A Small Dummy Disrupting Database Reconstruction in a Cache Side-Channel Attack

Hyeonwoo Han[1], Eun-Kyu Lee[2]*, Junghee Jo[3]
Department of Information and Telecommunication Engineering,
Incheon National University, Incheon, Korea[1,2]
Department of Computer Education, Busan National University of Education, Busan, Korea[3]

***Abstract*—This paper demonstrates the feasibility of a database reconstruction attack on open-source database engines and presents a defense method against it. We launch a Flush+Reload attack on SQLite, which returns approximate, noisy volumes returned by range queries for a private database. Given the volumes, our database reconstruction uses two algorithms, a Modified Clique-Finding algorithm and Match-Extension algorithm, to recover the database. Experiments show that an attacker can reconstruct the victim's database with a size of 10,000 and a range of 12 with an error rate of up to 0.07% at most. To mitigate the attack, a small dummy data is added to the result volumes of range queries, which makes the approximation more confused. Experimental results show that by adding about 1% of dummy data, an attack success rate (in terms of the number of reconstructed volumes in the database) is reduced to 60% from 100% and an error rate increases to 15% from 0.07%. It is also observed that by adding about 2% of dummy data, the reconstruction is completely failed.**

***Keywords*—Attack; cache attack; side-channel attack; security; database reconstruction; privacy; clique-finding; database volume**

I. Introduction

Data processing continues to become more common in the cloud, and cloud computing is embedded in the business model of popular services such as Google's G Suite, Microsoft's Office 365, Adobe Creative Cloud [1]. In addition to cloud use by industry, federal agencies are also now leveraging cloud services, even for the storage and analysis of sensitive data. Microsoft, for example, won a $10 billion contract from US government that creates a secured cloud for the Pentagon.

One of the most significant security challenges in the cloud is on processing sensitive information. In extreme cases, cloud servers that handle sensitive information may not be trusted because they are presumed to be malicious. In this case, since the data must be encrypted, additional time is required to calculate the encrypted data [2], [3]. In this paper, it is assumed that a trusted server handles sensitive data, but a spying process is also running on the same public server. When the spy process is located in the same physical space as the victim, it shares hardware such as cache, which may act as a side channel. Our assumption is quite reasonable in a realworld setting because leading companies in the cloud service spend great amount of money to show that their servers are trusted.

The goal of this paper is to investigate the impact of side-channels on open-source database engines, to address the risks, and to present a method for defense. The model of the paper assumes that an external user requests a query a private database stored in the victim Virtual Machine (VM), and that the victim VM uses SQLite to process the query and returns the result to the external user. An attacker cannot make a query request directly to the database or observe the results of the query. The attacker is running a spy VM in the cloud with the victim VM, so it can monitor the shared cache to bring in side channel leaks. The attacker's goal is to reconstruct the volume of the victim's database. Reconstruction of the database here refers to recovering the volume rather than finding out all the information correctly. Suppose that a school's student database stores data from <Alice, Bob, Charlie> for grade A, <Dave, Eve> for grade B, <Frank, George, Henry, Ivan> for grade C, for example. The reconstruction is to restore the volume of three rows for grade A, two rows for grade B, and four rows for grade C.

This paper introduces a conventional side-channel attack that captures information leakage in the cache shared by victim VMs and a spy VM using SQLite. This recovers approximate volumes of database including some noise values. We also introduce a database reconstruction attack that uses a clique finding algorithm. It recovers database volumes by constructing a graph based on correct and noiseless volumes of the range queries. In order to make it deal with the approximate, noisy volumes, this paper shows its extension. The extended algorithm includes two sub algorithms. In a modified clique finding algorithm, a concept of noise budget is introduced and utilized in a edge creation step of the clique finding algorithm. A match-extension algorithm handles such a case that the modified algorithm sometimes fails to find a maximum clique in a given graph. Experimental results show that our database reconstruction can recover almost 100% volumes of database with low error rate (0.2-0.7%).

To mitigate the database reconstruction attack, this paper introduces a stratey to add noise to data. The intution behind the strategy is that the clique finding algorithm is an NP-Hard problem. The complexity of the algorithm grows quickly with the size of the range. Thus, adding a small number of nodes to a graph can significantly increase the time required to reconfigure the database; it can even cause the algorithm to fail completely. Technically, a small dummy data is added to the results of range queries. This approach is reasonable in that approximate volume data obtained from the side-channel attack is only exposed to attackers and they are not aware of the existence of dummy data. Regular users (in victim VMs) is able to access raw data and easily seperate the dummies before processing further. Experiments were conducted to investigate

*Corresponding authors.

the impact of the noise (dummy data) on performance of database reconstruction. With 1% of dummy data, only 60% of database were recovered with up to 15% of error rate. When adding 2% of dummy data, no databased was constructed at all; the database construction attack was completely failed.

The rest of the paper is organized as follows. Section II reviews previous research outcomes in cache attacks and database construction attacks. Section III describes a threat model, a cache side-channel attack, that this paper addresses. In Section IV, we introduce a database construction attack that aims to recover the volume of orginal database given noisy data obtained from the cache side-channel attack. Section V introduces a mitigations strategy, which is followed by experiments and results in Section VI. Finally, Section VII concludes the paper.

II. Related Works

Cache attack is performed based on attacker's ability to monitor cache accesses made by the victim in a shared physical system as in virtualized environment or a type of cloud service [4]. Tsunoo et al. [5] introduced it first by showing a timing attack caused by collisions in the memory lookups on a block cipher. Osvik et al. [6] revealed an inter-process leakage via cache's state and showed key extraction of AES. Acıiçmez [7] showed that instruction cache could be a target for attacks. Ristenpart et al. [8] explored the feasibility of side channel attacks on cloud; they demonstrated that an attacker could penetrate VM isoloation and harm confidentiality of victim VMs. In [9], authors introduced a cache side-channel attack technique, Flush+Reload, that exploited weakness of memory pages shared among processes. Cache behaves in a way to leak information on a victim's access to memory lines in shared pages; so that an attack can determine what victim does and infer the data the victim operates on. They also showed that the technique could be used to extract cryptographic keys for RSA. By using the Flush+Reload, Yarom and Benger [10] demonstrated that the ECDSA leaked the nonce and the secret key of the signer, allowing unlimited forgeries. Moghimi et al. [11] introduced an attack tool, CacheZoom, that could allow an attacker to monitor all memory accesses of SGX enclaves with high resolution. They demonstrated the feasibility of AES key recovery. Authors in [12], [13], [14] showed possibility of cache side-channel attacks by monitoring critical operations in AES T-table entry and other operations such as modular exponentiation, multiplication, or memory accesses. Yan et al. [15] used Prime+Probe and Flush+Reload for the cache side channel and successfully obtained a DNN's architecture that was considered a major commercial asset in a business. Hong et al. [16] showed a DNN fingerprinting attack where an attacker followed function invocations corresponding to architecture attributes of the victim network and thus fingerprinted the entire network.

