updated address by observing the access pattern of queries with the same key (key_1) before and after the latest insertion. Therefore, our method hides the access locations of the data in the search. As in the case of insertion, the accessed data are located in the enclave to hide the access locations. The

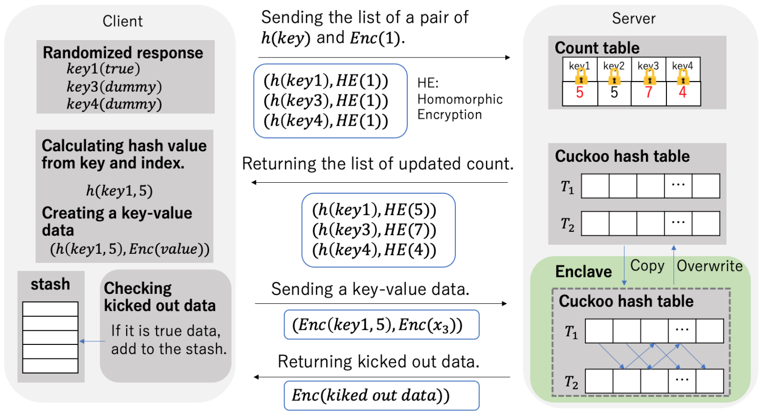


Figure 6: Process flow of the proposed method.

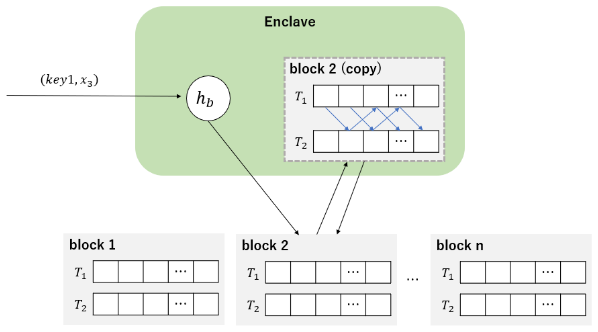


Figure 7: Partitioning hash tables into blocks to perform the mendacity operation in an enclave.

process flow of the search operation is as follows: First, the client sends a request *{h(key)}* to the server to retrieve the volume of the key. Next, the client sends a request *{Enc(key), Enc(volume)}* to the server to retrieve the data from the key and the retrieved volume. The server computes the hash values to identify the block to access and the hash addresses in the hash tables to retrieve the data. Then, the blocks containing the target data are copied into the enclave and the search is performed in turn.

5 SECURITY ANALYSIS

In this section, we evaluate the security of our proposed method. In our proposed method, we assume that the server administrator is semi-honest. This semi-honest administrator follows established protocols but tries to steal the maximum amount of data information they can. The adversary (e.g., server administrator) can observe all systems except the data and operations in the enclave. In addition, we assume that the adversary does not have any background knowledge on the stored data.

With these assumptions as our basis, we analyze whether an adversary can infer what the data are and where they are inserted when our proposed insertion operation is employed. Because the count table is encrypted via homomorphic encryption, the adversary cannot know the value of the count table, although they may know that the value of the count table has been updated. However, when a client gets the values, the client can access the index as many times as the volume of the key allows. Therefore, the administrator may infer the volume from the number of accesses to the index.

The volume information is guaranteed to be deferentially private by adding noise. Let us assume that two accesses occur before and after an insertion, and the adversary observes the difference in the key's volume. In this case, there are two possibilities:

- One addition for an actual (non-dummy) value insertion into the key with probability p.
- One addition for a selected dummy key by the randomized response with probability $(1 - p)$.

In other words, even if the number of reads increases by one, the adversary cannot know whether it is an increase by the insertion of actual or dummy data. Even if the adversary can see the increase in the number of reads before and after the insertion, it is not a problem because LDP is guaranteed.

Next, we describe the security of the mendacity operation to hide the position of the stored data. If there is only one updated position, the position where the actual (non-dummy) data are inserted can be leaked by observing the corresponding position. In our method, at least $k + 1 (k \geq 1)$ positions are updated, where k is a parameter that is set by the user for the number of times to force a mendacity operation to occur. Therefore, the server administrator can only determine where the true data have been inserted with a probability of 1/(k+1) in the worst case. However, the adversary may be able to ascertain the block to which it is assigned. In our experiment, we found that the maximum capacity of one block is approximately 1000 records. Therefore, it is possible to increase safety by perturbing the blocks as well. Specifically, the block that has the true key–value data inserted can be kept a secret by faking the update of the block.

