applied sciences

MDPI

Article

SQLSnoop: Secondary DBMS Attack by Expanding SQL Injection Techniques

Dowon Jeong [1], Jiho Kim [2], Aymen Fatima [3] and Daehee Jang [3,*]

[1] NAVER Corp., Seongnam 13561, Republic of Korea; rubiya805@gmail.com
[2] School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA; jkim4050@gatech.edu
[3] Department of Computer Science and Engineering, Kyung Hee University, Yongin 17104, Republic of Korea; fatimaaymen@khu.ac.kr
* Correspondence: daehee87@khu.ac.kr

Abstract

SQL Injection is a well-known vulnerability that has persisted in web applications for decades. A widely held assumption among developers is that even when SQL Injection is present, hashing or encrypting sensitive data using SQL-provided cryptographic functions, such as `sha256()` or `md5()`, renders stolen data unrecoverable. This paper challenges that assumption directly. We demonstrate that invoking cryptographic functions within SQL statements does not protect plaintext credentials against an attacker who already has SQL Injection access, not because the hash functions are weak but because their plaintext arguments are transiently exposed in DBMS in-memory monitoring views before the hash function executes. We exploit this window using a technique that we call *SQLSnoop*, which repurposes built-in SQL looping constructs to poll the monitoring view at high frequency within a single injected statement. We demonstrate *SQLSnoop* against four major RDBMS platforms: `MySQL`, `MSSQL`, `Oracle`, and `PostgreSQL`. Systematic quantitative evaluation is conducted on MySQL, while feasibility on MSSQL, Oracle, and PostgreSQL is confirmed through working Proof-of-Concept implementations against each platform's respective in-memory monitoring view. Our evaluation on MySQL shows attack success rates consistently above 90%, reaching 100% at 1.2 or more virtual CPU cores, and holding across all four Data Manipulation Language operations. The key practical implication is that SQL-layer hashing is fundamentally insufficient as a defense against SQL Injection, and sensitive data must be hashed at the application layer before the SQL statement is constructed.

Keywords: view table exploitation; PROCESSLIST vulnerability; database query interception; credential theft; real-time monitoring; hashing bypass

Academic Editor: Christos Bouras

Received: 15 May 2026
Revised: 8 June 2026
Accepted: 8 June 2026
Published: 12 June 2026

1. Introduction

SQL Injection has remained one of the most persistent threats to web application security for decades. The vulnerability arises when applications fail to separate client-supplied data from server-side SQL commands, typically due to inadequate input sanitization during query construction. The Open Web Application Security Project (OWASP) ranked injection vulnerabilities first among the top ten security risks in 2017 and third in 2021 [1], reflecting how deeply this problem is embedded in modern web development. OWASP also maintains a dedicated SQL Injection prevention cheat sheet [2] that developers continue to refer to widely. Even widely adopted frameworks such as Django require careful handling

to avoid introducing SQL Injection flaws [3], and the sheer volume of legacy systems still in production means that vulnerable codebases are unlikely to disappear in the near future.

In response, a widely accepted defensive practice has emerged: even if an attacker gains SQL Injection access to a database, sensitive data such as passwords should be stored only in hashed or encrypted form, rendering stolen records practically useless. SQL itself supports this through the built-in cryptographic functions `md5()`, `sha256()`, and `encrypt()` and others that allow developers to hash data directly within the SQL statement before it reaches permanent storage. This approach has become so commonplace that it is treated as a fixed safeguard: even a fully compromised database exposes only ciphertext, not recoverable credentials [2–4].

This paper challenges this assumption. We show that invoking cryptographic functions within SQL statements does not protect sensitive data against an attacker who already has SQL Injection access, not because the hash functions themselves are weak but because the plaintext arguments to those functions are transiently visible before the functions ever execute. Every major Relational Database Management System (RDBMS) maintains an in-memory monitoring view that records SQL statements during execution. When a developer writes `MD5('secretdata')` inside an SQL statement, the string `'secretdata'` sits in that view in plaintext for the duration of query execution—before `MD5()` has had any effect. An attacker with SQL Injection capability can read this view continuously, intercepting credentials in that window. We call this technique *SQLSnoop*.

The distinction we draw is between what SQL Injection can steal from stored data—which hashing effectively protects—and what it can steal from the query processing pipeline, which hashing does not protect at all. Previous work and developer guidance have addressed the former. The latter remains unexamined. SQLSnoop does not require breaking any cryptographic primitive. The hash is irrelevant because the plaintext is captured before the hash runs.

Mounting a practical SQLSnoop attack is non-trivial. The window during which an SQL statement resides in the monitoring view is brief, and a naive polling attempt will miss it far more often than not. The technical contribution of this work lies in crafting SQL payloads that exploit loop-like behavior in database built-in functions—re-purposing mechanisms such as `BENCHMARK()` in MySQL, `REPLICATE()` in MSSQL, `GENERATE_SERIES()` in PostgreSQL, and `CONNECT BY LEVEL` in Oracle—to poll the monitoring view at high frequency within a single injected statement. This makes the attack reliable under real-world conditions without requiring repeated round trips to the database over a network.

We demonstrate SQLSnoop against all four major RDBMS platforms: `MySQL`, `MSSQL`, `Oracle`, and `PostgreSQL`. In each case, credentials passed to database-side hash functions are recovered in plaintext despite being stored only as hashes. Our experimental evaluation, conducted on MySQL across a range of versions and resource configurations, shows attack success rates consistently above 90% and reaching 100% once the database container has 1.2 or more virtual CPU cores available. The results hold across the `INSERT`, `SELECT`, `UPDATE`, and `DELETE` operations, confirming that the vulnerability is not limited to account creation scenarios. The practical implication for developers is direct: sensitive data must be hashed at the application layer before the SQL statement is constructed, not inside it.

2. Background

2.1. SQL Injection Attack Types

A database is most susceptible to SQL Injection attacks when it executes dynamically assembled SQL commands. However, mitigating SQL Injection is not as straightforward as simply disallowing SQL statements enclosed within quotation marks. There exist more sophisticated threats, such as second-order SQL Injection [5] and blind SQL Injection [6],

which pose greater challenges for mitigation efforts. A comprehensive review of detection and prevention techniques for SQL Injection attacks is provided in [7–9]. Beyond these well-documented attack categories, tiny oversights or vulnerabilities can be exploited in unexpected ways in SQL Injection attacks, resulting in a significant impact on the security and integrity of the database. These vulnerabilities can be exploited in numerous inventive ways to compromise the database's defenses.

2.2. DBMS Schema, View Tables, and Data Flow in Web Applications

A database schema serves as a fundamental architectural blueprint for organizing a database, defining the structure and interconnections among data entities, tables, and associated data models [10]. Within this schema, views function as virtual tables that selectively expose subsets of data by projecting only user-required attributes [11]. Modern RDBMS platforms maintain several system-level views for administrative and monitoring purposes. In MySQL, for example, `INFORMATION_SCHEMA` provides a collection of read-only views that expose server metadata, table definitions, column data types, and access privileges [12]. Among these, the `INFORMATION_SCHEMA.PROCESSLIST` view records all SQL statements currently in execution, including their full query text. The security implications of this monitoring view in the context of SQL Injection are examined in Section 3.2.

Figure 1 illustrates the data flow within a typical web application, from the client browser to the database layer, and identifies the critical point at which credential security diverges between two architectural approaches. When a user submits login credentials, the username and password are transmitted from the browser to the web server as an HTTP POST request. The web server forwards this request to the application server, which constructs and sends an SQL statement to the database.

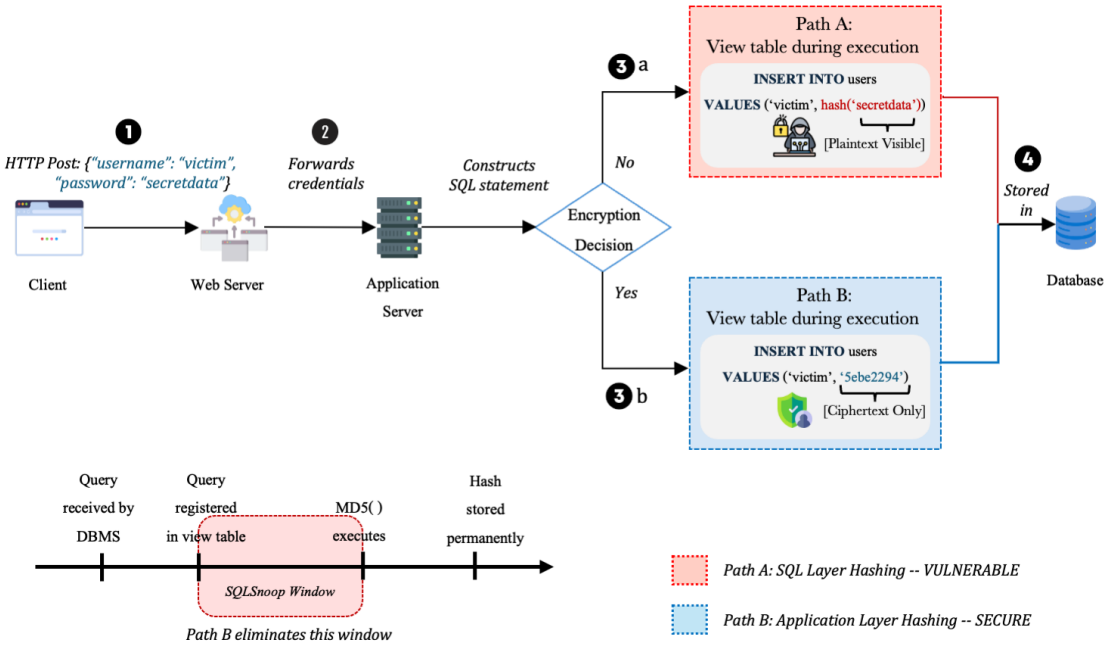

Figure 1. Data flow from client to database illustrating two credential hashing architectures and their divergent security properties under SQL Injection. **Path A** (red): the application server transmits the plaintext credential as an argument to a database-side hash function, e.g., `MD5('secretdata')`. The SQL statement is registered in `INFORMATION_SCHEMA.PROCESSLIST` prior to hash execution, exposing the plaintext parameter during the SQLSnoop interception window τ_{sql}. **Path B** (blue): the hash is computed at the application layer before SQL transmission; only the ciphertext reaches the DBMS, and `PROCESSLIST` contains no recoverable plaintext. Both paths produce an identical stored hash value; the vulnerability in Path A arises solely from the timing of hash computation relative to query registration in the monitoring view.