In database reconstruction, Kellaris et al. [17] demonstrated reconstruction attacks on securely outsourcing database storage systems implementing searchable symmetric encryption or order-preserving encryption. Their attacks were developed for a weak adversarial model where underlying query distribution was only known to an attacker. He does not directly query a database and not need to know any of queries nor answers; he may observe encrypted resonses of queries. They identified that either access pattern or communication volume was leaked for range queries. A reconstruction attack run with the auxiliary information in N^4 queries, where N is a domain size, which recovered the number of records having each specific value in a database, i.e., database counts. Lacharité et al. [18] considered a setting where access pattern (a set of records matched by each query) and rank information (the position of a record in a sorted list of records) were leaked. They presented three attacks, full reconstruction, approximate reconstruction, and reconstruction using auxiliary information, that recovered values in different levels of accuracy. Grubbs et al. [19] presented a database reconstruction attack given the volume leakage of the response of range queries. Their attack used a graph-theoretic approach; they reduced database reconstruction to finding a clique in a graph constructed from the volume information. In the graph, volumes were represented as nodes, and an edge connected two nodes when their absolute difference was represented as a node. After multiple iterations of adding and deleting nodes, their algorithm returned the counts of all values in the database. Shahverdi et al. [20], unlike prior a threat model, considered an honest cloud server on which an adversary virtual machine and a victim one co-located. The adversary could not issue queries to the victim's database but obtain information leaked via a shared cache. This implies that he could not obtain response volumes of range queries accurately; instead, obtaining noisy volumes. In this scenario, authors presented a database reconstruction attack by extending the Grubbs's algorithm [19] and by developing a noise reduction algorithm.

Naveed et al. [21] showed attacks on database systems capturing property-preserving encryption. Using encrypted database columns and publicly known auxiliary informtion, they could recover certain encrypted attributes by up to 80%. Kornaropoulos et al. [22] considers an encrypted spatial database that handles data in a geometric space and k-nearest neighbor (kNN) queries that return the nearest k points in the database for a given query point based on distances between points. Authors exploited a query leakage profile and used a convex polytope to characterize a set of reconstructions. The attack could recover values of the database with approximation error of 2.9% to 0.003%. The same authors in [23] developed a reconstruction attack without assumptions about the knowledge of query distribution and the data. The attack exploited search-pattern leakage, computed distances between encrypted values, and eventually recovered plaintext values.

III. Threat Model: Cache Side-Channel Attack

A threat model in this paper assumes multiple users in a cloud server, each of which has access to its own database. For instance, it is assumed that user 1 can access database 1, user 2 can access database 2, and server is a trusted server. Once one of the users becomes an attacker with a malicious intend, he can abuse physical properties of the model; his VM is in the same cloud server and shares the cache with victim VMs. Although an attacker cannot directly query the database of a victim, he can interact with the server to obtain some information about the inaccessible database.

The attacker does not know what range of queries were requested and what rows were returned exactly. This means that only an approximate volume with noise can be obtained.

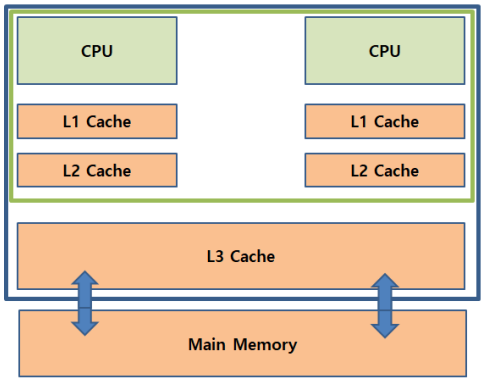

Fig. 1. A general cache architecture.

It is noted that the side-channel attack proceeds regardless of whether the victim's database is encrypted or not, as it proceeds by approximating the volume of range queries. An attack is possible as long as the spy VM can see the code line that the database engine executes for each record returned in the range query.

A. Cache Architecture

Cache is a memory located at the top of the memory hierarchy. It is much smaller in size than the main memory and is very expensive, but it is characterized by a very fast speed. The cache is placed between the CPU and the main memory as shown in Fig.**??**. Data in the main memory is loaded into the cache memory, and performance can be expected to be improved by allowing the CPU to access the cache memory first instead of the main memory with relatively slow access time. The cache serves as a buffer to reduce the speed gap between the CPU and the main memory.

Typical modern CPUs have multiple levels of cache to reduce access time to the main memory. Level 1 cache is the smallest and fastest, while Level 3 cache forms the slowest and largest hierarchical structure. Level 1 cache is divided into two caches, one to store data and the other to hold instructions. In higher-level caches, data and instructions are kept in the same cache. Level 3 cache is a shared memory space where data and instructions are kept in the same cache and is the Last Level Cache (LLC). The LLC includes all lower level architectures, which means that all data present in Level 1 and Level 2 also exists in the LLC. Each cache consists of multiple sets, each set containing multiple cache lines. Each line in the main memory is mapped to a unique set of caches. However, within this set, memory lines can be mapped to cache lines. Typically, each cache line stores 64 bytes of data. If it's already full, we have to decide which memory lines to be removed when writing new lines. This decision is called the Replacement Policy and depends on a cache structure. The most popular replacement policy is the Least-Recently Used (LRU), which replaces the least recently used items with new ones.

B. Flush + Reload Attack

Cache is vulnerable to information leakage because an attacker that is with the victim on the same process can access meaningful information about activities of the victim. In particular, an attacker can monitor and use his own access time to the cache to guess whether the victim has accessed specific memory lines. The reason why such an attack is possible is that the attacker and the victim share the same resource, that is, the cache. Moreover, in a setting where the attacker and victim share the library, both will have access to the physical memory location where a single copy of the library is stored. An attacker can explicitly remove lines corresponding to shared physical memory from the cache. In order to exploit the shared physical memory in a useful way, this paper uses a *Flush+Reload* attack introduced by Yarom and Falkner [9].

This attack method takes advantage of the fact that the access time to cache memory is shorter than that to main memory, and targets the L3 cache line shared by a victim and an attacker. An attacker uses a special command called `clflush` to remove a monitored line from the cache. This command removes the monitored line from the L1, L2, and L3 caches. As mentioned earlier, L3 includes L1 and L2. Thus, if L3 is removed, the lines that are removed are removed from all other caches. The attacker then allows the victim to continue running a program. After a certain period of time, the attacker regains control and measures the memory access time to determine whether the monitored line still remains in the cache. If the reload runs quickly, the monitored line is still in the cache. Then, an attacker deduces that the victim accessed the same line during execution. If the reload runs slowly, the monitored line is not in the cache. Then, the attacker infers that the victim did not access the line while running. Thus, an attacker can know whether the victim has accessed a particular line. To perform the *Flush+Reload* attack, this paper used a package provided in the Mastik framework [24]. Mastik is a toolkit for experimenting with micro-architectural side-channel attacks and provides implementations of published attack and analysis techniques.