6 EXPERIMENTAL EVALUATION

In this section, we present the details of the experiments conducted to evaluate the proposed method. The specific purpose was to evaluate the improvement in the performance of insertion and query by SGX and the influence of the noise addition under LDP on the communication volume and processing time.

To this end, we conducted the following experiments:

- **Comparison of the processing times of data insertion using the proposed method, our previous method, and a non-secure method using one million artificial data records and real-world data.** In this experiment, we first evaluated whether the processing time for inserting records using our proposed approach is sufficient for practical use by comparing our previous method and the non-secure method. In addition, we evaluated the impact of SGX and LDP on the processing time and the communication volume by comparing the proposed method with the non-secure method.

- **Comparison of the amount of noise between DP and LDP.** In this experiment, we evaluated the impact of LDP on the amount of noise by comparing with DP. Specifically, we compared the amount of noise in the volume when noise is added using DP after all data are inserted and when noise is added using LDP at the time of data insertion.
- **Comparison of the search times of the proposed method, Sarvar's method, and the non-secure method using one million artificial data records and real-world data.** In this experiment, we first evaluated the effect of the difference in noise between DP and LDP on the search time by comparing it with Sarvar's method. Next, we evaluated the impact of SGX on the search time by comparing it with the non-secure method.

The experimental environment was as follows:

- **Server:** Microsoft Azure, Intel(R) Xeon(R) E-2288G CPU 3.70 GHz, 4 GB RAM, Ubuntu 18.04.5 LTS
- **Client:** MacBook Pro, dual-core Intel Core i7 3.1 GHz , 16 GB 1867 MHz DDR3, macOS (ver. 11.5.2)

The datasets used in the experiments were artificial and real-world data. The artificial data comprised 1,000 keys and one million data records. We used two artificial datasets: uniform and normal distributions. We used UCI Online Retail II [3] as real-world data. It has 5,305 keys and 1,067,371 data records.

6.1 Processing time of data insertion

We compared the data insertion processing times of the proposed method, our previous work, and the non-secure method. We introduced our previous method Section 3.4. In the non-secure method, all records and indexes are stored in plain text and no noise entry is added to the count table, although the processing flow is the same as that of the proposed method.

Figure 8 shows the average time taken to insert one key–value pair when a client inserts 1,000 data items using the proposed and our previous method with artificial data following a uniform distribution, artificial data following a normal distribution, and real-world data. When a record is inserted into the multi-map,

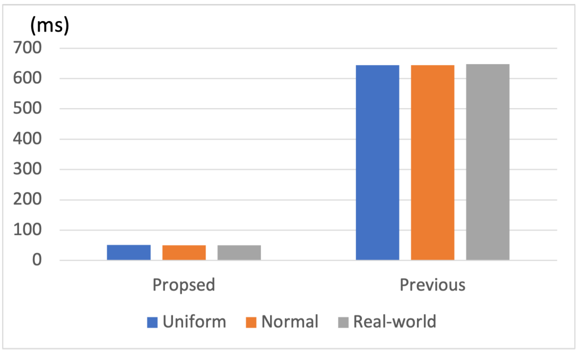


Figure 8: Insertion: proposed method vs previous method.

the hash function is calculated by using the key and the number of stored records according to the key. Therefore, the time required for insertion is not related to the distribution of the key values. The proposed method is approximately 12x faster than the previous method.

Table 1: Processing time breakdown of the proposed method vs the non-secure method.

	Proposed	*Non-secure*
Noise addition (ms)	0.0024	0.0000
Updating count table (ms)	0.0221	0.0004
Updating cuckoo hash table (ms)	10.9438	0.0032
Communication (ms)	2.4915	0.6229
Other (ms)	37.2359	0.0000

Uniform

	Proposed	*Non-secure*
Noise addition (ms)	0.0024	0.0000
Updating count table (ms)	0.0228	0.0005
Updating cuckoo hash table (ms)	10.9699	0.0033
Communication (ms)	2.5335	0.6334
Other (ms)	36.7747	0.0000

Normal

	Proposed	*Non-secure*
Noise addition (ms)	0.0024	0.0000
Updating count table (ms)	0.0225	0.0006
Updating cuckoo hash table (ms)	10.9600	0.0038
Communication (ms)	2.4960	0.6240
Other (ms)	37.0466	0.0000

Real-world

The total processing time of the proposed method for insertion is approximately 118x slower than that of the non-secure baseline. Note that in Table 1, the communication time is estimated based on the observation of the communication volume with the non-secure baseline in Figure 9. The communication time of the non-secure method was obtained by subtracting the processing time from the total time.