To protect sensitive data at rest, any plaintext credential received by the application server must be stored in hashed or encrypted form within the database [13]. Two distinct approaches exist for achieving this, and their security properties under SQL Injection differ fundamentally.

In **Path A** (depicted in red in Figure 1), the application server passes the plaintext credential directly as an argument to a cryptographic function provided by the database within the SQL statement itself, for example:

```
INSERT INTO users VALUES ('victim1', hash('secretdata'));
```

In **Path B** (depicted in blue), the application server computes the hash independently before constructing the SQL statement, transmitting only the resulting ciphertext to the database.

```
INSERT INTO users VALUES ('victim1', '5ebe2294ecd0...');
```

Although both paths store an identical hash value in permanent database storage, they differ critically in what is transiently registered in the DBMS monitoring view during query execution. In Path A, the full SQL statement containing the plaintext argument `'secretdata'` is written to `INFORMATION_SCHEMA.PROCESSLIST` at the moment of query receipt, before the `hash` function executes. This creates a brief but exploitable window τ_{sql} during which the plaintext credential is accessible to any party capable of reading the process list. In Path B, no plaintext ever reaches the DBMS layer, and the process list entry exposes only ciphertext, rendering any interception attempt ineffective.

This paper demonstrates that an attacker with SQL Injection capability can systematically exploit this window in Path A by polling `PROCESSLIST` at high frequency—a technique we term *SQLSnoop*. Critically, this attack does not require breaking the underlying hash function. The cryptographic strength of `MD5()` or `SHA256()` is irrelevant; the plaintext is intercepted before cryptographic processing begins. The distinction between Path A and Path B thus represents not merely a stylistic implementation choice but a fundamental security boundary, the implications of which are examined throughout this paper.

3. SQLSnoop Design and Practicality

In this section, we explain the technical design of SQLSnoop and discuss its practicality in a real-world attack scenario. Figure 2 depicts the simplified steps of the SQLSnoop attack, which steals *supposedly-encrypted* user password information from a DBMS. The attack proceeds as follows:

Step ①: The attacker crafts the SQLSnoop payload and delivers it to the target system through an SQL Injection vulnerability.

Step ②: The injected payload begins to execute inside the DBMS, repeatedly polling the in-memory view table to intercept any concurrently executing SQL statements.

Step ③: The victim triggers a sign-up or login action, causing the web application to assemble and submit an SQL query containing the plaintext password as an argument to a database-side hash function (e.g., `hash('thisissecret')`).

Step ④: The DBMS receives the assembled SQL query and registers it in the Memory-Backed View Table prior to execution. At this moment, the plaintext argument is transiently visible in the view.

Step ⑤: The SQLSnoop payload detects the victim's query in the view table, applies a filter to exclude its own entries, and captures the plaintext credential before the hash function executes.

Step ⑥: In conjunction with Step ⑤, the DBMS executes the hash function and writes the resulting ciphertext to the File-Backed Main Table. The plaintext credential is never written to permanent storage.

Step ⑦: Once execution is complete, the query is flushed from the view table. The attacker retains the captured plaintext credential, while the database holds only the hash.

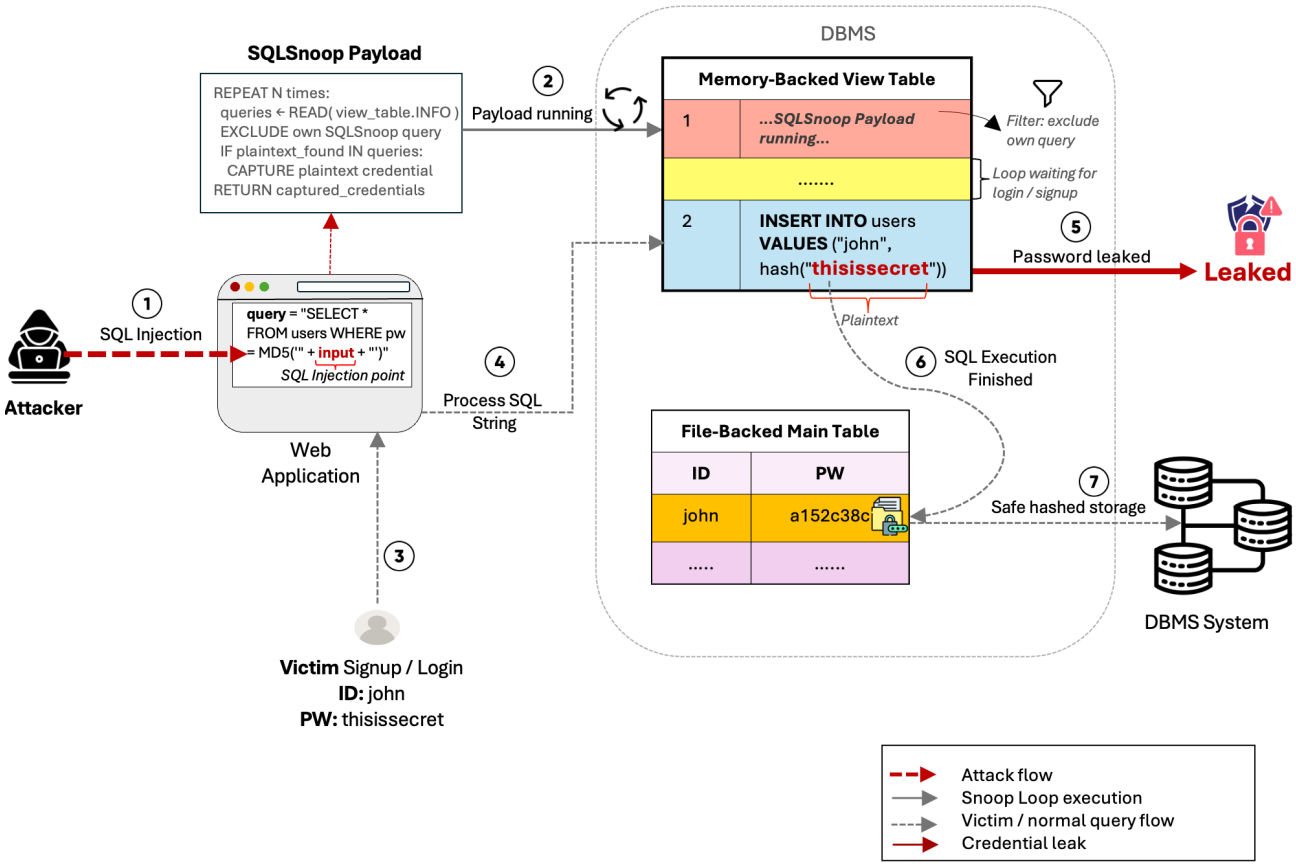

Figure 2. SQLSnoop Attack Flow. The attacker intercepts plaintext credentials from the Memory-Backed View Table before the database-side hash function executes.

In short, SQLSnoop bypasses DBMS-side encryption and hashing by intercepting the SQL query on the fly, during the window between query registration in the view and hash function execution. In the following sections, we investigate the practicality of this attack under real-world conditions.

3.1. SQLSnoop Attack Probability

The key principle of SQLSnoop is based on the transient nature of real-time DBMS data, which exists only briefly within in-memory database views. Consequently, intercepting SQL statements from these views is inherently challenging.

To increase the chances of a successful SQLSnoop, it is necessary to perform SQLSnoop statements repetitively, as depicted in Figure 3. In Figure 3, the red lines signify each instance in which the view was accessed to fetch an executing SQL statement. A blue block represents the presence of an SQL statement in a DBMS view, with the block's length denoting the statement's lifespan in the view. The lifespan is the cumulative duration, which encompasses the time required for the SQL statement to execute and the time needed for it to be flushed from the view.

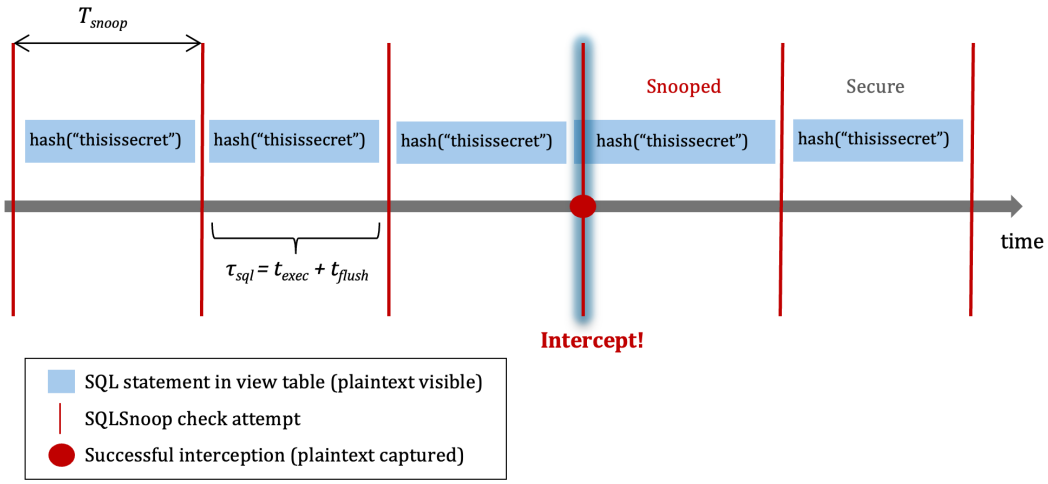

Figure 3. Conceptual Diagram of SQLSnoop Attack. SQLSnoop attempts (in red) are repeated to target temporary data (in blue) within view tables.