C. Detecting Two Lines to Monitor

This paper uses SQLite [25], a popular open-source database engine that uses the BTree data structure to store columns. We looked into the SQLite program and used a `gcov` command to detect a line being called once in each iteration of the range query. It is possible to determine query response volume by monitoring the number of times these rows are called. It should be noted that the duration of each query can also be measured and used as an indicator of the volume. However, there is no reliable way to convert time to volume, resulting in introducing big noise. So we decided to count the numbers explicitly. The library was compiled using the `-fprofile-arcs` and `-ftest-coverage` flags to obtain the number of times each line runs. We, then, ran the range query command and used the `gcov` command to count the number of times each line was executed in the main.c and sqlite3.c files. After finding more lines in the SQLite program, two lines are monitored simultaneously to improve measurement accuracy. The advantage of monitoring two lines is as follow. Even if an attack code may fail to detect activity on one of the lines due to overlap between the attacker reload and victim access, it is still likely to see activity on the second line. There may be some false positives due to the mismatch of hits on the two lines, which is mitigated by considering the number of proximity hits to be from the same activity.

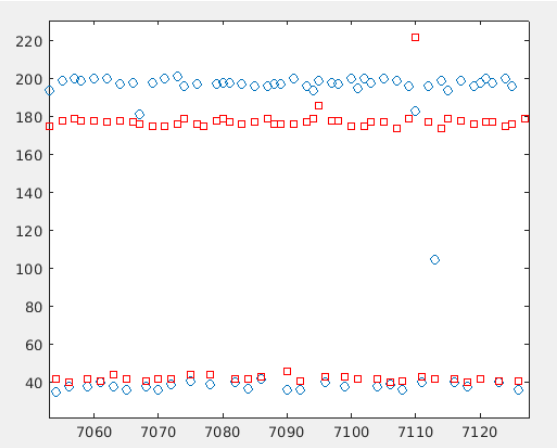

Fig. 2. We launched the Flush+Reload attack on SQLite and showed one sample measurement; the two monitored lines are shown in blue and red.

D. Using Mastik Toolkit

After detecting a line leaking query volume, we, by using the Mastik Toolkit, monitor the lines while SQLite processes range queries. Fig. **??** draws results of one sample measurement. The two lines being monitored are colored blue and red. The x-axis represents the sample point at which the reload occurs, and the y-axis represents the time required to reload the monitored line from memory on that time instance. Because two lines are being monitored, there are two sets of measurements for each time instance.

A `FR-trace` utility in Mastik automatically starts measuring when it detects a hit on one of the lines monitored by the SQLite program. When Mastik does not detect more activities for a while, the measurement automatically end. There are samples with reload time of less than 100 cycles during the interval in which the range query runs. These sample points are recorded when SQLite accesses the line the attacker is monitoring, so the attacker sees a small reload time. Then the number of hits in blue or red measurements are calculated. The number of hits corresponds to the inquiry volume. It is also important to monitor the associated cache lines in order to detect when/whether the range query is executed. Because the measurement is not a noise-free environment, the number of hits counted may differ from the actual value of the volume. Some of the sources of noise are explained below.

False Positive (FP). Instruction to be executed are brought into the cache before memory lines are executed. In the case of the Flus+Reload attack, it still looks that this instruction was executed because of the fast access. In general, true hit counts occur at fixed time intervals. If observing a hit that occurred much earlier than the expected hit time, it is likely a FP. It is presumed to have occurred by speculative execution, and it is not considered to be hit.

False Negative (FN). A FN occurs when the victim accesses the monitored code line after the spy process reloads the line and before the spy process flushes the line. This paper attempts to detect FNs only algorithmically; an asymmetric window around each observed volume is used to compensate for the fact that the actual volume is usually larger than the observed volume. In our experiments, we assign 90% of the window width to a value larger than the observed volume.

E. Approximate Noisy Volumes

A single trace is collected by randomly selecting and executing a range query [a, b] while simultaneously monitoring the line using Mastik FR-trace. This experiment is repeated several times to collect enough traces. We count, for each trace, the number of times that one of the two lines represents a hit, mitigate the FP problem, and then report the number of hits as a range query volume for that trace. Some volumes are observed much more frequently than others, and their values are stored as approximations to the expected volumes. It is expected to see $\binom{N}{2} + N$ values at most in a noiseless setting. There are some traces noted *good enough* but with incorrect volume. By putting all traces together, the effect of such instances would be minimal and the exact approximation of the volume would be distinguishable. It is noted that the volume being recovered is an approximate volume of the database, not a correct volume.

IV. Database Reconstruction

Given a set of volume of database recovered from the cache side-channel attack, the attacker aims at reconstructing the database. This section explains our database reconstruction based on a clique finding algorithm. It shows, first, the Grubbs' work [19] that constructs a graph based on the observed correct and noiseless volumes of the range queries. Note that each volume is represented by a node in a graph. Then, an extention of the clique finding algorithm to handle noisy volumes [20] is introduced.

A. Clique Finding without Noise

The clique finding algorithm has two main parts for graph construction: node creation and edge creation.

1) Creating Nodes: Given a set of recovered volumes V, this part creates a node representing each volume and labels the node with that volume. This means that node v_i has volume v_i.

A recovered volume refers to a volume that has been reconstructed through an attack. Suppose that we have a <5,10,15,20> database of size 4. A query request with a range of 1 will return 5 rows, and a query request with a range of 1 to 2 will return 5 + 10 = 15 rows. It returns 10 + 15 + 20 = 45 rows if a user asks for a query of 2 to 4. From the attacker's point of view, it is not possible to know exactly what range the query request is, but it can be seen that the volume returned is {5,10,15,20,25,30,35}. These volume sets are called recovered volumes.

2) Creating Edges: This part creates an undirected edge between two nodes $v_i, v_j \in V$ if there exists a node $v_i \in V$, where $v_i = v_j + v_k$.

A volume can be recovered by running a clique finding algorithm on the constructed graph. Assuming the value ranges from 1 to N, there are $\binom{N}{2} + N = \frac{N(N+1)}{2}$ possible ranges, so the graph shows $\frac{N(N+1)}{2}$ nodes. Each range $[i, j]$ for $1 \leq i \leq j \leq N$ is denoted by a node. In other words, node $[i, j]$ means

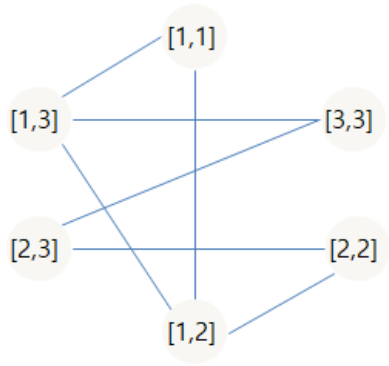

Fig. 3. A graph constructed with nodes of ranges.

