Measuring the Absolute and Relative Prevalence of SQL Concatenations and SQL-Identifier Injections

Parisa Momeni*, Gabriel Laverghetta*, Kevin Dennis, Bianca Dehann, and Jay Ligatti

Computer Science and Engineering, University of South Florida, Tampa, Florida, USA
{parisamomeni, glaverghetta, kevindennis, bad7, ligatti}@usf.edu

Abstract. SQL Injection Vulnerabilities (SQLIVs), one of the top security risks in modern web applications, arise when unsanitized input is concatenated into dynamically constructed SQL statements. Because existing prepared statement implementations cannot insert identifiers into prepared statements, programmers have no choice but to concatenate dynamically determined identifiers directly into SQL statements. This incompleteness of prepared statement implementations enables a kind of SQLIV called a SQL-Identifier Injection Vulnerability (SQL-IDIV). This article investigates the absolute prevalence of SQL concatenations and SQL-IDIVs via what is, to our knowledge, the largest analysis of open-source software to date. We crawled 4,762,175 files in 944,316 projects on GitHub to identify SQL statements constructed using concatenation. We further classified whether each concatenation constitutes a SQL-IDIV. Our crawler classified 42% of Java, 91% of PHP, and 56% of C# files as constructing at least one SQL statement via concatenation. It further found that 27% of the Java, 6% of the PHP, and 22% of the C# files that concatenate to construct at least one SQL statement concatenate identifiers. Manual analysis indicates that our automated SQL-IDIV classifier achieved an overall accuracy of 93.4%. Further vulnerability testing suggests approximately 22.7% of the web applications meet all of the additional requirements to be exploitable in practice via SQL-identifier injection. PHP applications were particularly exploitable at 38% of applications. We also investigated the prevalence of SQL-IDIVs relative to other software vulnerabilities by analyzing all 3,757 CVE reports of SQLIVs published in 2022-2023, and finding that at least 8% of these reports pertain to SQL-IDIVs. We observed that the prevalence of SQL-IDIVs has risen from 2022 to 2023.

Keywords: SQL Injection · Database Security · GitHub Analysis.

1 Introduction

Code injection vulnerabilities remain one of the top security risks in modern web applications. The 2021 Open Worldwide Application Security Project (OWASP) Top Ten list [47] ranked injection vulnerabilities in the top three with the second most recorded occurrences. Injection vulnerabilities arise when untrusted and unsanitized input is used to generate an output program [50]. One of the most common examples of injection vulnerabilities are SQL injection vulnerabilities (SQLIVs), where unsanitized input gets

* The first two authors contributed equally to this work.

inserted into SQL queries. This input insertion is typically performed using concatenation but may be accomplished using equivalent string-builder functions or string interpolation. For the sake of brevity, as string interpolation is primarily syntactic sugar for concatenation (i.e., interpolation is a form of concatenation), the term concatenation in this article also refers to interpolation unless otherwise noted. Previous work showed that any concatenation of unsanitized input into a SQL statement constitutes a SQL injection vulnerability [50].

As an example, the following C# code is vulnerable to SQL injection.

```
var password = Request.form("password");
var sql = "SELECT * FROM Accounts WHERE Password = '" +
    ↪ password + "'";
```

If the provided password is `'OR 1=1 -`, the constructed SQL query will be:

```
SELECT * FROM Accounts WHERE Password = '' OR 1=1 --'
```

Using this injection, an attacker can bypass password authentication, retrieving all rows in the Accounts table. Many additional SQL injection techniques exist [23].

SQL-Identifier Injection Vulnerabilities (SQL-IDIVs), first formally defined as SQL-Identifier Injection Attacks (SQL-IDIAs) in [6], are a subset of SQLIVs where the user data is inserted into a portion of the SQL statement reserved for a SQL identifier. Identifiers in SQL include the names of tables, columns, indexes, databases, views, functions, procedures, or triggers. The following C# code contains a SQL-IDIV.

```
var column = Request.form("column");
var sql = "SELECT * FROM Accounts ORDER BY " + column;
```

This code is intended to allow the user to specify the column by which to order the output. The expected user input is a column name from the Accounts table. An example malicious user input is as follows.

```
FirstName UNION SELECT * FROM Admins
```

The output SQL program, given the malicious user input, is as follows.

```
SELECT * FROM ACCOUNTS ORDER BY FirstName UNION SELECT *
    ↪ FROM Admins;
```

Assuming that the Accounts table and the Admins table have the same schema, this injection causes the output program to return all records from the Admins table.