The fourth SQL statement is successfully snooped since the red line intersects with the blue block. Conversely, the last SQL statement remains secure as the view was not accessed while the statement was present. The probability of successfully snooping an SQL statement increases as the lifespan of the statement in the view extends and the period of SQLSnoop attempts decreases. If the statement's lifespan exceeds the snooping period, it will invariably be intercepted. This probability can be defined as follows:

$$P_{\text{snoop}} = \begin{cases} 1, & \text{if } \tau_{\text{sql}} \geq T_{\text{snoop}} \\ \dfrac{\tau_{\text{sql}}}{T_{\text{snoop}}}, & \text{else} \end{cases} \quad (1)$$

The duration required to flush an SQL statement from the view is contingent on an undisclosed algorithm and, thus, is beyond our purview. However, the execution time of an SQL statement can be logically calculated, relying on factors such as the statement's load, the DBMS scheduling, and the hardware performance allocated for the task. Similarly, the duration of SQLSnoop is subject to the same factors when the SQLSnoop statement is executed in a loop within the database. If the SQLSnoop statement is transmitted in a loop from the remote attacker's code to the database, the snooping period would substantially increase due to the overhead associated with network data transmission delay, in addition to SQL-irrelevant processing times at intermediary nodes.

In our threat assessment, we seek to evaluate the viability of our attack in snooping victim SQL statements within the brief window during which they appear in the view. In theory, the computational overhead introduced by hash functions such as MD5 increases the execution time of the SQL statement, thereby extending the window during which the statement remains visible in the view.

3.2. View Tables

Our research focus is particularly directed toward views designed to transiently store SQL statements during their execution. In particular, all four main Relational Database Management Systems (RDBMSs), namely MySQL, MSSQL, Oracle, and PostgreSQL, incorporate views that can be exploited to intercept SQL Statements during execution, as comprehensively outlined in Table 1.

Table 1. SQL Statement Views and Privilege Requirements across Major RDBMS Platforms.

DBMS	Reserved Name of View	Privilege Req.
MySQL	`INFORMATION_SCHEMA.PROCESSLIST`	None (default access)
MSSQL	`sys.dm_exec_sql_text`	None (default access)
Oracle	`V$SQL`	Required
PostgreSQL	`pg_stat_activity`	None (default access)

For instance, MySQL's `INFORMATION_SCHEMA.PROCESSLIST` includes an `INFO` column, providing read access to the SQL statement that is currently being executed by a thread [14]. Similarly, in MSSQL, the dynamic management view of the system (e.g., `sys.dm_exec_requests`) provides an `sql_handle` column [15], which can be used as an argument for another system dynamic management view (e.g., `sys.dm_exec_sql_text`), thus revealing the text column containing the current running SQL query [16]. Oracle's dynamic performance view (e.g., `V$SQL`) provides an `SQL_FULLTEXT` column, which offers analogous functionality, granting access to the running SQL query as a string [17]. Finally, within PostgreSQL, the dynamic statistics view (e.g., `pg_stat_activity`) provides a query column that facilitates the retrieval of executed SQL statements in a similar fashion [18].

Aside from Oracle, all DBMSs by default grant access to these view tables. In Oracle, the `V$SQL` view is part of the dynamic performance views accessible through the `SELECT_CATALOG_ROLE` role [17]. This is the minimum privilege required: an attacker who has obtained SQL Injection access and holds `SELECT_CATALOG_ROLE` can read `V$SQL` without requiring full DBA or SYSDBA privileges. While the principle of least privilege may prevent arbitrary database users from holding this role, it is commonly granted to application accounts and monitoring users in enterprise deployments, making it a plausible assumption in many real-world configurations.

3.3. Experiment Design

To investigate the practicality of the SQLSnoop technique, we designed an experimental system. Specifically, we simulated commodity servers using physical nodes with a system architecture consisting of four Docker containers: `attack`, `user`, `backend`, and `database` (we also use these keywords as container names throughout this paper). While it is customary to have both front-end web and backend servers, our experiment simplified this configuration by having only a backend server, with user requests directly transmitted to it. Figure 4 illustrates the overall layout and evaluation parameters of the experiment, where solid arrows represent attacker-controlled flows and dashed arrows represent victim-triggered flows.

- **Step** ①: The `attack` container sends the SQLSnoop payload to the `backend` server via a solid arrow, representing the attacker-controlled flow. Within our threat model, we assume the attacker possesses the capability to execute complete SQL payloads through SQL Injection (some SQL Injection vulnerabilities do not allow the attacker to execute a full SQL statement; this is discussed further in Section 6). For convenience, we implemented a *backdoor* on the backend server to receive SQLSnoop queries from the attack container, simulating the SQL Injection vulnerability.
- **Step** ②: The `backend` server forwards the SQLSnoop payload to the `database` (arrow ②). The payload is designed to iterate snooping attempts continuously and intercept any victim SQL statement containing sensitive information. Detailed implementation is explained in Section 4.2. The number of iterations was adjusted to ensure that the SQLSnoop payload remained active throughout the execution window of all targeted SQL statements. Once inside the DBMS, the payload enters the READ

PROCESSLIST loop, repeatedly checking the in-memory view for any concurrently executing SQL statements that do not belong to the attacker itself.

- **Step ③ (victim side)**: Independently and concurrently, the `user` container initiates one of the backend RESTful APIs via a dashed arrow, providing randomly generated usernames and passwords. This represents the victim-triggered flow—the user has no knowledge of the ongoing attack.
- The `backend` constructs and forwards the victim's SQL statement to the `database` (arrow ④), where it is temporarily stored in the PROCESSLIST view prior to execution. In this step, we use SQL-provided cryptographic APIs as discussed in Section 2.2. Sensitive data is passed as a plaintext argument to encryption functions such as `MD5()` within the SQL statement, for example `MD5('thisissecret')`. As the SQL command is registered in the DBMS view, it becomes passively available to the attacker's loop.
- If the password were instead hashed at the `backend` without using the SQL-provided API, the database would receive the SQL query with already-hashed data from the outset, and no plaintext would ever appear in the view. We argue this is the most practical mitigation against SQLSnoop. We discuss mitigation issues in more detail in Section 6.
- The SQLSnoop loop running inside the DBMS detects the victim's query in the PROCESSLIST view, applies a filter to exclude its own entries, and captures the plaintext credential before the hash function executes. Once the loop reaches the configured iteration limit N, execution completes and the query result is returned to the `backend` (arrow ⑤).
- Finally, the result is sent as part of an HTTP response to the `attack` container. Upon a successful attack, the SQL statement invoked by the user—including the sensitive plaintext credential —is exposed to the attacker, while the database stores only the resulting hash value.

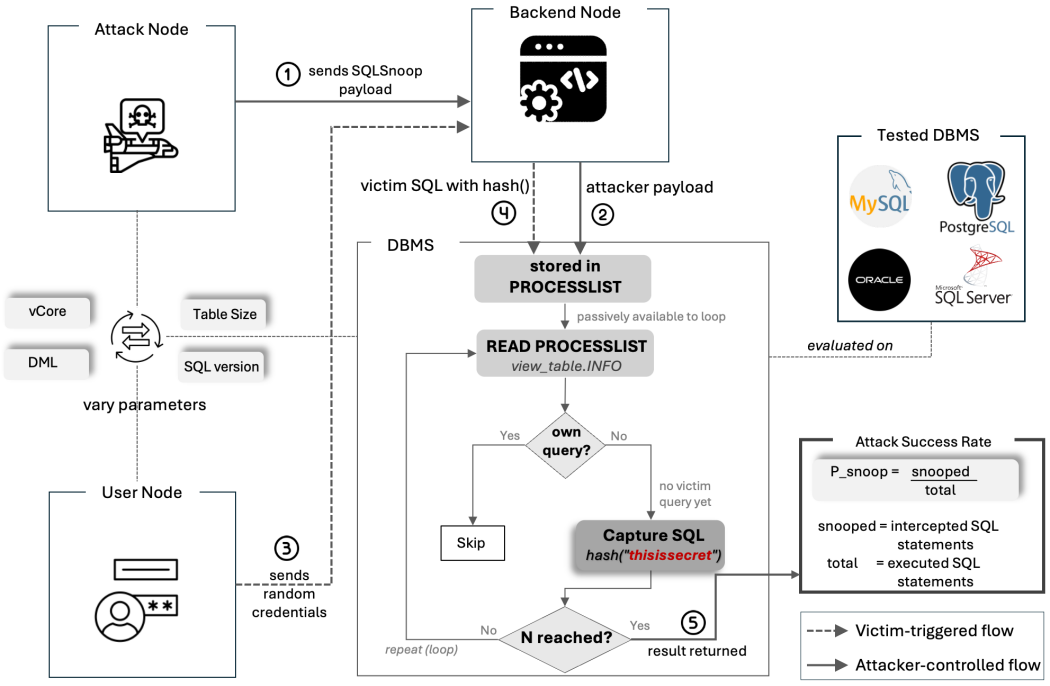

Figure 4. Experiment System Design Diagram. The experiment was structured with four Docker containers, each simulating a physical machine in a real-life scenario. Solid arrows indicate attacker-controlled flows; dashed arrows indicate victim-triggered flows.

To summarize the experiment system design, we defined attacker and victim actions and deployed multiple Docker-based servers and clients to simulate the vulnerable system,

the attacker, and the victim. The two flows operate concurrently and independently: the attacker's payload runs a continuous polling loop inside the DBMS while the victim submits credentials through the backend in the normal course of a sign-up or login action. We also applied various evaluation parameters—including vCore allocation, MySQL version, DML operation type, table size, and SQL statement length—to precisely measure the attack success rate under multiple environmental configurations. We use the term *payload* throughout this paper to refer to the SQLSnoop SQL query that performs the interception.

4. Implementation

4.1. Infrastructure

As described in Section 3.3, multiple Docker container nodes are used to build the simulated system. All nodes are deployed as virtual instances on a single physical machine. The system specifications are summarized in Table 2.

Table 2. Hardware Specifications of the Evaluation Workstation. The operating system is Ubuntu 22.04 Server.