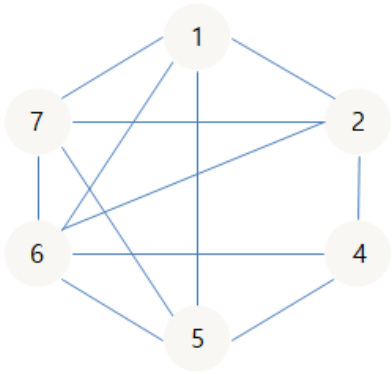

Fig. 4. A graph constructed with nodes of volumes corresponding to Fig. **??**.

that the range of the query is i to j. Nodes corresponding to ranges of of the form $[1, i]$ for $1 \leq i \leq N$, i.e., base volumes, forms a clique for pairs of ranges of $[1, i]$ and $[1, j]$ ($1 \leq i \leq j \leq N$). There is a range of $[i + 1, j]$. This means that there exists an edge between $[1, i]$ and $[1, j]$ due to the way the graph is constructed. The clique finding algorithm finds nodes $[1, i]$, $1 \leq i \leq N$. In order to recover the original range of form $[i, i]$, we simply sort the nodes based on labels and subtract them sequentially because $|[i, i]| = |[1, i]| - |[1, i - 1]|$, $1 < i \leq N$.

TABLE I. A SAMPLE DATABASE OF STUDENTS AND GRADES. IT INCLUDES STUDENTS' NAMES AND THEIR GRADES IN A MATH CLASS

Student name	Alice	Bob	Charlie	Dave	Eve	Frank	George
Grade	1	2	2	2	2	3	3

For instance, suppose a student database shown in Table I. A set of known ranges and corresponding observed volumes are $[1, 1]$ = 1, $[2, 2]$ = 4, $[3, 3]$ = 2, $[1, 2]$ = 5, $[2, 3]$ = 6, $[1, 3]$ = 7. Fig.**??** shows that a graph is constructed based these data, where each node represents a range. The edge creation part created lines connecting two nodes. Since $|[1, 2]| = |[1, 1]| + |[2, 2]|$ and there exists a node $[2, 2]$ connecting $[1, 1]$ and $[1, 2]$, the two nodes are connected via a line. Similarly, $|[1, 3]| = |[1, 2]| + |[3, 3]|$, so there is a connection between $[1, 3]$ and $[3, 3]$.

It is observed that three nodes $[1, 1]$, $[1, 2]$, and $[1, 3]$ in the graph are connected each other and form a complete graph. A subset of nodes in which every two different nodes in an undirected graph are connected is called a clique. Among the cliques that can be found, the largest clique becomes the clique of the graph. Upon finding the clique, $[2, 2]$ can be obtained by subtracting $[1, 1]$ from $[1, 2]$, and $[3, 3]$ by subtracting $[1, 2]$ from $[1, 3]$. This allows us to reconstruct the database easily. Although the exact range is unknown, if the $\{1, 4, 2, 5, 6, 7\}$ volume set is recovered through an attack, it is possible to find 1, 5, 7 cliques as shown in Fig. **??** and reconstruct the database in the same way.

B. Modified Clique Finding with Noise

From the cache side-channel attack in Section III, correct volumes may not be recovered; that is, recovered ones are approximate and noisy volumes. Therefore, a conventional clique finding algorithm that assumes noiseless volumes (described in Section IV-A) does not find cliques of sufficiently large size. This happens because the condition for connecting nodes v_i and v_j almost always fails since it is not possible to find the third volume v_k, where the equation $v_i = v_j + v_k$ is satisfied. This implies that too many edges are missing in the constructed graph to form large cliques.

To mitigate the effect of noise, our method modifies the second parts (Creating Edges) in the clique finding algorithm, a *Modified Clique-Finding algorithm*. Because traces are noisy, it may not possible to obtain the correct volume. The recovered volume is close to the correct volume, and error ranges can be calculated. Here, we define a *noise ratio* as (the recovered volume / the correct volume). An attacker, at the first step, performs a preprocessing by launching an attack on a database known to the attacker. The attacker then examines the recovered volume, selects a noise ratio, and compares it with the actual volume, from which he evaluates the accuracy of the traces to find an approximate value for the noise ratio. Based on all the noise ratios, the attacker then sets a value for *noise budget* - the average of the noise ratios observed across all volumes. The attacker, once the noise budget is fixed, then creates a window of acceptable values around it for each recovered volume. The window is created around v_i using lower bound of $v_i \times (1 - 0.1 \cdot noise\ budget)$ and upper bound of $v_i \times (1 + 0.9 \cdot noise\ budget)$. The window is asymmetric with 90% of the width on the right side because the noisy volume is usually smaller than the actual volume. For volume v_i, the window is denoted as $w(v_i)$. The modified part in the clique finding algorithm that enables to construct a graph from nosiy volumes is as follow.

- *Creating Edges (Modified):* This part creates an undirected edge between two nodes $v_i, v_j \in V$ if there exists a node $v_k \in V$, where $|v_i - v_j| \in w(v_k)$.

C. Match-Extension

Although the noise budget is appropriately adjusted, the Modified Clique-Finding algorithm sometimes fails to find the maximum clique of size N. This section decribes an improvement of the algorithm to handle such a case. First, note that in the clique finding algorithm, a graph corresponding to the volume in the range $[1, i]$ for $1 \leq i \leq N$ should have a clique if there is a full range of volumes present in the data.

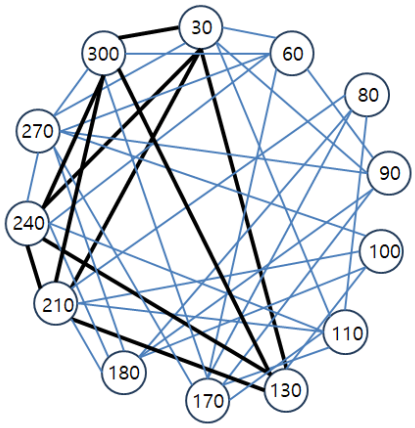

Fig. 5. A graph constructed from noiseless volumes in the database <30, 100, 80, 30, 60>. The maximum cliques corresponds to N, 5. The maximum cliques found are shown in bold connections.

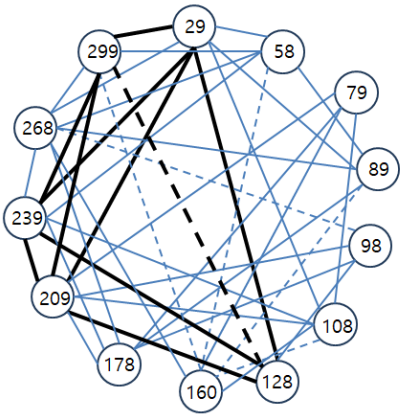

Fig. 6. A graph constructed from noisy volumes in the database <30, 100, 80, 30, 60>. A missing edge in the maximum clique is represented in the dotted line.