Prepared statements are considered the standard defense against SQLIVs, as they separate the user data from the output program [46]. However, modern prepared statement implementations are incomplete. As discussed in Section 3, we surveyed 10 public implementation of prepared statements, and found that none of them support identifier insertions. These implementations, and all other prepared statement implementations that lack support for identifier insertion, do not adequately mitigate SQL-IDIVs [6].

To estimate the need or potential impact of adding support for identifier insertion to prepared statement implementations, this article investigates the prevalence of SQL concatenations and SQL-IDIVs. We propose the following research questions:

1. **RQ1 (Absolute Prevalence)**. What is the prevalence of SQL concatenations and SQL-IDIVs in publicly available code?
2. **RQ2 (Relative Prevalence)**. What is the prevalence of SQL-IDIVs, relative to other software vulnerabilities, in publicly available vulnerability reports?

To address RQ1, we performed what is, to our knowledge, the most comprehensive analysis of open-source software to date. Our automated Github crawler analyzed 4,762,175 files in 944,316 GitHub projects to classify their usage of SQL concatenation. These files contained Java, PHP, or C# source code; these languages were chosen for their popularity (each language returned several million results during initial testing) and their well-established database libraries/frameworks. We also further classified whether the concatenations are into portions of SQL statements reserved for identifiers. We found that 42% of Java, 91% of PHP, and 56% of C# web-application files construct SQL statements via concatenation. Furthermore, we found that 27% of the Java, 6% of the PHP, and 22% of the C# files that concatenate to construct SQL statements concatenate identifiers. Manual analysis of a random sampling of these files indicates that the automated SQL-IDIV classifier achieved an overall accuracy of 93.4%. After confirming the classifier's accuracy, we attempted to exploit a subset of the applications and determined approximately 22.7% of the web applications meet all of the additional requirements to be exploitable in practice via SQL-identifier injection. PHP applications were particularly exploitable at 38% of applications. The repository owners of these exploitable applications were informed of the vulnerabilities.

To address RQ2, we manually analyzed all 3,757 CVE reports of SQLIVs published in either 2022 or 2023, finding that at least 300 (8%) of these 3,757 reports are for SQL-IDIVs. We observed that 8.6% of the classifiable CVE reports from 2022 were for SQL-IDIVs compared to 12.7% in 2023. These results provide further evidence that SQL-IDIVs comprise a nontrivial portion of SQLIVs and indicate that the prevalence of SQL-IDIVs appears to be increasing.

This article presents a definition of SQL-IDIVs that is modified from the original definition of SQL-IDIAs [6]. The definition is improved to allow SQL identifier lists, enabling our classifier to recognize not only SQL-statement locations reserved for identifiers but also locations reserved for a comma-separated list of identifiers. This updated definition is a strict generalization of the original and encompasses SQL-IDIVs that would not have been detected otherwise. Our crawler correctly classified 658 Java and 174 C# files due to this updated definition.

This article contributes:

- An analysis of 10 popular prepared statement implementations in Java, PHP, C#, and JavaScript, and a catalog of the various SQL types they support (Section 3).
- An improved definition of SQL-IDIVs that allows our classifier to correctly classify an additional 832 potentially vulnerable files on GitHub (Section 4).
- An analysis of the absolute prevalence of concatenation in SQL statements and SQL-IDIVs across 4,762,175 GitHub files (Section 5).
- An analysis of the performance of GitHub's code-search API (Section 5.2).
- An analysis of the relative prevalence of SQL-IDIVs via a manual classification of all 3,757 SQLIV CVE reports published in 2022 or in 2023 (Section 6).

This article draws heavily from our earlier paper, which originally appeared in the 21st International Conference on Security and Cryptography (SECRYPT 2024) [12]. We extend that earlier work as follows:

- Section 2 now provides a more detailed and up-to-date overview of related work and includes a new discussion on the limitations of a WordPress input sanitization function.
- Additional methodological details for analyzing prepared statement incompleteness are now provided in the largely new Section 3.
- Section 5 now includes a new analysis of the performance of GitHub's code-search API.
- Section 6 now includes results from an all-new, manual classification of the 1,982 SQLIV CVE reports published in 2023.
- Section 7 now discusses additional avenues for future work.
- Our crawler code and manual analysis data are now publicly available [29,30,31].