Component	Specification
CPU	AMD Ryzen 9 5900X 12-Core Processor
Number of vCores	24
Base Clock	3.7 GHz
RAM	4 × (32 GB 3200 MHz PC4-25600 DDR4)
SSD	2 TB PCIe Gen4 NVMe

4.2. Crafting SQLSnoop Attack Payload

Our SQLSnoop query has been carefully designed to iterate through the selection of SQL statements currently in execution within the DBMS view. While executing this statement, it is important to note that the SQLSnoop statement itself is also momentarily present in the DBMS view. Therefore, a `WHERE` clause is specifically constructed and added to the statement to exclude the SQLSnoop statement itself (co-existing inside the view alongside the victim statement). Subsequently, the snooped SQL statements are concatenated into a result set, which will later be transmitted back to the attacker. A notable challenge persists: determining how to effectively run a loop-based query via an SQL Injection vulnerability.

In `INSERT`, `SELECT`, `UPDATE`, `DELETE` statements, DBMSs do not inherently offer features designed for the repetition of SQL commands, which can be leveraged in SQL Injection. Nevertheless, certain features, originally not intended for SQL command repetition, can be ingeniously integrated into an SQL Injection payload to facilitate the iterative access of the DBMS view. For example, in MySQL, the `benchmark()` function can be utilized to iterate snoop attempts. We provide four Proof-of-Concept code snippets for the SQLSnoop attack payload for MySQL, MSSQL, PostgreSQL, and Oracle in Listings 1–4.

All Proof-of-Concept SQLSnoop attack payloads have a similar structure but different details/syntax for each DBMS. For example, while `benchmark()` is utilized to loop the snooping query, `replicate()` is utilized as an alternative with a slightly different syntax structure. In all cases, all Proof-of-Concept code shows that SQLSnoop requires (i) loop implementation, (ii) string concatenation, and (iii) filtering logic to exclude the attack statement itself.

Listing 1 demonstrates the MySQL implementation, where `BENCHMARK()` is repurposed to drive the snooping loop. The `GROUP_CONCAT()` function accumulates captured SQL statements, and a `WHERE` clause with a unique string filters out the SQLSnoop query itself.

Listing 1: Simplified Proof-of-Concept SQLSnoop code snippet against `MySQL`.

```
SELECT @Result = ""
UNION
SELECT Benchmark(100000,
-- Built-in functions for implementing loop
  (@TempResult :=
    (SELECT GROUP_CONCAT(`INFO`) FROM `information_schema`.`PROCESSLIST`
    -- Bring SQLs from view except SQLSnoop related ones
      WHERE `INFO` NOT LIKE "%UNIQUE_STRING%"
      -- Filter out SQLSnoop related queries
      OR SLEEP(0)
      -- Prevent caching query results
    )
  )
  ^ IF(
    (@TempResult != 0x00)
    -- Check if there are snooped SQLs
    && (@Result NOT LIKE CONCAT("%", @TempResult, "%")),
    -- Filter out duplicates
    @Result := CONCAT(@Result, @TempResult),
    -- Concatenate snooped results
    0
  )
)
UNION
SELECT @Result
```

Listing 2 shows the MSSQL variant, which uses `REPLICATE()` to implement the loop. Unlike MySQL, MSSQL requires `CROSS APPLY` with `sys.DM_EXEC_SQL_TEXT()` to retrieve the full SQL text from an execution handle.

Listing 2: Simplified Proof-of-Concept SQLSnoop code snippet against `MSSQL`.

```
SELECT REPLICATE(
-- Built-in functions for implementing loop
  (SELECT `sqltext`.`text` FROM `sys`.`dm_exec_requests`
  -- Get SQL handle for view table except SQLSnoop related ones
    CROSS APPLY `sys`.DM_EXEC_SQL_TEXT(sql_handle) AS sqltext
    -- Read SQL query from handle
    WHERE `text` NOT LIKE '%UNIQUE_STRING%'
    -- Filter out SQLSnoop related queries
  ), 100000
)
```

Listing 3 presents the PostgreSQL implementation, where `GENERATE_SERIES()` provides the loop mechanism. The `pg_stat_activity` view is queried directly, and `PG_BACKEND_PID()` is used to exclude the attacker's own session.

Listing 4 covers the Oracle implementation, which uses the `CONNECT BY LEVEL` clause to construct the loop. Oracle's `V$SQL` view is queried via `LISTAGG()` to concatenate the captured SQL statements.

Listing 3: Simplified Proof-of-Concept SQLSnoop code snippet against `PostgreSQL`.

```
SELECT ""
UNION
SELECT(
  SELECT ARRAY_TO_STRING(ARRAY(
  -- Concatenate the result
    SELECT `query` FROM `pg_stat_activity`
    -- Read SQL in view table except SQLSnoop related ones
    WHERE `pid` != PG_BACKEND_PID()
    -- Filter out SQLSnoop queries from results
    AND `query` != "<insufficient privilege>")
    -- Filter out insufficient privileged queries
  , '::')
  )
FROM GENERATE_SERIES(1, 100000)
-- Built-in function used to implement loop
```

Listing 4: Simplified Proof-of-Concept SQLSnoop code snippet against `Oracle`.

```
SELECT CONCAT('', (SELECT LISTAGG(`SQL_FULLTEXT`, '::')
-- Concatenate the result
  FROM `v$sql` WHERE 1 = 1)
  -- Read SQL from view table except SQLSnoop related ones
) FROM `dual`
WHERE LEVEL > 1 CONNECT BY LEVEL <= 100000
-- Built-in clause used to implement loop
```

4.3. Nodes

In this subsection, we delve into the implementation details of the Docker container-based nodes and explain their detailed usage for the evaluation.

4.3.1. Attack Node

The `attack` node is mainly composed of a Python 3.10 script, taking the DBMS type and the desired number of iterations as arguments. The script is responsible for dispatching the SQLSnoop payload to the `backend` server using the Python `requests` module. Subsequently, it awaits the server's response and analyzes the returned results to determine the number of successfully snooped SQL statements.

4.3.2. User Node

Similarly, the `user` node functions by executing a Python script that repeatedly sends randomly generated user information to the `backend` server. The script accepts the number of user information entries to be sent to the server as an argument. These randomly generated user details are transmitted to the backend server in the form of a `JSON` payload via Python requests.

4.3.3. Backend Node

The `backend` node plays a dual role within our test system. First, it serves as a Spring Boot backend responsible for handling user requests. Depending on the RESTful API invoked by the user container, the backend uses the user information to generate and execute SQL commands for actions such as insertion, selection, updating, and deletion in the database. Second, the backend node provides an intentional backdoor as a simulation of the SQL Injection vulnerability provided to the attacker to execute the malicious SQLSnoop

payload. In both cases, the backend container establishes a connection with the database using the `JDBC` protocol.

4.3.4. Database Node

For the `database` node, we simply utilize the Docker container images from the official `Docker Hub` repository [19]. We do not modify these database images in any way so that we can prove that our attack works against real-world server environments.

5. Evaluation

To evaluate the extent to which SQLSnoop is practical, we designed a simulated web-application attack environment based on MySQL, one of the most widely deployed DBMS platforms in the market. We also tested if the SQLSnoop technique can intercept SQL statements for the most prevalent `insert`, `select`, `update`, and `delete` (i.e., Data Manipulation Language operations).

5.1. Attack Success Rate

To rigorously quantify the SQLSnoop threat, we introduce the attack success rate P_{snoop}, defined as:

$$P_{\text{snoop}} = \frac{N_{\text{snooped}}}{N_{\text{executed}}} \tag{2}$$

where N_{snooped} denotes the number of successfully intercepted SQL statements, and N_{executed} denotes the total number of SQL statements executed during the attack.

Since most commodity servers impose time limits on SQL execution, measuring success rates over millions of statements is impractical. We therefore used a sample size of 1000 SQL statements per experiment, large enough to yield meaningful statistics while remaining within server time constraints. We also carefully adjusted the number of SQLSnoop iterations to ensure the payload remained active throughout the entire execution window of all targeted statements.

We conducted 30 repetitions per experiment to calculate standard deviations. The following subsections investigate the factors affecting P_{snoop} within a MySQL environment.

5.2. Attack Success Rate Depending on MySQL Version

A critical aspect of assessing the threat level posed by SQLSnoop is determining whether its success is contingent on the specific version of the database. To comprehensively analyze the influence of the database version on the attack success rate, we conducted experiments in which every other variable was controlled, including the number of `vCores` (virtual cores) allocated to the container.

In these experiments, our focus was on SQL `insert` statements comprising two attributes: username and password, each consisting of 10 lowercase letters (a–z).

Although no significant differences were observed between the various database versions, it is worth noting that the standard deviation decreased in all versions as the container was assigned more vCores, simultaneously achieving a higher success rate. The standard deviation reached `zero` when the success rate reached `1.0`, indicating a highly stable and predictable performance under these conditions. Figure 5 presents the attack success rate across MySQL versions under three vCore configurations.

Our analysis covered three different vCore settings: `0.5`, `1.0`, and `1.5` vCores allocated to the database container. The results showed that the success rate remained exceedingly high across all versions of the database, consistently exceeding `0.9`. Remarkably, this high success rate persisted even when the container had only `0.5` vCores and reached a perfect score of `1.0` in all versions when the container was allocated `1.5` vCores.

The MySQL versions selected for this evaluation span several major release cycles, covering versions 5.5, 5.6, 5.7 and 8.0. These versions represent a wide deployment range in real-world production environments, from legacy systems still running on 5.6 to modern deployments on 8.0 and beyond. By testing across this range, we ensure that the results reflect the vulnerability as it exists across the actual installed base of MySQL servers, rather than a single controlled version. The `INFORMATION_SCHEMA.PROCESSLIST` view, which SQLSnoop relies upon, has been present and accessible in all these versions without structural changes, making version-independent exploitability a reasonable hypothesis to evaluate.