Now suppose we are given an approximate, noisy volume corresponding to the range $[i, j]$. $|[1, i]| \approx |[1, i-1]| + |[i, j]|$ should have a connection, but the connection from node [1,j] to node [1,i-1] is missing. A maximum clique of size N is not formed as a result of missing connections. Executing the clique finding algorithm on data with missing volumes returns cliques of size smaller than N and recovers the candidate database for each clique. The algorithm then merges information from these small databases to create a larger database.

1) Observation: This subsection explains the idea of this improvement with an example. Consider a database with a range of 5 possible values (i.e., N = 5), <30, 100, 80, 30, 60>. This means $|[1, 1]|$ = 30, $|[2, 2]|$ = 100, $|[3, 3]|$ = 80, $|[4, 4]|$ = 30, $|[5, 5]|$ = 60. A set of possible values for the volume of range queries V is, then, {30, 60, 80, 90, 100, 110, 130, 170, 180, 210, 240, 270, 300}. For instance, $|[1, 2]|$ = 130. A graph constructed for these volumes is shown in Fig. 5. The maximum cliques found by the algorithm are shown in bold connections. The nodes {30, 130, 210, 240, 300} are returned, and a database <30, 100, 80, 30, 60> is reconstructed.

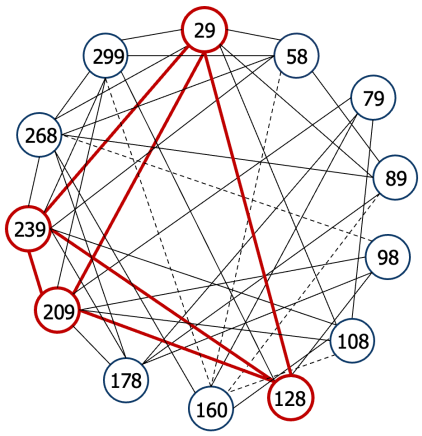

Fig. 7. Given a graph missing edges, the Modified Clique-Finding algorithm returns a clique of size 4 (less than N = 5) with values {29, 128, 209, 239}.

Now, assume that the recovered volume has noise and that a set of possible values for the volume of range queries V is {29, 58, 79, 89, 98, 108, 128, 160, 209, 239, 268, 299} as shown in Fig. 6. Almost all the noisy volumes are close to the true values; exceptionally volume 160 is far from the correct volume 170. In this setting, our method uses the Modified Clique-Finding algorithm to construct a graph. It first uses a window around a volume v_i allowing to have lower and upper bounds of $v_i - 1$ and $v_i + 3$, respectively. Some connections will be missing due to measurement errors. For example, a connection from node 299 to node 128 will not be formed because there is no window containing 171 anymore. If our method runs the algorithm on the new graph, the result will be a smaller clique (less than N = 5). As shown in Fig. 7, the algorithm returns cliques whose size is 4 with values {29, 128, 209, 239}, which creates a database $< 29, 99, 81, 30 >$. In the next, we describe main steps in the Match-Extension algorithm to reconstruct a database from noisy volumes.

2) Finding cliques: This step aims at finding all the cliques in the constructed graph.

Maximum Clique. The first stage is to find the maximum clique in the graph. Let K be the size of the maximum clique recovered in this stage. If multiple cliques with the same maximum size are found, one of them is chosen randomly. Once a clique is found, a corresponding database is reconstructed. We call this database an initialized solution, `initSolution`. If the size of the maximum clique found in this stage is N, it is done; the database is successfully reconstructed. Otherwise, the rest parts of the algorithm extend this `initSolution`.

All Other Cliques. The next stage recovers all cliques of sizes $K, K-1, K-2, \ldots,$ and $K-l$, and sorts them from the largest size to the smallest size. For each clique, a corresponding database is reconstructed and referred to as a candidate solution `candSolution`. It is in the form of a sorted list of volumes corresponding to contiguous ranges in the database. The cliques found in this stage are not limited to ranges of the form $[1, 1]$, $[1, 2]$, $\ldots$, $[1, K]$ for some K. Note that this holds true even in the noiseless setting. Any set of volumes in the range of $[i, i_1]$, $[i, i_2]$, $\ldots$, $[i, i_k]$, where $i \leq i_1 < i_2 < \cdots < i_k$, forms a clique of size k if all

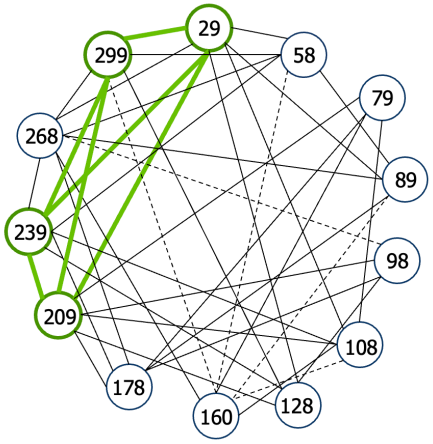

Fig. 8. The Modified Clique-Finding algorithm returns a clique with values {29, 209, 239, 299} (size = 4), which creates a database $\langle 29, 180, 30, 60\rangle$

.

differences in volumes corresponding to the ranges exist in the data. This fact allows this algorithm to recover the original database by discovering the volumes of different parts of the actual database and combining those parts.

3) Combining solutions: Given the `initSolution` and a `candSolution` in the form of lists of volumes, in this step, the Match-Extension algoriithm *combines* the information in them into one large solution.

Longest Common Substring The first stage is to find the longest common substring of the two solutions; it is the longest contiguous list of volumes where the two solutions match. We call this substring `lcList`. To find it, our method uses a modified standard longest common substring algorithm where the elements of the substring are approximately equal to the corresponding elements of `initSolution` and `candSolution`. This stage returns the `lcList` and the starting and ending indices (locations) of `lcList` in two given solutions. However, even after the first stage, there may be volumes where `initSolution` and `candSolution` match, which are not recognized in the first stage. In order to show an example, we take the previous sample from Fig. 7 and let the `initSolution` $\langle \mathbf{29}, 99, 81, 30\rangle$. `candSolution` is $\langle \mathbf{29}, 180, 30, 60\rangle$ (see Fig.8 for its construction). In this case, `lcList` can be found as $\langle 29\rangle$, but it is observed that the two solutions match at $\langle 99, 81\rangle$. This information is presented as a volume of a range $\langle 180\rangle$ that is the union of two adjacent ranges in `initSolution`.