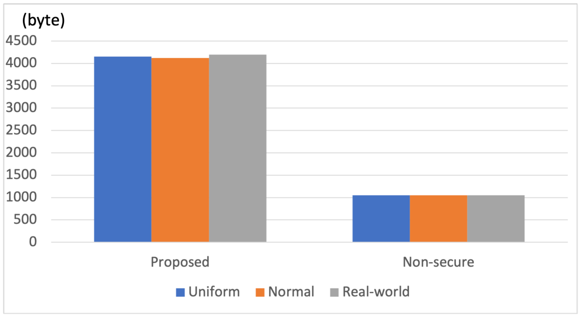


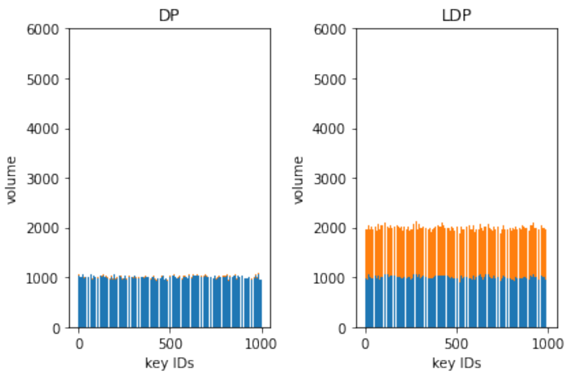
Figure 9: Communication volume comparison.

The communication volume required to insert data is approximately 4x greater than that of the non-secure method. Therefore, the communication time of the proposed method is also four times longer than that of the non-secure method.

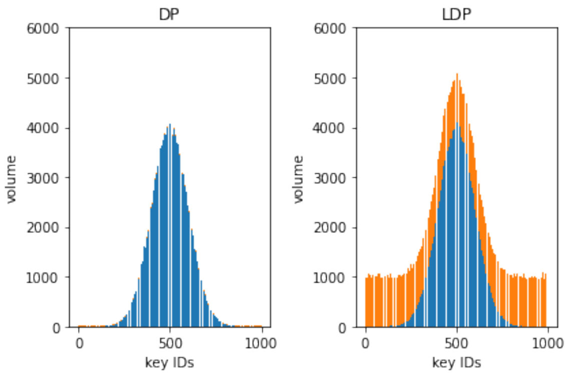
In Table 1, we can see that the proposed method has a significant difference in the cuckoo hash table update and other processing times compared to the non-secure method. The reason for this large difference in updating the cuckoo hash table can be attributed to the characteristics of SGX. SGX is known for its fast processing inside the enclave, but very slow (2x to 2000x) [2, 17] processing of functions calling inside the enclave from outside the enclave (and vice versa) compared to normal function calls. The other processing time is obtained by subtracting each processing time from the overall processing time. This is considered to be the processing time for encryption and compounding. In the proposed method, the encryption-combination is performed 11 times on average for each data insertion. This results in a large overhead.

Thus, although there is more overhead in each process compared to the non-secure method, we believe that the fact that the time required for data insertion is kept at 0.5 seconds while ensuring the high security of our method is practical.

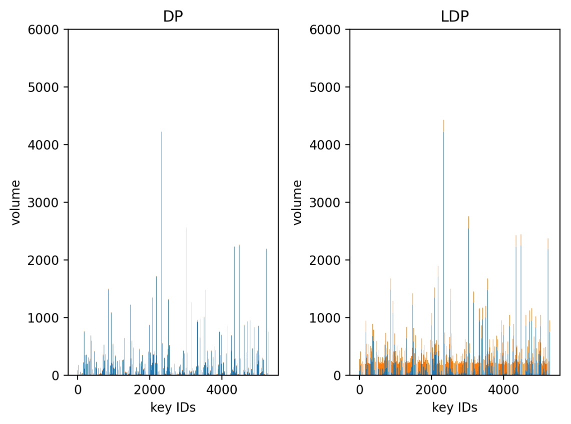
6.2 Noise comparison between DP and LDP



Uniform distribution data



Normal distribution data



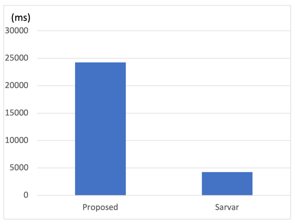
Real-world data

Figure 10: Graph comparing the amount of noise between DP and LDP.