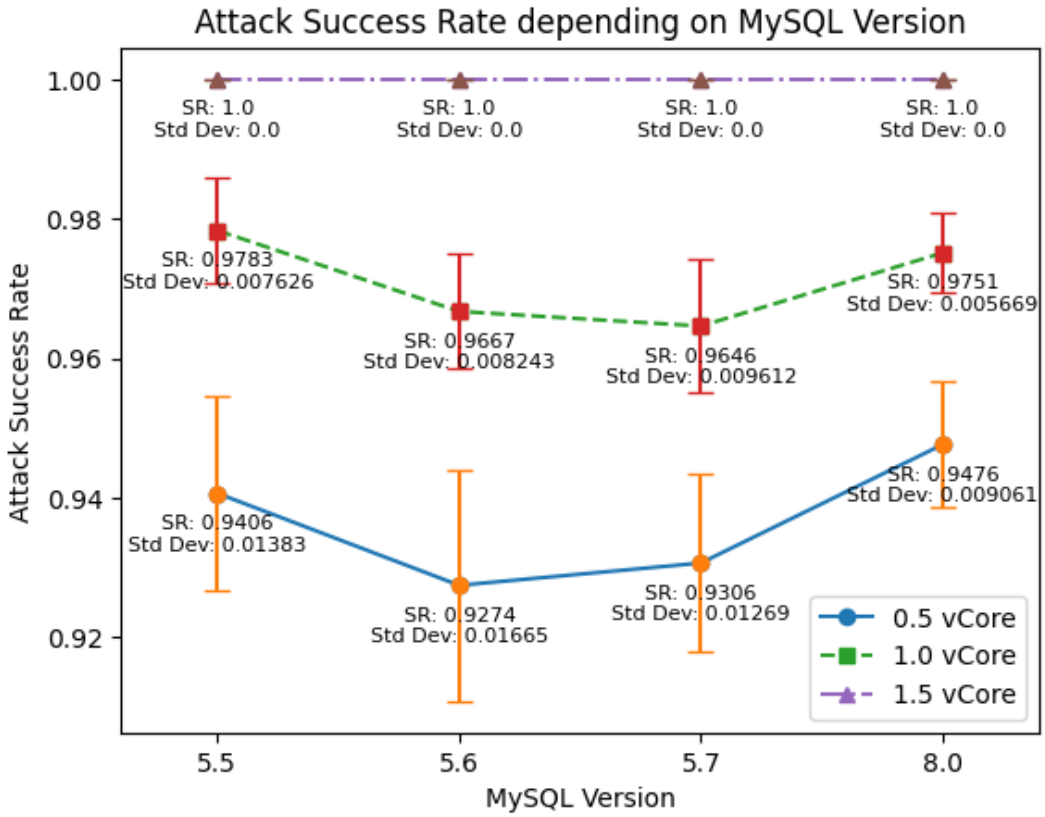

Figure 5. Attack Success Rate Depending on MySQL Version. The success rate was exceptionally high across all versions, consistently exceeding 0.9, even when the container had only 0.5 vCores at its disposal.

Understanding whether SQLSnoop's effectiveness varies across database versions is particularly important from a threat modeling perspective. A version-dependent attack would allow administrators to mitigate the threat simply by upgrading or downgrading their DBMS. If, however, the attack succeeds consistently across all tested versions, it implies that the underlying vulnerability is architectural rather than implementation-specific, and it cannot be resolved through routine software updates alone. This distinction has direct implications for the urgency and nature of any recommended countermeasures.

5.3. Attack Success Rate Depending on the Number of vCores Allocated to the Database Container

To explore the relationship between the success rate and the number of vCores allocated to the database container, we conducted experiments using MySQL version 8.0. We targeted the same SQL insert statement used in Section 5.2, but this time, we examined its performance under varying vCore settings.

The number of vCores allocated to a Docker container directly controls how much CPU time the containerized process can consume. When a database container operates

under tight CPU constraints, the DBMS scheduler must time-share the available processing capacity across competing tasks—including both the SQLSnoop polling loop and the victim's incoming SQL statements. This scheduling contention creates irregular timing gaps between consecutive view accesses, which in turn reduces the probability that any given victim SQL statement will be observed during its brief residence in the PROCESSLIST view. Conversely, when sufficient CPU resources are available, the SQLSnoop loop can execute continuously without interruption, maximizing the frequency of view polling and thus the likelihood of interception.

In our evaluation, we varied the vCore allocation from `0.1` to `2.0` vCores, covering a range that spans severely resource-constrained deployments to moderately provisioned production-like environments. This range was chosen to identify the minimum resource threshold at which the attack achieves perfect reliability, as well as to characterize attack behavior under the kind of limited resource conditions common in containerized microservice deployments.

Our findings revealed a clear pattern: as the number of vCores allocated increased, the attack success rate increased correspondingly, steadily climbing until it reached a perfect score of `1.0`. In particular, we observed that this threshold was reached when the container was allocated `1.2` vCores. Figure 6 illustrates the relationship between the number of allocated vCores and the attack success rate.

Notably, we also discovered that the attack success rate was unexpectedly high at `0.8594` when the container had only `0.1` vCore allocated. This observation suggests that SQLSnoop can perform remarkably well even when the database is operating with limited computational resources.

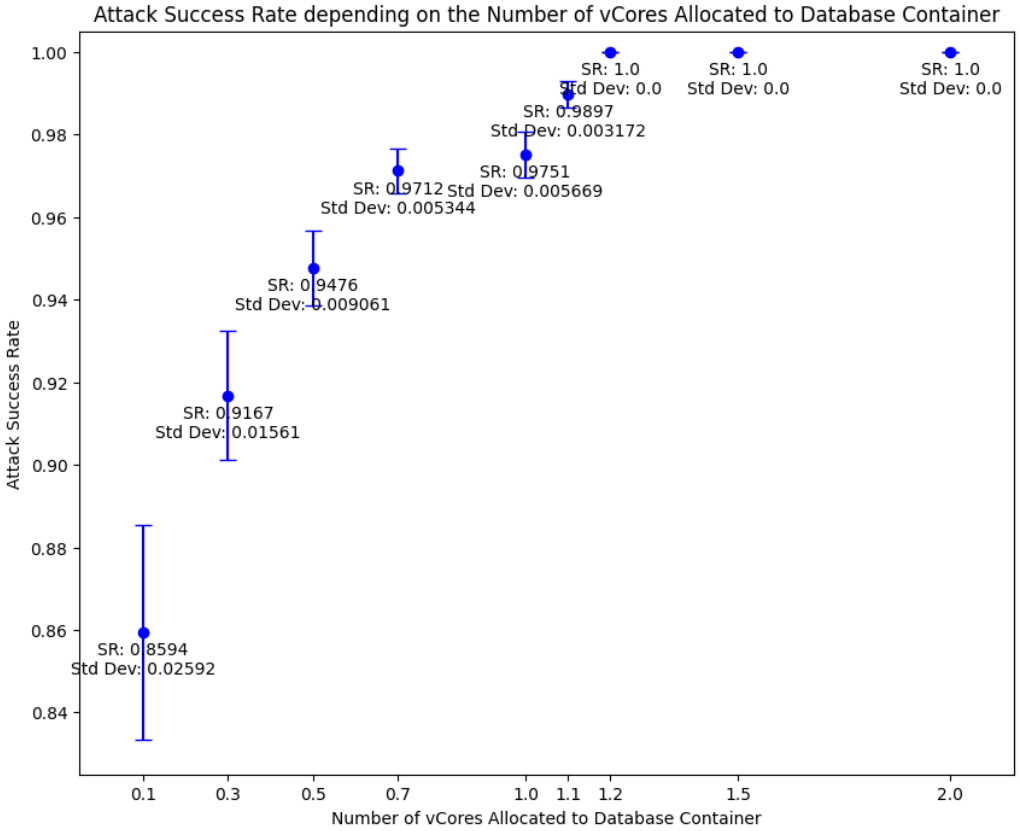

Figure 6. Attack Success Rate Depending on the Number of vCores Allocated to the Database Container. The success rate exhibited a positive correlation with the number of vCores allocated to the database container.

To gain a deeper understanding of the significance of this 1.2 threshold, we conducted an experiment in which we assigned 16 vCores to the database container and executed the SQLSnoop statement while continuously sending SQL insert requests to the database. This analysis revealed that CPU usage approached approximately 117% as shown in Table 3, indicating that up to 1.17 vCores, the container lacked sufficient CPU resources and had to depend on scheduling logic to process requests. This reliance occasionally led to increased time gaps between accessing the view table, resulting in a reduced attack success rate.

Table 3. Observed Resource Utilization of the MySQL Container During High-Load SQLSnoop Execution.

Metric	Value	Unit	Remarks
CPU Usage	117.22	%	Indicates CPU saturation beyond single core
Memory Usage	363.8/125.7	MiB/GiB	Very low relative usage
Memory Utilization	0.28	%	Negligible memory pressure

This phenomenon further clarifies why the success rate remains constant at 1.0 when the container is allocated greater than or equal to 1.2 vCores. In such scenarios, the view is accessed consistently without interruption, allowing for the successful interception of all executed SQL commands.

The observed trend in the standard deviation can also be explained by taking into account the role of DBMS scheduling. As the number of vCores allocated to the database container increased, not only did the attack success rate improve, but there was also a general decrease in the standard deviation. However, there was one notable exception when vCores increased from 0.7 to 1.0, resulting in a slight increase in the standard deviation.

The overall decline in the standard deviation can be attributed to the reduced dependence of the container on scheduling logic, which, in turn, introduces instability in our SQLSnoop attack. When the database relies more heavily on scheduling, it allows a series of user-invoked SQL commands to be executed consecutively without any opportunities for SQLSnoop attempts to be made. This dynamic situation contributes to the observed pattern in the standard deviation between different vCore settings.

CPU Utilization and Stealth Considerations. The observed CPU usage of approximately 117% during active SQLSnoop execution raises a practical consideration for real-world deployments: sustained CPU saturation at this level may trigger anomaly-based monitoring alarms in enterprise environments with database activity monitoring (DAM) systems. This represents an inherent trade-off between attack reliability and stealth. At lower vCore allocations (e.g., below 0.5 vCores), CPU saturation is reduced and the attack becomes less conspicuous, though success rates decline accordingly as shown in Figure 6. Conversely, at 1.2 or more vCores, success rates reach 1.0 but at the cost of higher CPU visibility. An attacker seeking stealth may therefore calibrate the iteration count of the SQLSnoop payload to operate below the saturation threshold, accepting a lower success rate in exchange for reduced detectability. This trade-off between attack intensity and evasion is an important direction for future work on SQLSnoop countermeasures.