Extension Our combine algorithm identifies such a case and extends `lcList` accordingly in the next stage. It searches for cases where a particular volume v_i next to the end of `lcList` in one solution (say `initSolution`) is approximately equal to the sum of the volumes $u_j, u_{j+1}, \ldots, u_{j+r}$ if $r \geq 0$ next to the end of `lcList` of the other solution (e.g. `candSolution`). The algorithm, then, extends the `lcList` by adding $\langle u_j, u_{j+1}, \ldots, u_{j+r}\rangle$ and changing the endpoints of the `lcList` in `initSolution` and `candSolution`. In the database example above, the combine algorithm examines the neighbors of $\langle 29\rangle$ and finds that $180 \approx 99 + 81$, and extends the `lcList` to $\langle 29, 99, 81\rangle$. It again examines the neighbors of $\langle 29, 99, 81\rangle$ and finds $30 \approx 30$, extending the `lcList` to $\langle$**29, 99, 81, 30**$\rangle$. It is noted that while the values in the example are exactly the same (e.g., $180 = 99 + 81$) by chance, the algorithm also accepts the case where values are approximately equal.

$$\texttt{initSolution} = \langle \texttt{prefix1, lcList, suffix1}\rangle$$
$$\texttt{candSolution} = \langle \texttt{prefix2, lcList, suffix2}\rangle$$

After the `lcList` is maximally extended, the two solutions have the form above. Any of prefixes and/or suffixes can be empty. The algorithm then performs one of four options:

(a) If `prefix1` and/or `suffix1` is empty, then it extends the `lcList` to `lcList` = `prefix2`||`lcList` and/or to `lcList`||`suffix2`.

(b) If `prefix2` and/or `suffix2` is empty, then it extends the `lcList` to `lcList` = `prefix1`||`lcList` and/or to `lcList`||`suffix1`.

(c) If the lengths of both `prefix1` and `prefix2` are 1 (say, their volumes are a and b, with $a < b$), and if the absolute value of difference appears in the volume measurement, then `lcList` = $\langle b-a, a\rangle$||`lcList`. This option also applies to the case when the lengths of both `suffix1` and `suffix2` are 1.

(d) Otherwise, the algorithm stops combining and repeats stages for another `candSolution`.

Back to our example, we found the `lcList` to be $\langle$**29, 99, 81, 30**$\rangle$. The last stage has `initSolution` = $\langle$`lcList`$\rangle$ and `candSolution` = $\langle$`lcList, 60`$\rangle$. This is the case of option (a) where `suffix1` is empty. Thus, the algorithm adds `suffix2` = $\langle 60\rangle$ to `lcList` and returns `initSolution` = $\langle$**29, 99, 81, 30, 60**$\rangle$ as a solution.

4) Finding best solution: Once a combine is successful, two solutions are merged and one larger solution is created. The clique finding algorithm may not discover this large solution in the first place because some volumes or connections in the graph were missing, preventing potential cliques corresponding to this solution from being formed. Whenever two solutions are combined, it identifies the number of missing volumes that prevented finding the combined solution. In fact, if the missing volumes could be added to the graph and starts the algorithm, we could get the combined solution in all listed solutions. Therefore, the number of missing volumes is used as a metric to evaluate how good a candidate solution is. On one hand, showing few missing volumes indicates that the `initSolution` and `candSolution` match on many volumes in the database and therefore are compatible. On the other hand, a large number of missing volumes suggests that two solutions may have different information about the volumes. At this, last step finds a candidate solution among all cliques having the lowest number of missing volumes with respect to being combined with the `initSolution`.

V. Mitigation Strategy

Given noisy volumes, the Modified Clique-Finding algorithm described in the previous section shows a high attack success rate (as will be shown in Section VI); that is, it finds

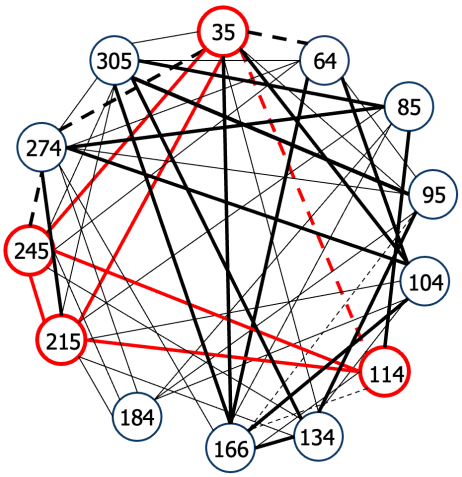

Fig. 9. A graph constructed from noisy volumes in the database $\langle 30, 100, 80, 30, 60\rangle$ when 6 dummy elements are added to the query responses.

the maximum clique, finds the longest common substring to combine subsolutions, and eventually reconstructs the victim's database correctly with high probability. However, it is noted that the clique finding algorithm is an NP-Hard problem that can be only solved by checking the numbers of all cases one by one, not by polynomial solutions. The complexity of the algorithm grows quickly with the size of the range. We figure out that it works well for the ranges up to size 15 in practice. Adding a small number of nodes to a graph can significantly increase the time required to reconfigure the database; it can even cause the algorithm to fail completely. It is also important to remind that the goal of the attack is to reconstruct the volume, the size of columns on which a victim is making range queries, not exact records.

Based on the observation, this paper proposes to add noise to mitigate the reconstruction attack. Technically, dummy elements are added to results of range queries, which makes it more difficult for an attack to recover approximate volumes in the side-channel attack. The increased size of volume may create edges that should not exist or not create edges that is essential to recover a database. This directly causes the first step to find incorrect cliques; say, the size of the maximum clique found may be greater than N. The next step is also affected; it may fail to find the longest common substring or result in reconstructing a wrong database eventually. This approach works because the attacker is concerned only about the volume where the elements are not exposed obviously. However, regular users (i.e., victims) are not confused with the additional elements because they can access exact records in query responses and discern such elements immediately.

Back to our example of database, $\langle 30, 100, 80, 30, 60\rangle$ in Fig.5. Remind that $|[1,1]|$ = 30, $|[2,2]|$ = 100, $|[3,3]|$ = 80, $|[4,4]|$ = 30, $|[5,5]|$ = 60, and a set of possible values for the volume of range queries V is {30, 60, 80, 90, 100, 110, 130, 170, 180, 210, 240, 270, 300}. Now, suppose that 6 dummy elements are added to the response of each range query. Then, $|[1,1]|$ = 36, $|[2,2]|$ = 106, $|[3,3]|$ = 86, $|[4,4]|$ = 36, $|[5,5]|$ = 66, and V = {36, 66, 86, 96, 106, 116, 136, 176, 186, 216, 246, 276, 306}. It is noted that $|[1,2]|$ = 136 (= 130 + 6), not $|[1,2]| + [2,2]|$ = 36 + 106 = 142, because 6 dummy rows are being added to the result. Fig. **??** shows a graph constructed from the given volumes when 6 dummy elements are added. Comparing the edges in black to Fig. 6, it is observed that new connections are formed (solid bold lines) and some connections are missing (dotted bold lines). The edges in red correspond to the clique found in Fig. 7. It is observed that a connection between node 35 and node 114 (a dotted red line) is not formed and thus the clique is not found in the algorithm. Missing one clique implies that we lose a string information (a contiguous list of volumes) in it, which hiders from finding `lcList` and eventually leads to the failure of the combining step. The algorithm may find another clique and successfully reconstruct a database, but it must be far from the original database.