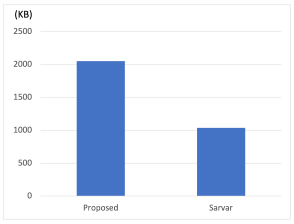
Figure 10 compares the noise in DP and LDP for different datasets: artificial data following a uniform distribution, artificial data following a normal distribution, and real-world data. The results show that the amount of noise in LDP is large compared to DP. In the case of experiments using artificial data, about 1,000 noise units are assigned to each key compared to DP. In the case of experiments using real-world data, about 200 noise units are assigned to each key compared to DP. This is because, in LDP, the amount of noise is proportional to the number of data records. In the proposed method, noise is only added to the number of counts in the count table, but not to the index. Therefore, the amount of this noise does not affect the amount of spatial computation. However, because the search is performed based on the number of count tables, this amount of noise will affect the search time. Therefore, in Section 6.3, we evaluate the impact on the search time.

6.3 Processing time of data search

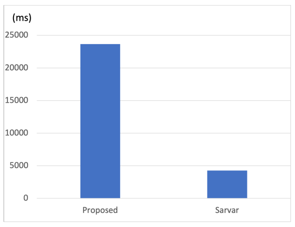
We conducted search experiments for the proposed method and Sarvar's method. Figure 11 compares the average time and the average communication volume by the client to search all the keys for each method in which artificial data following a uniform distribution, artificial data following a normal distribution, and real-world data are stored. The figure shoiws that the amount



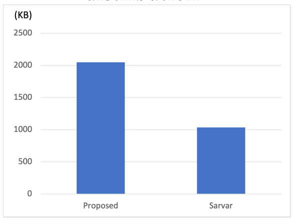
Search time for uniform distribution



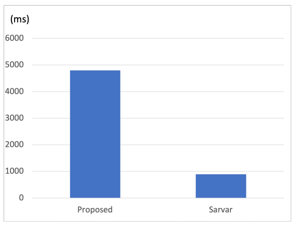
Communication volume of the search for uniform distribution



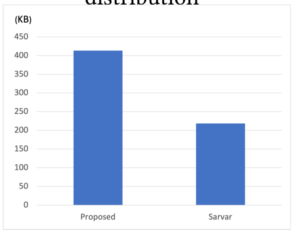
Search time for normal distribution



Communication volume of the search for normal distribution



Search time for real-world data



Communication volume of the search for real-world data

Figure 11: Average search times and average communication volumes.

of noise in the proposed method is about two times lesser than

Sarvar's method, whereas the search time of the proposed method is about five times slower than Sarvar's method. This is due to the overhead caused by SGX. As mentioned in Section 4.2, the proposed method processes search operations in the enclave area to hide the access location during searches. Therefore, we compared the processing times incurred to search data from the cuckoo hash table on the server side. Figure 12 shows the results. As mentioned in Section 6.1, SGX has a characteristic wherein

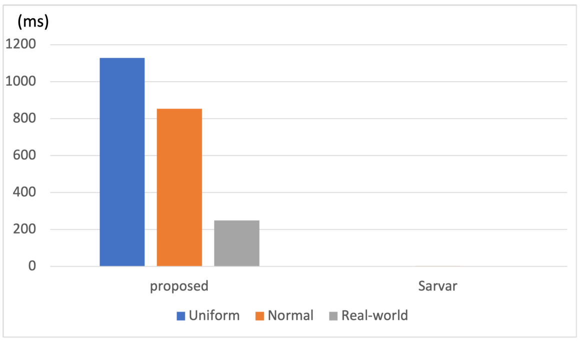


Figure 12: Comparison of cuckoo hash table search times.

the processing inside the enclave is fast, while calling a function inside the enclave from outside the enclave (or vice versa) is very slow. Thus, the processing time is slow when compared to Sarvar's method. However, Sarvar's method searches for the key based on a uniquely defined token generated from the key, and there is a risk that the provider may identify the key. In contrast, the proposed method performs the search process in the enclave, and thus the search is performed in secrecy.

7 CONCLUSION AND FUTURE WORK

In this paper, we proposed a method that supports data insertion in encrypted multi-maps using a trusted execution environment (TEE), such as Intel SGX. In the proposed method, the tables are partitioned into blocks and each block is processed independently to cope with the limited memory space (96 MB) in Intel SGX. Our experimental results indicate that it takes about 50 ms to insert a key–value pair. This is a speedup of about 12 times when compared to our previous method. In addition, we think that it is practical to have the data insertion time at 50 ms while keeping the key volume and update location secret.