5.4. Attack Success Rate Depending on the Execution Time of Statements

As discussed in Section 3.1, it was established that an increase in the execution time of SQL commands makes them more susceptible to SQLSnoop attacks. To investigate this phenomenon, we designed an experiment using SQL select statements. Specifically, we focused on the `select` SQL queries executed on tables of different sizes, as the execution time of such queries varies significantly with the table size.

In this experiment, conducted with MySQL version 8.0, we allocated 2 vCores to the database. The user table, which we used as the basis for this analysis, contained entities ranging from 1 to 10^7.

The results of our experiment were striking. Even with just a single entity in the table, where the SQL `select` command executes almost instantaneously, it was consistently intercepted by SQLSnoop, highlighting the attack's remarkable efficiency. As expected, the attack success rate remained at a perfect `1.0`, with a standard deviation of `zero`, as the number of entities increased in the table. This result is consistent with the 1.2 vCore saturation threshold established in Section 5.2: with 2 vCores allocated, the view table is accessed without interruption regardless of the execution time of the statement, resulting in $P_{\text{snoop}} = 1.0$ in all tables tested.

5.5. Attack Success Rate Depending on the Length of SQL Statements

In this experiment, our objective was to explore both factors discussed in Section 3.1, specifically, the execution time of the SQL statements and the period of SQLSnoop attempts. We utilized MySQL version 8.0 and allocated 2 vCores to the database, as in previous experiments. However, this time, we used an insert statement with three columns: username, password, and blob. For experimental purposes, the `blob` column was long-text-type, and we examined how the size of the blob affected the success rate. The usernames and passwords each consisted of 10 lowercase alphabet letters as before.

In this scenario, as the size of the blob increases, the time required to insert data into the database also increases. Furthermore, due to our implementation of the SQLSnoop attack, in which snoop queries are repeated and snooped results are concatenated, there was a noticeable increase in the time gap between accessing the DBMS view as more snooped queries accumulated in the results. Figure 7 shows the attack success rate as a function of blob size for three insert count settings.

It is crucial to distinguish between two key factors at play here: First, the execution time of the `insert` command is primarily influenced by the size of the blob. Second, the execution time of the SQLSnoop command is influenced by both the blob size and the number of snooped SQL statements. To account for this distinction, we conducted our experiment with varying numbers of insert commands: `10`, `100`, and `1000`. This allowed us to comprehensively analyze the success rate under the impact of both blob size and the accumulation of snooped SQL statements.

When analyzing the effects of snooping on a limited number of insert statements (`10`), we observed that the size of the blob had minimal impact on the attack success rate, which consistently remained at a perfect `1.0` with a standard deviation of `zero`. However, as we increased the number of insert statements to `100` and `1000`, a decline in the success rate became apparent.

With `100` insert statements, a slight reduction from `1.0` to `0.9933` was observed when the blob size increased from `1000` to `10,000` bytes. However, with `1000` `insert` statements, a more pronounced decrease occurred, with the success rate dropping from `1.0` to `0.9376` as the blob size increased from `100` bytes to `1,000` bytes. Furthermore, a significant drop occurred, starting from `0.9933` to `0.7013` as the blob size increased from `10,000` to `100,000` bytes with `100` inserts, and from `0.9376` to `0.4274` as the blob size increased from `1000` bytes to `10,000` bytes with `1000` inserts.

These observations align closely with our initial hypothesis: as the results became increasingly saturated with large SQL statements, the time gap between accessing the DBMS view grew substantially larger than the lifespan of SQL statements on the view. This phenomenon ultimately led to a notable decrease in the success rate of the SQLSnoop attack.

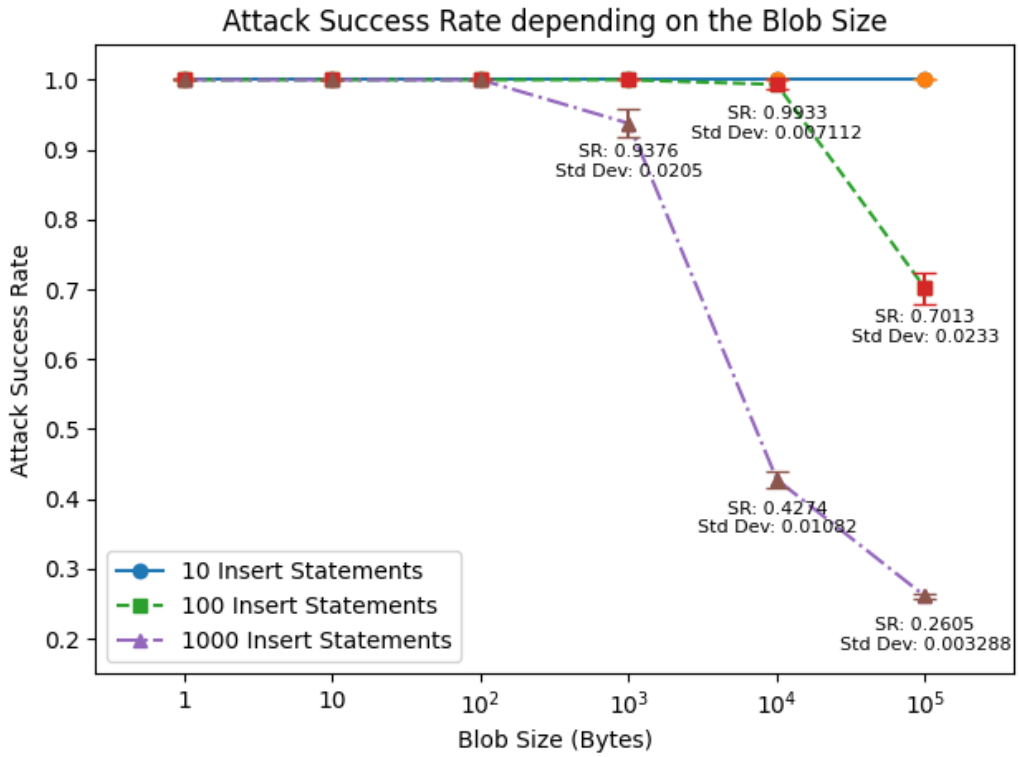

Figure 7. Attack Success Rate Depending on the Blob Size. The primary factor influencing the success rate was the growing time required to process the concatenation of stacked results as the blob size increased.

In contrast to our previous experiments, we observed that the decline in the success rate was not accompanied by an increase in the standard deviation. This observation aligns with our earlier findings, since the increase in the blob size does not compel the database to rely more on scheduling logic when it has an adequate number of vCores.

The results of this experiment also imply that to optimize the success rate of an SQLSnoop attack, the attacker should carefully calibrate the number of iterations of the SQLSnoop attempts. This adjustment is crucial to prevent the results from becoming excessively stacked with SQL statements, which can, in turn, diminish the success rate of the attack. Balancing the number of iterations is a key strategic consideration to achieve the highest possible success rate in SQLSnoop attacks.

6. Limitations

Limited SQL Injection. SQLSnoop assumes the presence of an attacker who has already obtained SQL Injection access. Therefore, if there is no SQL Injection vulnerability, SQLSnoop is not a threat to DBMS systems. However, not all SQL Injection vulnerabilities permit full SQL command execution. Some vulnerabilities are powerful enough to yield complete SQL execution capability to the attacker, while others restrict the attacker to only limited SQL operations. Since SQLSnoop involves the execution of loops and advanced SQL syntax features, SQL vulnerabilities that give limited capabilities to attackers can render the SQLSnoop attack infeasible. For example, if the attacker could not use subquery syntax, the SQLSnoop attack becomes infeasible because of the lack of a nested `select` statement. Nevertheless, a significant volume of legacy web applications built prior to the widespread adoption of modern defensive practices remains in active deployment. Such systems are often difficult to update or replace due to operational and financial constraints, and SQL Injection vulnerabilities in these codebases continue to represent a

real and persistent threat [1]. For these systems, SQLSnoop represents an additional and previously uncharacterized risk that warrants consideration during security assessments.

NoSQL. Unlike the conventional RDBMS, NoSQL does not support a table-based schema interface consisting of rows and columns. However, this does not guarantee that NoSQL databases are totally immune to SQLSnoop attacks in the sense that a similar attack vector might exist in another form. If any accessible information schema contains confidential query data, that data will be exposed to SQLSnoop-style attacks.

SQLite. While SQLite supports an in-memory schema view table (e.g., `sqlite_master`), it does not process running SQL statements in default configuration. Therefore, it is not possible to apply SQLSnoop against SQLite in its current form. However, similarly to NoSQL, any alteration to the system that exposes sensitive data accessible to the SQL interface will be affected by the SQLSnoop attack.

Experimental Scope and Quantitative Evaluation. The systematic quantitative evaluation presented in Section 5 is based exclusively on MySQL. While we provide working Proof-of-Concept implementations for MSSQL, Oracle, and PostgreSQL, demonstrating that the underlying vulnerability exists in each platform's monitoring view, rigorous statistical measurement of attack success rates under varying resource and workload conditions was conducted only on MySQL as the representative platform. The architectural vulnerability—plaintext SQL statement registration in in-memory views prior to hash function execution—is common to all four platforms by design, and we expect similar success rate characteristics; however, quantitative characterization of the other three platforms remains a direction for future work.

Synchronization Assumption and Upper-Bound Conditions. The experiments reported in Section 5 were conducted under conditions where the SQLSnoop payload was pre-loaded and actively polling before victim requests arrived, and the iteration count was configured to cover the full execution window of targeted statements. These conditions represent an upper bound on attack effectiveness. In practice, an attacker may not know in advance when victims will issue SQL statements, leading to scenarios where the SQLSnoop payload is activated asynchronously relative to victim activity. Under such conditions, the success rate would depend on the overlap probability between the snooping window and victim statement lifetime, as formalized in Section 3.1. Evaluating success rates under asynchronous and unpredictable victim timing is an important avenue for future experimental work.