VI. Experiments and Results

This section performs experiments for four sets in total and shows their results. The first two sets (I and II) are the cases where an attacker launches a cache side-channel attack to measure approximate noisy volumes and then performs database reconstruction to recover a victim's database. In experiment I, distribution over all the possible queries are uniform; that is, each range are queried with equal probability. Experiment II performs non-uniform range queries. Although each range must be queried at minimum number of times, the query distribution do not need to be uniform. For this, `rand()` function is used in our system. The rest of settings are the same for I and II. In the rest two sets (III and IV), our countermeasure strategy is applied with different percentages of additional dummy elements and with different query distributions. With results, we assess their effects on the database reconstruction attack.

A. Preliminary

For experiments, this paper uses the Nationwide Inpatient Sample (NIS) 2008 dataset [26]. The NIS is part of the Healthcare Cost and Utilization Project (HCUP) used to analyze national trends in healthcare. This data is collected annually and it has approximately 5 to 8 million hospitalization records. Since data is in an SPSS file format, we use IBM SPSS Statistics [27] to read it. Then, 10 SQLite database is created by selecting 10,000 records randomly out of 8 million ones. We perform range queries on the AMONTH element, an "admission month coded from (1) January to (12) December" (refer to [26] for the full description of elements). The list of ranges are $[1,1], [1,2], \ldots [1,12], [2,1], [2,2], \ldots [12,11], [12,12]$ and possible values in the range are $N = 12$. For the non-uniform range queries, ranges $[i, j]$ are set by randomly selecting i and j in $1 \leq i,\ j \leq 12$. For each set, experiments are repeated 10 times and average values are returned.

We run experiments on a desktop computer with Intel Core i5-9400F CPU @ 2.90 GHz running Ubuntu 16.04. The capacities of the L1, L2, and L3 caches are 384 KB, 1.5 MB, and 9 MB, respectively. For a database engine, we use the SQLite Amalgamation version 3.20.1 with a single large file of C-code named `sqlite3.c` [28]. Table II summarizes information of our setup. We found in a heuristic manner that when approximately 120 measurements are collected for

TABLE II. SETUP INFORMATION FOR EXPERIMENTS

Dataset	NIS 2008 database [26]
Noise budget	0.002
Possible values in the range	12
Total number of records	10,000
Capacities (L1, L2, L3)	384KB, 1.5MB, 9MB

a range query, the aggregated measurements in the cache side-channel show a peak regarding the approximate volume. Because there are 78 different range queries for our data, around 10,000 traces are collected first to see if the traces for each range are sufficient so that we can check a peak for each approximate volume.

B. Database Reconstruction from Noisy Volumes

For Experiments I and II, this paper collects 10,000 traces corresponding to 10,000 range queries; that is, 1 trace for each query. It then processes all of those traces to get a set of approximate, noisy volumes, with which the Modified Clique-Finding algorithm and the Match-Extension algorithm are performed. This returns an output of reconstructed database. The former algorithm runs with different values for the noise budget, whereas the latter algorithm is with a fixed value 0.002 of noise budget. For each value in $1, 2, \ldots, 12$, it is expected to recover a candidate volume that corresponds to the number of records in the database using that value.

For a database of size N, a *success rate* is defined as the number of candidate volumes recovered out of N. For instance, if our method recovers 10 candidate values in our experiments with the range of size $N = 12$, the success rate is $(\frac{10}{12}) \times 100$ [%]. It is noted that an attacker can distinguish between successful and unsuccessful attacks because he knows N, the size of the database that needs to be recovered. We also define an *error rate* of a recovered volume. For each candidate volume $\tilde{v}$ recovered, we compare it to the corresponding value v in the actual database and report the error rate of $(\frac{|\tilde{v}-v|}{v}) \times 100$ [%]. For example, if the actual volume is 1,000 and the recovered volume is 980, then the error rate is reported as 0.2%. If our database reconstruction recovers 10 values out of 12, error rates only for the recovered 10 values are reported.

From experiments, it is observed that increasing a noise budget increases the average error rate and the confidence interval for the Modified Clique-Finding algorithm. For noise budgets of 0.005, some of the databases recovered in Experiment I were very far from the true database, resulting in much larger error intervals. In short, increasing the noise budget seems to have helped increase the success rate, but as the error rate increases, the quality of the recovered database decreases. It is also noted that its average running time increases with the size of the noise budget. In the case of the Match-Extension algorithm, the average error and the width of the error interval are similar to those in the Modified Clique-Finding algorithm. But, it achieves much higher success rate with a small noise budget. Experiments in this paper fix the noise budget of 0.002 and thus the average running time remains low. The Modified Clique-Finding algorithm with a noise budget of 0.006 shows better success rate than with a smaller noise budget, and is selected as an algorithm comparable to the Match-Extension algorithm. For Experiment II, this paper performed non-uniform range queries and used the same sets of database as in Experiment I. As mentioned earlier, we need approximately 120 measurements for a range query to detect approximate, noisy volumes from the cache side-channel. This do not require that the query distribution be uniform. Technically, this experiment tests an hypothesis that the success of a database reconstruction attack relies on the capability of identifying peaks corresponding to the volumes of range queries. In this sense, a query distribution in which a peak disappears as its neighboring peaks dominate it could be challenging to attackers. One setting for the challenging distribution, for instance, is that a range [*a, b*] is queried more often than a range [*c, d*] when two ranges have close volumes (i.e., close peaks). Our random setting generate this distribution with probability, given requirement that each query be observed at least 120 times.

TABLE III. RESULTS (SUCCESS RATE AND ERROR RATE) FROM EXPERIMENTS I AND II

	Exp. I	Exp. II
Success rate [%]	100 %	100 %
Error rate [%]	0.4-0.7 %	0.2-0.7 %

Table III shows our results from Experiments I and II. For Experimental I with uniform distribution for all queries, it was possible to reconstruct all 10 databases with the error rate of up to 0.7%, which is considerable in accuracy. Even in the case of Experiment II, where the query range was not uniform, we succeeded in reconstructing 10 databases correctly. The error rate was also within 0.7%. It can be seen that our database reconstruction attack has 100% success rate with low error.

C. Database Reconstruction with Dummy Data

For Experiments III and IV, our method adds dummy data to response volumes returned by range queries. The rest of settings are same to those in Experiments I and II. Experiment III and IV perform uniform and non-uniform range queries, respectively and use approximate, noisy volumes. The database used in the experiment has 10,000 rows, and the range of the AMONTH attribute is 1 to 12. Thus, there are about 800 rows in each range. In the first setting, our method adds 8 dummies (rows) to the response of each range query. That is, it corresponds to 1% of total volumes.