This article also contains the following presentational improvements to the earlier conference paper:

- This article now uses the term "SQL-identifier injection vulnerabilities" (SQL-IDIVs) throughout, rather than the older term "SQL-identifier injection attacks" (SQL-IDIAs), in order to more accurately convey that these vulnerabilities can exist without being exploited.
- This article's experimental results are now organized around two new research questions, RQ1 and RQ2, which are concerned with the absolute and relative prevalence of SQL-IDIVs. These research questions introduce a common theme that ties together the experimental designs and results.

2 Background and Related Work

This section reviews previous efforts made to extract data from GitHub and surveys SQLIV mitigation techniques. Given their prevalence, several papers have focused on SQLIVs, including attempts to classify SQLIVs from GitHub.

2.1 Obtaining Data from GitHub

GitHub is the largest public code hosting service with over 100 million developers and over 420 million repositories as of November 2024 [16]. The data available from GitHub is of prime interest to researchers, and several attempts have been made to archive the data and make it more accessible. GHTorrent [17] and GH Archive [21] allow users to view or download GitHub data. Lean GHTorrent [19] is an evolution of GHTorrent that can retrieve data dumps on demand. However, none of these tools offer the data necessary for the present article's experiment. The tools primarily provide metadata about GitHub users, projects, and various other events. While some useful data can be extracted from commit comments and diffs, this article makes use of the GitHub code-search API to obtain a larger set of up-to-date files for analysis.

In addition, the GHTorrent service appears to be deprecated: the GHTorrent web page [18] is no longer available and the once-active GHTorrent Twitter (X) account [14] has not posted since March 7, 2021. The original papers describing the GHTorrent service [17], however, served as inspiration for automating our crawling process. For example, the project data extracted from GitHub is stored in a MySQL database using a schema with similarities to the schema employed by GHTorrent. Several other services, such as the SearchCode [4] service, while still available, appear to be incomplete, with only partial or outdated results, and suffer from unpredictable downtime.

More recent work in the field of MSR (Mining Software Repositories) has produced additional code-search tools. GitHub Search (GHS) [10] continuously mines GitHub repositories, allowing them to be queried based on various attributes. However, GHS does not support projects written in PHP, one of the languages of interest for the present article. PyDriller [53] is a Python framework that provides detailed information about repositories, such as commits, diffs, and source code. Input repository names must be supplied to PyDriller manually (they must be of type string or list of strings), making the tool unsuitable for the automated, large-scale nature of this article's study.

Prior works with goals similar to our own have examined datasets smaller than the dataset studied in the present article, either by design or due to the GitHub API restrictions. In [69], the GitHub code-search API was employed to search for publicly available configuration files that may leak passwords, secret keys, or other sensitive data. This search was limited to a small data set due to the GitHub API only returning a maximum of 1000 results. The crawler program designed for our work overcomes this limitation by utilizing the GitHub API's file size feature, described in more detail in Section 5.1. This technique would likely be insufficient for searching configuration files because such files would be expected to concentrate highly over a small file-size range.

GitHub has been leveraged for other studies focusing on SQL injection attacks. In [62], a total of 117 vulnerabilities were identified across 64,415 PHP source files by comparing the syntactic artifacts of publicly available tutorial code to code on GitHub. The study concludes that there is a substantial link between insecure tutorials and insecure web-application vulnerabilities. The present article, rather than starting from known vulnerable code, starts by identifying public code that interacts with a DBMS and then analyzes that code. The findings of [62] may inform why the analyzed code does not make use of available mitigation strategies. In addition, the technique may be unable to identify SQL-IDIVs due to a lack of representation in tutorials.

2.2 SQLIV Mitigation Techniques

Several solutions have been developed to mitigate SQL injection and code injection vulnerabilities in general. Dynamic methods [50] and tools [2,22,52] that attempt to catch injections at runtime have been proposed but may incur large performance overheads and have not seen widespread adoption. Some dynamic methods, such as SQLPsdem [70], rely on rule-based attack matching, potentially restricting their ability to mitigate zero-day attacks. Static methods [41] that perform information-flow analyses to identify where untrusted user input is concatenated into output SQL programs have not

been adopted due to a high false-positive rate [24] and their inability to detect all types of vulnerabilities [56].