Oracle Privilege Requirements. As discussed in Section 3.2, accessing `V$SQL` in Oracle requires the `SELECT_CATALOG_ROLE` privilege, which is not granted to ordinary database users by default. While this role is commonly held by application accounts and monitoring users in many enterprise deployments, environments strictly adhering to the principle of least privilege may restrict access to this view, thereby limiting the applicability of SQLSnoop in Oracle deployments. The attack is most effective against Oracle when the attacker's injected code executes under a session that already holds `SELECT_CATALOG_ROLE`.

Cloud and Distributed Databases. This work focuses on traditional on-premises RDBMS deployments. Cloud-managed database services (such as Amazon RDS, Google Cloud SQL, and Azure SQL Database) and distributed database systems (such as CockroachDB, TiDB, and Google Spanner) introduce architectural differences that may affect the accessibility of in-memory monitoring views. In managed cloud services, administrative views may be restricted or abstracted away from the SQL interface, potentially preventing SQLSnoop-style access. Evaluating SQLSnoop against cloud-native and distributed database architectures is an important direction for future work.

7. Related Works

DBMS Encryption. Various previous works have attempted to protect the database system from information leakage through the encryption technique [20–25]. Furthermore, to apply encryption while data is being processed inside memory, the idea of homomorphic encryption has also been introduced [26–28]. However, a major drawback of homomorphic encryption is the excessive consumption of system resources [26,27]. Some research has attempted to overcome this by encrypting only parts of the database queries [25,29,30]. These previous works apply encryption to the database system; however, their attack model and goal are different from our work. For example, *CryptDB* [25] applied encryption during in-memory data processing of DBMS and was later extended as part of the Relational Cloud database-as-a-service system [31]; however, the design goal of such schemes is to protect data against the database management administrator (DBA) rather than against SQL Injection attackers. Therefore, designing the SQLSnoop defense system can be more effective because the threat model is different. In theory, fully encrypted processing of DBMS data [32,33] will mitigate SQLSnoop; however, prior work has demonstrated that such approaches incur severe performance costs [34,35]. Currently, the most practical way to prevent SQLSnoop is using hashing and encryption [36] at the application layer and not using the SQL-provided API. Unfortunately, enforcing this programming practice across existing codebases is not straightforward.

SQL Injection Defense in General. To defend against SQL Injection attacks, the most traditional way is to sanitize user-supplied data. Such sanitation should consider various issues such as the data type and encoding of meta-characters, as discussed in [37]. In addition, using prepared statements with parameterized queries and stored procedures is a good way to defend against SQL Injection as introduced in [4]. Some research goes beyond these defense methods, proposing techniques such as tokenizing input queries to detect SQL Injection [9,38–42] or employing runtime monitoring [43,44]. Despite these efforts, SQL Injection remains an ongoing issue, with recent studies continuing to propose new detection and prevention approaches including machine-learning-based methods [8,9,45]. While this prior literature focuses on eradicating SQL Injection vulnerability [46–48], our work proposes a new attack vector using SQL Injection as a prerequisite. It is important to note how existing defenses relate to the SQLSnoop attack chain. Input sanitization and parameterized queries [4,37] block SQL Injection entirely and therefore prevent SQLSnoop at its prerequisite stage. Runtime monitoring and anomaly detection [43,44] may detect the looping behavior of the SQLSnoop payload, but only after the attacker has already gained SQL Injection access. Application-layer hashing—i.e., computing the hash before the SQL statement is constructed—directly eliminates the plaintext exposure window that SQLSnoop exploits, regardless of whether SQL Injection is present. Full database encryption [32,33] would in theory prevent plaintext exposure in monitoring views, but it incurs severe performance costs as noted above. This analysis clarifies that while existing SQL Injection defenses can prevent SQLSnoop indirectly by eliminating its prerequisite, application-layer hashing is the only defense that specifically addresses the plaintext exposure window that SQLSnoop exploits.

SQL Injection Issues in Other Fields. SQL Injection issues are not limited to web servers alone. The vulnerability can also occur in various environments, including the Web SQL Database API in web browsers [49], SQLite in mobile systems [50], various embedded systems [51], and various NoSQL databases [52–54], including Redis. Notably, Internet of Things (IoT) systems are also often threatened by SQL Injection attacks [55], and numerous research efforts have been made to prevent them [56–60]. Mitigating SQL Injection is one way to address SQLSnoop attacks; however, as previous works suggest, mitigating

SQL Injection is not an easy problem, and thus further research specifically addressing SQLSnoop is needed.

8. Conclusions

In this paper, we introduced *SQLSnoop*, a technique that exploits SQL Injection to bypass database-side cryptographic protections. We demonstrated that using SQL-provided hashing and encryption APIs are insufficient to protect sensitive data when an SQL Injection vulnerability is present. SQLSnoop exploits the in-memory view table within modern Relational Database Management Systems (RDBMSs) that temporarily store SQL statements during their execution. Using the SQLSnoop technique, an attacker can circumvent the cryptographic protections provided by the SQL built-in hash functions.

In our experiments, we demonstrated SQLSnoop against four major DBMS platforms: `MySQL`, `MSSQL`, `Oracle`, and `PostgreSQL`. Systematic quantitative evaluation was conducted on MySQL, where SQLSnoop achieved attack success rates consistently above 90%, reaching 100% at 1.2 or more virtual CPU cores, across multiple versions and all four DML operations. Feasibility on MSSQL, Oracle, and PostgreSQL was confirmed through working Proof-of-Concept implementations, each exploiting the platform's native in-memory monitoring view. This makes SQLSnoop a credible and practical threat in real-world web applications. This work provides actionable guidance for web application developers seeking to protect their systems against this class of attack.

Author Contributions: Conceptualization, D.J. (Dowon Jeong) and D.J. (Daehee Jang); methodology, D.J. (Daehee Jang); software, D.J. (Dowon Jeong) and J.K.; validation, D.J. (Dowon Jeong), J.K. and D.J. (Daehee Jang); formal analysis, D.J. (Daehee Jang); investigation, D.J. (Dowon Jeong) and J.K.; writing—original draft preparation, D.J. (Dowon Jeong) and J.K.; writing—review and editing, A.F. and D.J. (Daehee Jang); visualization, A.F.; supervision, D.J. (Daehee Jang). All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the Convergence Security Core Talent Training Business Support Program (IITP-2026-RS-2023-00266615) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation), and by the Korea Research Institute for Defense Technology Planning and Advancement (KRIT) grant funded by the Korean government (Defense Acquisition Program Administration) (KRIT-CT-24-001, Defense Space Security Research Lab, 2025). This work was also supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2026-25489421).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in this study are included in the article. The experimental results were generated through controlled, reproducible experiments using publicly available DBMS software (Docker Hub images) and configurations fully described within the manuscript. No proprietary or externally sourced datasets were used. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: Author Dowon Jeong was employed by the company NAVER Corp. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DBMS	Database Management System
RDBMS	Relational Database Management System

SQL	Structured Query Language
OWASP	Open Web Application Security Project
DBA	Database Administrator
IoT	Internet of Things
JDBC	Java Database Connectivity
API	Application Programming Interface
PoC	Proof of Concept

References

1. OWASP, T.O.W.A.S.P. OWASP Top Ten. Available online: https://owasp.org/www-project-top-ten/ (accessed on 9 September 2023).
2. OWASP Cheat Sheet Series. SQL Injection Prevention Cheat Sheet. Available online: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (accessed on 9 September 2023).
3. Django Software Foundation. Django Documentation: SQL Injection. Available online: https://docs.djangoproject.com/en/4.2/topics/security/#:~:text=SQL%20injection%20is%20a%20type,are%20constructed%20using%20query%20parameterization (accessed on 9 September 2023).
4. Kumar, B.S.; Anaswara, P. Vulnerability detection and prevention of SQL injection. *Int. J. Eng. Technol.* **2018**, *7*, 16. [CrossRef]
5. Ping, C. A second-order SQL injection detection method. In *Proceedings of the 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*; IEEE: New York, NY, USA, 2017; pp. 1792–1796.
6. Spett, K. *Blind SQL Injection*; Technical Report; SPI Dynamics: Atlanta, GA, USA, 2003.
7. Nasereddin, M.; ALKhamaiseh, A.; Qasaimeh, M.; Al-Qassas, R. A systematic review of detection and prevention techniques of SQL injection attacks. *Inf. Secur. J. A Glob. Perspect.* **2023**, *32*, 252–265. [CrossRef]
8. Alghawazi, M.; Alghazzawi, D.; Alarifi, S. Detection of SQL Injection Attack Using Machine Learning Techniques: A Systematic Literature Review. *J. Cybersecur. Priv.* **2022**, *2*, 764–777. [CrossRef]
9. Joshi, N.; Sheth, T.; Shah, V.; Gupta, J.; Mujawar, S. A Detailed Evaluation of SQL Injection Attacks, Detection and Prevention Techniques. In *Proceedings of the 2022 5th International Conference on Advances in Science and Technology (ICAST)*; IEEE: New York, NY, USA, 2022; pp. 352–357. [CrossRef]
10. IBM. What Is a Database Schema? Available online: https://www.ibm.com/topics/database-schema (accessed on 17 March 2023).
11. Microsoft Corporation. SQL Server Views Documentation. Available online: https://learn.microsoft.com/en-us/sql/relational-databases/views/views?view=sql-server-ver16 (accessed on 9 September 2023).
12. MySQL Documentation: Information Schema. Available online: https://dev.mysql.com/doc/refman/8.0/en/information-schema-introduction.html (accessed on 9 September 2023).
13. Shmueli, E.; Vaisenberg, R.; Elovici, Y.; Glezer, C. Database encryption: An overview of contemporary challenges and design considerations. *ACM SIGMOD Rec.* **2010**, *38*, 29–34. [CrossRef]
14. MySQL Documentation: Information Schema PROCESSLIST Table. Available online: https://dev.mysql.com/doc/refman/8.0/en/information-schema-processlist-table.html (accessed on 9 September 2023).
15. Microsoft SQL Server Documentation: sys.dm_exec_requests (Transact-SQL). Available online: https://learn.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-exec-requests-transact-sql?view=sql-server-ver16 (accessed on 9 September 2023).
16. Microsoft SQL Server Documentation: sys.dm_exec_sql_text (Transact-SQL). Available online: https://learn.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-exec-sql-text-transact-sql?view=sql-server-ver16 (accessed on 9 September 2023).
17. Oracle Database 19c Documentation: V$SQL. Available online: https://docs.oracle.com/en/database/oracle/oracle-database/19/refrn/V-SQL.html#GUID-2B9340D7-4AA8-4894-94C0-D5990F67BE75 (accessed on 9 September 2023).
18. PostgreSQL Documentation: pg_stat_activity View. Available online: https://www.postgresql.org/docs/current/monitoring-stats.html#MONITORING-PG-STAT-ACTIVITY-VIEW (accessed on 9 September 2023).
19. Overview of Docker Hub. Available online: https://docs.docker.com/docker-hub/ (accessed on 17 October 2023).
20. Song, D.X.; Wagner, D.; Perrig, A. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy. S&P 2000*; IEEE: New York, NY, USA, 2000; pp. 44–55.
21. He, J.; Wang, M. Cryptography and relational database management systems. In *Proceedings of the 2001 International Database Engineering and Applications Symposium*; IEEE: New York, NY, USA, 2001; pp. 273–284.
22. Hacigümüş, H.; Iyer, B.; Li, C.; Mehrotra, S. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*; SIGMOD'02; Association for Computing Machinery: New York, NY, USA, 2002; pp. 216–227. [CrossRef]