Table IV shows our results from Experiments III and IV. In both experiments, 6 out of 10 databases (60%) were successfully reconstructed, and the error rate in the reconstructed databases was 9%-15%. They are comparable to results, 100% and 0.7%, in previous experiments in Table III. It is reminded that the success criterion for reconstruction is that all 12 candidate values is recovered. 60% of recovery, therefore, is far from the success of an attack. In order to investigate the impact of dummy data further, we add 2% of dummy data out of total volumes in the second setting. Table V shows its results. In both experiments, no database was reconstructed at all. This is mainly attributed to the fact that the database construction attack primarily relies on the volume size of query responses. It is so sensitive to the size that small inclusion of dummy can even weaken its success rate.

TABLE IV. RESULTS (SUCCESS RATE AND ERROR RATE) FROM EXPERIMENTS III AND IV (1% OF DUMMY)

	Exp. III	Exp. IV
Success rate [%]	60%	60%
Error rate [%]	10-14%	9-15%

TABLE V. RESULTS (SUCCESS RATE AND ERROR RATE) FROM EXPERIMENTS III AND IV (2% OF DUMMY)

	Exp. III	Exp. IV
Success rate [%]	0%	0%
Error rate [%]	-	-

VII. CONCLUSION AND DISCUSSION

This paper investigated the impact of side-channels on open-source database engines. We triggered a side-channel attack on a cache shared by victim VMs and a spy VM using SQLite to obtain approximate, noisy volumes of the database. Two algorithms that extended a clique finding algorithm were introduced in order to perform a database reconstruction attack. This paper also introduced a mitigation method that added additional dummy data to results of range queries. Experiments were conducted with a database of 10,000 records and with 12 ranges for queries. Results showed that the database reconstruction attack could recover almost 100% volumes of the database with maximum error rate of 0.7%. With 1% of additional dummy data, however, only 60% of database were recovered with up to 15% of error rate. When adding 2% of dummy data, no database was constructed at all; the database construction attack was completely failed.

As a future work, it would be interesting to investigate the impact of the volume of dummy data further. Given the number of records (10,000) used in our experiment, the size of the volume for each range query is about 800. 1% of additional data represents 8 records only, which does not seem to be a problem. But, as the size of the database increases, this volume also increases. This will naturally increase the processing time of query requests and use a lot of memory. It is necessary to find solutions to defend against attacks without performance degradation in large scale database settings. It would be also interested to examine our models on a scenario where victim VMs and a spy VM do not share a library with a more generic form of cache side-channel attack [6].

ACKNOWLEDGMENT

This work was supported by Incheon National University Research Grant in 2022.

REFERENCES

[1] "Cloud Services," https://www.datamation.com/cloud-computing/slideshows/.

[2] K. Lewi and D. J. Wu, "Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, October 2016, pp. 1167–1178.

[3] R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakri, "CryptDB: protecting confidentiality with encrypted query processing," in *ACM Symposium on Operating Systems Principles*, October 2011, pp. 85–100.

[4] "Cache side-channel attack," https://en.wikipedia.org/wiki/Side-channel_attack#cache_side-channel_attack.

[5] Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Hiyauchi, "Cryptanalysis of Block Ciphers Implemented on Computers with Cache," in *ISITA*, October 2002.

[6] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *Cryptographers' Track at the RSA conference*. Springer Berlin Heidelberg, February 2006, pp. 1–20.

[7] O. Aciiçmez, "Yet another MicroArchitectural Attack:: exploiting I-Cache," in *ACM workshop on Computer security architecture*, November 2007, pp. 11—18.

[8] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," in *ACM conference on Computer and communications security*, November 2009, pp. 199—212.

[9] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, August 2014, pp. 719–732.

[10] Y. Yarom and N. Benger, "Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack," Cryptology ePrint Archive, 2014. [Online]. Available: https://eprint.iacr.org/2014/140

[11] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX Amplifies the Power of Cache Attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*, August 2017, pp. 69–90.

[12] A. C, R. P. Giri, and B. Menezes, "Highly Efficient Algorithms for AES Key Retrieval in Cache Access Attacks," in *IEEE European Symposium on Security and Privacy*, March 2016, pp. 261–275.

[13] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a Minute! A fast, Cross-VM Attack on AES," in *International Workshop on Recent Advances in Intrusion Detection*, September 2014, p. 299–319.

[14] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache Attacks Enable Bulk Key Recovery on the Cloud," in *International Conference on Cryptographic Hardware and Embedded Systems*, August 2016.

[15] M. Y. C. W. Fletcher and J. Torrellas, "Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures," in *USENIX Security Symposium*, August 2020, pp. 2003–2020.

[16] S. Hong, M. Davinroy, Y. Kaya, S. N. Locke, I. Rackow, K. Kulda, and D. D.-S. andand Tudor Dumitras, "Security Analysis of Deep Neural Networks Operating in the Presence of Cache Side-Channel Attacks," 2018. [Online]. Available: https://arxiv.org/abs/1810.03487

[17] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic Attacks on Secure Outsourced Databases," in *ACM SIGSAC Conference on Computer and Communications Security*, October 2016, p. 1329–1340.

[18] M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage," in *IEEE Symposium on Security and Privacy*, May 2018.

[19] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, October 2018, pp. 315—331.

[20] A. Shahverdi, M. Shirinov, and D. Dachman-Soled, "Database Reconstruction from Noisy Volumes: A Cache Side-Channel Attack on SQLite," in *USENIX Security Symposium (USENIX Security)*, August 2021, pp. 1019–1035.

[21] M. Naveed, S. Kamara, and C. V. Wright, "Inference Attacks on Property-Preserving Encrypted Databases," in *ACM SIGSAC Conference on Computer and Communications Security*, October 2015, p. 644–655.

[22] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "Data Recovery on Encrypted Databases with k-Nearest Neighbor Query Leakage," in *IEEE Symposium on Security and Privacy*, May 2019, pp. 1033–1050.

[23] ——, "The State of the Uniform: Attacks on Encrypted Databases Beyond the Uniform Query Distribution," in *IEEE Symposium on Security and Privacy*, May 2020, pp. 1223–1240.

[24] "Mastik: A Micro-Architectural Side-Channel Toolkit," https://github.com/0xADE1A1DE/Mastik.

[25] "SQLite," https://www.sqlite.org/.

[26] "Introduction to the HCUP Nationwide Inpatient Sample (NIS) 2008," https://hcup-us.ahrq.gov/db/nation/nis/NIS_Introduction_2008.jsp.

[27] "IBM SPSS Statistics," https://www.ibm.com/products/spss-statistics.

[28] "The SQLite Amalgamation," https://www.sqlite.org/amalgamation.html.