Object-Relational Mapping (ORM) libraries abstract the entire process of writing SQL queries into the object-oriented paradigm. For example, a developer might use code like `Users()->select("name")->execute()` to obtain the names of all users. The ORM library then handles the entire process of constructing the query, connecting to the database, and returning the results. The ORM is also responsible for constructing the SQL statement securely using input validation or prepared statements. These implementations are not without their own vulnerabilities; a CVE report reviewed as part of this project was for an ORM implementation one of the authors had used extensively and believed to be secure at the time [3,37].

Techniques of artificial intelligence and machine learning have been applied to SQLIV mitigation. In [26] and [34], kernel functions and support vector machines are used to classify SQL queries as benign or malicious. Another study [57] trained a neural network to detect SQL injections in URLs. An IDS that detects SQL injection attacks was developed in [32], making use of temporal classification methods to improve classification accuracy. All of these approaches focus on detecting existing SQL injections, as opposed to preventing new injections from occurring.

Input validation, also known as input sanitization, refers to any attempts to filter out, escape, or otherwise remove special characters, control symbols, and other non-data values in untrusted user input. The filtered value is then concatenated directly into the output program. The OWASP SQL Injection Cheat Sheet notes that input sanitization alone cannot prevent all SQL injections in all situations [46]. This limitation of input sanitization arises due to the large number of input symbols and edge cases that must be accounted for. Even well-tested input validation implementations are not always reliable. For example, the `esc_like` function for escaping a string in a `LIKE` statement in WordPress does not output a string that can be safely inserted into a SQL query. As explained in the function's documentation for version 4.0.0 (the latest version of the function) [65], further input sanitization or prepared statement usage is required to reliably prevent SQLIVs. In addition, storing filtered data in a database and later reusing it without filtering the data again can lead to second-order injection vulnerabilities [1].

WordPress Prepare Function The `prepare` function from the wpdb class in WordPress [67] is particularly relevant to this article. This function performs SQL-escaping on a given input string, outputting a sanitized string which may then be executed by a DBMS as a SQL command. Format specifiers may be used as placeholders for user input. Although the name "prepare" may suggest that this function is a prepared statement implementation, this is not the case. The `prepare` function does not cause the DBMS to precompile the given query string. Another function, such as `query` [68], must be used to execute the sanitized query string output by `prepare`. An example invocation of the function is shown below, where `$data` is obtained from user input:

```
$sql = $wpdb->prepare("INSERT INTO Table VALUES(%d);",
    ↪ $data);
```

Originally, the function supported the %d, %f, and %s specifiers for integers, floats, and strings, respectively. The 6.2.0 update (released on October 8, 2022) [33], added support for the %i specifier, which acts as a placeholder for SQL identifiers.

The main documentation page for the wpdb class [66] discusses the `prepare` function. However, the main page does not include information on the %i specifier, listing only the %s, %d, and %f specifiers in the possible placeholders. This hole in the documentation may contribute to a lack of awareness of the %i specifier among WordPress developers, as reflected in multiple online tutorials not containing information about the %i specifier [27,28,51]. Software vulnerabilities are often propagated through the reuse of insecure tutorial code [62,63], so this gap in the documentation may inhibit the effectiveness of the %i specifier in practice. In any case, the reliance on input sanitization is incomplete, as discussed above.

3 Prepared Statement Incompleteness

Prepared statements are considered the standard SQLIV mitigation technique [7,46], as they provide a clear distinction between code and noncode in constructed SQL statements. Instead of concatenating or otherwise inserting noncode (e.g., a string or numeric literal) directly into the constructed statement, the programmer inserts a placeholder symbol where the noncode should appear. The noncode value is then passed alongside the constructed statement to the DBMS, which begins executing the statement and referring to the noncode value when a placeholder is encountered. Prepared statements can be used to prevent other types of injection attacks but require that the output programming language and the corresponding interpreter provide support for them. This section presents an analysis of 10 prepared statement implementations (2 in Java, 2 in PHP, 3 in C#, and 3 in JavaScript) and catalogs the various SQL types they support.

For Java, we reviewed Java Database Connectibity (JDBC), the standard Java API for database access, and jOOQ, an alternative library.

- JDBC [48] includes support for replacing a placeholder with any of the possible types of SQL literals. However, the library does not support defining placeholders for identifiers and replacing placeholders with identifiers.
- In jOOQ, all SQL statements are prepared statements by default [25]. The DefaultPreparedStatement class implements the standard Java PreparedStatement interface [13]. Thus, jOOQ supports the same values in prepared statements as JDBC.

For PHP, we surveyed mysqli and PDO, neither of which supports identifier insertions in prepared statements.