23. Ozsoyoglu, G.; Singer, D.A.; Chung, S.S. Anti-tamper databases: Querying encrypted databases. In *Data and Applications Security XVII: Status and Prospects*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 133–146.
24. Bao, F.; Deng, R.H.; Ding, X.; Yang, Y. Private query on encrypted data in multi-user settings. In *Proceedings of the Information Security Practice and Experience* ; Springer: Berlin/Heidelberg, Germany, 2008; pp. 71–85.
25. Popa, R.A.; Redfield, C.M.; Zeldovich, N.; Balakrishnan, H. CryptDB: Protecting confidentiality with encrypted query processing. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, Cascais, Portugal, 23–26 October 2011; pp. 85–100.
26. Gentry, C. Computing arbitrary functions of encrypted data. *Commun. ACM* **2010**, *53*, 97–105. [CrossRef]
27. Gentry, C. Fully homomorphic encryption using ideal lattices. In Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing, Bethesda, MD, USA, 31 May–2 June 2009; pp. 169–178.
28. Gentry, C.; Sahai, A.; Waters, B. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Proceedings of the Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, 18–22 August 2013*; Proceedings, Part I; Springer: Berlin/Heidelberg, Germany, 2013; pp. 75–92.
29. Arasu, A.; Blanas, S.; Eguro, K.; Kaushik, R.; Kossmann, D.; Ramamurthy, R.; Venkatesan, R. Orthogonal Security With Cipherbase. In Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR'13), Asilomar, CA, USA, 6–9 January 2013.
30. Naveed, M.; Kamara, S.; Wright, C.V. Inference Attacks on Property-Preserving Encrypted Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*; CCS '15; Association for Computing Machinery: New York, NY, USA, 2015; pp. 644–655. [CrossRef]
31. Curino, C.; Jones, E.; Popa, R.; Malviya, N.; Wu, E.; Madden, S.; Balakrishnan, H.; Zeldovich, N. Relational Cloud: A Database-as-a-Service for the Cloud. In Proceedings of the 5th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, 9–12 January 2011; pp. 235–240.
32. Singh, P.; Kaur, K. Database security using encryption. In Proceedings of the 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), Greater Noida, India, 25–27 February 2015; pp. 353–358. [CrossRef]
33. Priebe, C.; Vaswani, K.; Costa, M. EnclaveDB: A secure database using SGX. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*; IEEE: New York, NY, USA, 2018; pp. 264–278.
34. Zhou, X.; Liu, J. A novel efficient database encryption scheme. In Proceedings of the 2012 9th International Conference on Fuzzy Systems and Knowledge Discovery, Chongqing, China, 29–31 May 2012; pp. 1610–1614. [CrossRef]
35. Iyer, B.; Mehrotra, S.; Mykletun, E.; Tsudik, G.; Wu, Y. A framework for efficient storage security in RDBMS. In *Proceedings of the Advances in Database Technology-EDBT 2004: 9th International Conference on Extending Database Technology, Heraklion, Greece, 14–18 March 2004*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 147–164.
36. Ah Kioon, M.C.; Wang, Z.S.; Deb Das, S. Security analysis of MD5 algorithm in password storage. *Appl. Mech. Mater.* **2013**, *347*, 2706–2711. [CrossRef]
37. Halfond, W.G.; Viegas, J.; Orso, A. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*; IEEE: New York, NY, USA, 2006; Volume 1, pp. 13–15.
38. Wei, K.; Muthuprasanna, M.; Kothari, S. Preventing SQL injection attacks in stored procedures. In *Proceedings of the Australian Software Engineering Conference (ASWEC'06)*; IEEE: New York, NY, USA, 2006; p. 8
39. Hlaing, Z.C.S.S.; Khaing, M. A detection and prevention technique on sql injection attacks. In *Proceedings of the 2020 IEEE Conference on Computer Applications (ICCA)*; IEEE: New York, NY, USA, 2020; pp. 1–6.
40. Hasan, M.M.; Bhuiyan, M.M.R. SQL Injection Attacks Prevention Using Machine Learning. *Int. J. Comput. Appl.* **2022**, *184*, 1–7.
41. Su, Z.; Wassermann, G. The essence of command injection attacks in web applications. *ACM Sigplan Not.* **2006**, *41*, 372–382. [CrossRef]
42. Li, Z.; Qin, Z.; Huang, K.; Yang, X.; Ye, S. A Survey of Deep Learning-Based Intrusion Detection in Cybersecurity. *Appl. Sci.* **2021**, *11*, 1657. [CrossRef]
43. Halfond, W.G.; Orso, A. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, CA, USA, 7–11 November 2005; pp. 174–183.
44. Boyd, S.W.; Keromytis, A.D. SQLrand: Preventing SQL injection attacks. In *Proceedings of the Applied Cryptography and Network Security: Second International Conference, ACNS 2004, Huangshan, China, 8–11 June 2004*; Proceedings 2; Springer: New York, NY, USA, 2004; pp. 292–302.
45. Sri-thong, P.; Bunkhumpornpat, C. Improving Detection of SQL Injection Attack by SMOTE. In *Proceedings of the 2023 27th International Computer Science and Engineering Conference (ICSEC)*; IEEE: New York, NY, USA, 2023; pp. 243–246. [CrossRef]

46. Pietraszek, T.; Berghe, C.V. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the Recent Advances in Intrusion Detection: 8th International Symposium, RAID 2005, Seattle, WA, USA, 7–9 September 2005*; Revised Papers 8; Springer: Berlin/Heidelberg, Germany, 2006; pp. 124–145.
47. Bandhakavi, S.; Bisht, P.; Madhusudan, P.; Venkatakrishnan, V.N. CANDID: Preventing Sql Injection Attacks Using Dynamic Candidate Evaluations. In Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07, New York, NY, USA, 29 October–2 November 2007; pp. 12–24. [CrossRef]
48. Bisht, P.; Madhusudan, P.; Venkatakrishnan, V. CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. *ACM Trans. Inf. Syst. Secur.* **2010**, *13*, 1–39. [CrossRef]
49. Laine, M. *Client-Side Storage in Web Applications*; Aalto University: Espoo, Finland, 2012.
50. Li, L.; Qian, K.; Chen, Q.; Hasan, R.; Shao, G. Developing Hands-on Labware for Emerging Database Security. In Proceedings of the 17th Annual Conference on Information Technology Education, New York, NY, USA, 28 September–1 October 2016; SIGITE '16; pp. 60–64. [CrossRef]
51. Jang, Y.S. Detection of SQL injection vulnerability in embedded SQL. *IEICE Trans. Inf. Syst.* **2020**, *103*, 1173–1176. [CrossRef]
52. Hou, B.; Qian, K.; Li, L.; Shi, Y.; Tao, L.; Liu, J. MongoDB NoSQL injection analysis and detection. In *Proceedings of the 2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*; IEEE: New York, NY, USA, 2016; pp. 75–78.
53. Eassa, A.M.; Elhoseny, M.; El-Bakry, H.M.; Salama, A.S. NoSQL injection attack detection in web applications using RESTful service. *Program. Comput. Softw.* **2018**, *44*, 435–444. [CrossRef]
54. Islam, M.R.U.; Islam, M.S.; Ahmed, Z.; Iqbal, A.; Shahriyar, R. Automatic detection of NoSQL injection using supervised learning. In *Proceedings of the 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*; IEEE: New York, NY, USA, 2019; Volume 1, pp. 760–769.
55. Staddon, E.; Loscri, V.; Mitton, N. Attack categorisation for IoT applications in critical infrastructures, a survey. *Appl. Sci.* **2021**, *11*, 7228. [CrossRef]
56. Gowtham, M.; Pramod, H. Semantic query-featured ensemble learning model for SQL-injection attack detection in IoT-ecosystems. *IEEE Trans. Reliab.* **2021**, *71*, 1057–1074.
57. Rizvi, S.; Kurtz, A.; Pfeffer, J.; Rizvi, M. Securing the internet of things (IoT): A security taxonomy for IoT. In *Proceedings of the 2018 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE)*; IEEE: New York, NY, USA, 2018; pp. 163–168.
58. Soewito, B.; Andhika, C.E. Next generation firewall for improving security in company and iot network. In *Proceedings of the 2019 International Seminar on Intelligent Technology and Its Applications (ISITIA)*; IEEE: New York, NY, USA, 2019; pp. 205–209.
59. Johari, R.; Kaur, I.; Tripathi, R.; Gupta, K. Penetration testing in IoT network. In *Proceedings of the 2020 5th International Conference on Computing, Communication and Security (ICCCS)*; IEEE: New York, NY, USA, 2020; pp. 1–7.
60. Tweneboah-Koduah, S.; Skouby, K.E.; Tadayoni, R. Cyber security threats to IoT applications and service domains. *Wirel. Pers. Commun.* **2017**, *95*, 169–185. [CrossRef]