The time taken by the search process is about 5x longer than that of Sarvar's method. This is due to the search process using Intel SGX as well as the difference in the amount of noise. In Sarvar's method, the search positions are not secret, whereas in our proposed method, the search positions are secret. Therefore, we believe that our method is more useful when dealing with highly sensitive data.

In future work, we would like to make more effective use of the remaining enclave area. In this method, only 8 MB of the enclave area is used, owing to the argument-size limitation of Intel SGX. We are considering using the remaining area as a cache. In real-world workloads, there are many cases where access to keys is heavily skewed. Thus, we would like to cache frequently accessed keys in the enclave, instead of storing them in the cuckoo hash table, to speed up insertion and retrieval.

ACKNOWLEDGEMENTS

This work was supported by JSPS KAKENHI Grant Numbers JP20362832.

REFERENCES

[1] 2020. X-Force Threat Intelligence Index Reveals Top Cybersecurity Risks of 2020. https://securityintelligence.com/posts/x-force-threat-intelligence-index-reveals-top-cybersecurity-risks-of-2020/.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[3] Daqing Chen. 2019. Online Retail II. UCI Machine Learning Repository.

[4] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086. https://eprint.iacr.org/2016/086.

[5] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 857–874. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[6] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 727–743. https://www.usenix.org/conference/osdi18/presentation/crooks

[7] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2011. Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security* 19 (01 2011), 895–934. https://doi.org/10.1145/1180405.1180417

[8] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. 20–38. https://doi.org/10.1109/SP.2019.00021

[9] Cynthia Dwork. 2006. Differential Privacy. In *Automata, Languages and Programming*, Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12.

[10] Paul Grubbs, Marie-Sarah Lacharite, Brice Minaud, and Kenneth G. Paterson. 2018. Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 315–331. https://doi.org/10.1145/3243734.3243864

[11] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 216–227. https://doi.org/10.1145/564691.564717

[12] Shunta Ishihara, Chiemi Watanabe, and Toshiyuki Amagasa. 2021. Supporting Insertion in Encrypted Multi-Maps with Volume Hiding. In *IEEE International Conference on Smart Computing, SMARTCOMP 2021, Irvine, CA, USA, August 23-27, 2021*. IEEE, 264–269. https://doi.org/10.1109/SMARTCOMP52413.2021.00058

[13] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *in Network and Distributed System Security Symposium (NDSS*.

[14] Seny Kamara and Tarik Moataz. 2019. Computationally Volume-Hiding Structured Encryption. 11477 (2019), 183–213. https://doi.org/10.1007/978-3-030-17656-3_7

[15] Shiva Prasad Kasiviswanathan, Homin K. Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. 2011. What Can We Learn Privately? *SIAM J. Comput.* 40, 3 (2011), 793–826. https://doi.org/10.1137/090756090 arXiv:https://doi.org/10.1137/090756090

[16] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1329–1340. https://doi.org/10.1145/2976749.2978386

[17] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-Memory Key-Value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 14, 15 pages. https://doi.org/10.1145/3302424.3303951

[18] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2009. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM J. Comput.* 39, 4 (Dec. 2009), 1543–1561.

[19] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 557–574. https://www.usenix.org/

conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[20] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-An Tan. 2014. Search Pattern Leakage in Searchable Encryption: Attacks and New Construction. *Inf. Sci.* 265 (may 2014), 176–188. https://doi.org/10.1016/j.ins.2013.11.021

[21] R. Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* 51 (2004), 122–144.

[22] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 79–93. https://doi.org/10.1145/3319535.3354213

[23] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium 2017 (NDSS'17)* (network and distributed system security symposium 2017 (ndss'17) ed.). Internet Society. https://www.microsoft.com/en-us/research/publication/t-sgx-eradicating-controlled-channel-attacks-enclave-programs/

[24] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2021. DP-Sync: Hiding Update Patterns in Secure Outsourced Databases with Differential Privacy. *Proceedings of the 2021 International Conference on Management of Data* (Jun 2021). https://doi.org/10.1145/3448016.3457306

[25] Jinwen Wang, Yueqiang Cheng, Qi Li, and Yong Jiang. 2018. Interface-Based Side Channel Attack Against Intel SGX. arXiv:1811.05378 [cs.CR]

[26] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Oct 2017). https://doi.org/10.1145/3133956.3134038