- The mysqli extension [58] contains the `mysqli_stmt::bind_param` function [59], which binds variables to a prepared statement as parameters. The supported bind variable types include int, float, string, and blob (binary large object). There is no support for identifiers or replacing placeholders with identifiers.
- PDO (PHP Data Objects) [60] supports 6 types [61]: booleans, null values, integers, strings, large objects, and recordsets.

For C#, we reviewed three well-known prepared statement implementations: SqlDbType, DbType, and MySqlDbType. None of these libraries support replacing placeholders with SQL identifiers.

- SqlDbType [36] supports a total of 35 different types, all of which are SQL literals.
- DbType [35] supports 28 different types. These include various SQL literals as well as the "Object" type (a generic type representing any reference or value type not explicitly represented by another DbType value).
- MySqlDbType [49] supports 41 different types, including various literals, collections of literals (such as enums and sets of strings) and JSON.

For JavaScript, we reviewed one PostgreSQL library and two libraries for MySQL. One of these libraries does not implement prepared statements, and the prepared statement implementations for the other two libraries do not support identifier insertions.

- The node-postgres [5] package supports placeholders for null values, dates, buffers, arrays, objects, and all other standard SQL types. The documentation explicitly states that "PostgreSQL does not support parameters for identifiers." It is recommended to use the node-pg-format [11] package to sanitize input if dynamic identifiers are needed for queries. However, the most recent commit for node-pg-format was pushed on February 18, 2017; the repository was archived on March 25, 2022.
- The mysql [40] package does not support prepared statements (prepared statements are included in the "Todo" section of the documentation). There are functions that perform client-side escaping of SQL literals and SQL identifiers, but the documentation cautions that these are not prepared statements.
- mysql2 [39] supports null values, doubles, booleans, datetimes, JSON, buffers, and strings. Identifiers are not supported.

A summary of each library is given in Table 1. The results indicate that current prepared statement implementations do not support placeholders in locations where SQL identifiers are expected. If there is a need for dynamic, user-defined identifiers in constructed queries, these libraries will be insufficient for preventing SQL-IDIVs [6].

4 SQL-IDIVs with Identifier Lists

In SQL, some identifiers may appear in a list, including the two most common identifier types: table and column names. The original definition of SQL-IDIAs [6] is limited to applications that concatenate a single identifier into a SQL statement. This section presents a generalized definition of SQL-IDIVs which supports identifier lists.

Definition 1. *An* identifier list *consists of a sequence of one or more identifiers separated by commas, with initial and/or terminating commas also allowed.*

Examples of identifier lists are shown below, where ε represents the empty string.

```
id1
id1, id2, id3
ε, id2, id3, ε
```

Table 1: The data types supported by prepared statement libraries

Language	Library	Supported Types	Reference(s)
Java	JDBC	All SQL literals	[48]
	jOOQ	All SQL literals	[13,25]
PHP	mysqli	int, float, string, and blob (binary large object)	[58]
	PDO	boolean, null, int, string, large object, recordset	[60,61]
C#	SqlDbType	Various SQL literals (including integers, floating-point numbers, dates, times, chars, and strings)	[36]
	DbType	Various SQL literals as well as "Object" (a generic reference or value type)	[35]
	MySqlDbType	Various SQL literals, JSON, sets of string literals, and enums	[49]
JavaScript	node-postgres	All standard SQL literals	[5,11]
	mysql	None (library does not support prepared statements)	[40]
	mysql2	null, double, boolean, datetime, JSON, buffer, string	[39]

The following items are not considered identifier lists because they contain non-identifiers or violate the list format.

```
0, 1, id1
SELECT, ORDER BY, id1
id1, ε, id2, id3
```

Definition 2. *An application contains a* SQL-IDIV *iff the application constructs a SQL statement S by concatenating an untrusted input i into S and there exists an identifer list l such that concatenating l into S in place of i causes S to be a valid SQL statement.*

A SQL-identifier injection occurs when a SQL-IDIV-containing application concatenates a non-identifier list, or an invalid-identifier list, into a SQL statement in place of a valid identifier list.

Definition 3. *A* SQL-Identifier Injection *occurs in a SQL-IDIV-containing application iff the concatenated input i provided dynamically either is not an identifier list or is an identifier list that, when concatenated into S, makes S an invalid SQL statement.*

```
var sortOrder = Request.form("sortOrder");
var sql = "SELECT * FROM Accounts ORDER BY firstName " +
    ↪ sortOrder;
```

Fig. 1: A SQL-IDIV-containing application expecting either `ASC` or `DESC` as input

Figure 1 presents an application program that contains a column-name-based SQL-IDIV. The intent is for users to set the sorting order by specifying `ASC` or `DESC`. How-

ever, an identifier list such as `, lastName` may be substituted instead, as described in Definition 2, resulting in a query that orders by multiple columns. The following three examples demonstrate how this application may be attacked.

1. An attacker may input `, SLEEP(3600).` This input is not an identifier list, but the resulting SQL statement is valid and causes the database to sleep for 1 hour, a sleep-based SQL injection that may result in denial of service. This example input demonstrates that the attacker can execute malicious code; more complex attacks are also possible, for example, by using subqueries or other known techniques [23].
2. An attacker may input `, SELECT,` which is also not an identifier list but in this case produces a SQL statement that will fail to compile. Depending on the environment, this attack might leak DBMS metadata (e.g., through error messages), allowing the attacker to perform reconnaissance. The attack may also deny service to other users.
3. An attacker may input an identifier list that likewise produces an invalid SQL statement. In this case, the SQL statement may be invalid because the identifiers are undefined (e.g., specifying a column name not present in the schema), or due to an incorrect list size. Using the code in Figure 1, if an attacker inputs `, foo,` assuming `foo` is not a column defined in the schema, this injection will result in a runtime error, which again may leak metadata or result in denial of service.

All of these examples are SQL-identifier injections by Definition 3.

Proposition 1 (SQL-Identifier Injection Definition Generalization). *Definition 3 strictly generalizes the definition of SQL-IDIAs in [6].*

Proof Consider an arbitrary application, A, that is vulnerable to SQL-IDIAs using the definition in [6]. By that previous definition, A builds a SQL statement S where there exists some user input, a single identifier i, such that when A concatenates i into S, it makes S a valid SQL statement. Because Definition 1 allows for an identifier list to be composed of a single identifier, i must be a valid identifier list as well. Therefore, A meets the requirements of Definition 2, and any SQL-IDIA on A according to [6] also satisfies Definition 3. On the other hand, Figure 1 shows an application that allows an injection of an identifier list but not a single identifier, making it vulnerable to SQL-identifier injection by Definition 3 but not by the definition of [6]. Thus, Definition 3 strictly generalizes the definition of SQL-IDIAs in [6].

5 Experiment I: SQL Concatenation and SQL-IDIVs on GitHub

To address RQ1, we obtained 4,762,175 files via GitHub's code-search API and classified their usage of SQL statement concatenation. This section discusses the methodology and results of the GitHub experiment, along with our manual verification of the results. Our GitHub crawler code, classifier regular expressions, and manual verification data are available online [29,30,31].

5.1 Experiment I Methodology

An illustration of the general workflow for the GitHub crawler and the classifier is shown in Figure 2 [12]. This workflow follows the same high-level structure of other

tools such as GHTorrent but uses the GitHub code-search API exclusively. The database tracks all files individually and includes the commit that each file was last updated on. This commit-URL provides a static reference to the file and will not change as the file is updated, allowing for the data to continue to be accessible over time. These files can then be provided to the classifier for analysis. After analysis, the results are stored in their own table, allowing the classifier to be run multiple times or swapped out entirely without invalidating the stored files.

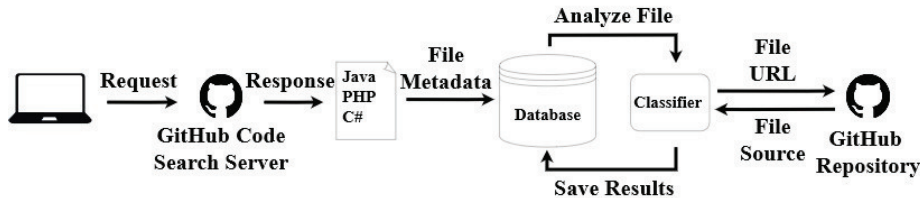

Fig. 2: Workflow for the GitHub crawler and classifier [12]

Crawling GitHub's API The experiment began by crawling the GitHub API to identify files to analyze. GitHub grants all authenticated users the ability to quickly search for specific strings in source files across the uploaded repositories. The GitHub API limits code searches to the first 1000 results, requiring a workaround; other limitations or challenges with the API are described in Section 5.3.

For each target programming language, a popular database library was chosen and the GitHub API was used to locate files that contain calls to the library function that executes a SQL command. For example, JDBC was chosen for Java, and the GitHub API was queried for Java files containing the string `executeQuery`. The library to use was determined by the number of results from GitHub; GitHub reported about 3.6 million entries for JDBC. The program recognizes and classifies several different libraries for each language and libraries can be quickly added; this is described in detail in the next subsection.

To overcome the API's limit of 1000 results for a query, the crawler program splits the data into subsets based on file size. The API allows users to specify the minimum and maximum file size and will only return files that are between the specified range. By decreasing the range width, the number of files in a subset can be fit into the result limit; we refer to these subsets as "frames". After retrieving the results for the current frame, the minimum and maximum file size can be increased, and the next frame of files can be requested. The crawler program automates this process, decreasing or increasing the maximum file size to keep the frame size as close to 1000 results as possible.

Finally, the API results are paginated, with each page containing up to 100 files. Thus, each frame can contain up to 10 pages of 100 results each, adding up to a maximum of 1000 files in the frame. Figure 3 demonstrates how the GitHub data is split into frames and pages.

Frame 1	Frame 2	...	Frame N
Page 1	Page 1	...	Page 1
Page 2	Page 2		Page 2
...	...		...
Page 10	Page 10		Page 10

Fig. 3: Division of GitHub data into frames by file size; each frame consists of 10 pages

GitHub places a restriction on the computation time allocated to any individual query. This restriction is in addition to the normal limits on the number of requests that an individual user can make in a given time frame. When the computation limit is reached, the GitHub API returns the current results, even if those results are not completed. Ideally, pages would always contain 100 results, but due to the computation limits, pages often contain significantly less than that. To take full advantage of the potential downtime between individual API requests, the classification process described in the following subsection was performed between subsequent API calls. After obtaining a set of results, the results were immediately processed, allowing time for the API limits to expire.

GitHub also allows for most of their API to be accessed using GraphQL [20], a query language for APIs. Using GraphQL to query an API greatly increases performance, as it allows users to specify exactly which data they need from a single endpoint, and multiple resources can be retrieved in a single request. Unfortunately, this does not include the code-search API. The crawler program uses GraphQL to retrieve all other metadata from GitHub, such as project descriptions and star counts.

SQL Classifier After obtaining candidate files using the code-search API, the program downloads and analyzes the files to find potential misuse of concatenation in the construction of SQL statements. The classifier determines the usage of prepared statements or concatenation in a given file using regular expressions. For example, the following PHP code interpolates the `$table` variable into a SQL statement.

```
$sql = "SELECT * FROM $table";
```

The classifier is primarily focused on identifying SQL statements inside string literals in source code. To that end, the classifier abstracts language-level syntax rules, such as identifier naming conventions and the symbol used for concatenation, from its regular expressions. This allows users to add support for new programming languages without modifying the underlying classifier program.

The classifier first identifies all instances where the given file constructs SQL code and then classifies the file into one of four categories: none, hardcoded, string concatenation, or string interpolation. The “none” classification indicates that given the file contained no SQL statements, “hardcoded” means all SQL statements were hardcoded

or used prepared statements, "string concatenation" means one or more statements were constructed using concatenation, and "string interpolation" means one or more statements were constructed using string interpolation or concatenation.

Next, all locations in SQL statements that contain or expect a SQL identifier are classified into the same categories, with the addition of a "string concatenation list" category which represents misconstructions that would only be detected using Definition 2. Any identifier types not found in the file are marked with the "none" classification (e.g., the file contains no SQL that calls a stored procedure).

5.2 Experiment I Results

Crawler and Classifier Results The crawler successfully obtained a total of 4,762,175 files from GitHub. The number of files per programming language is presented in Table 2 [12]. These files were spread across a total of 944,316 GitHub projects. We compared the hashes of the obtained files to identify and filter out duplicates. To avoid skewing the analysis, all of the results presented in this article are based on the data set of unique files containing SQL. We henceforth ignore duplicate files and files without SQL-statement construction.

Table 2: Files and projects reviewed per language [12]

	Total Files	Unique files containing SQL	Projects
Java	2,372,363	1,273,078	461,896
PHP	1,587,766	1,083,294	307,089
C#	802,046	526,921	175,331
Total	**4,762,175**	**2,883,293**	**944,316**

No limits were placed on the maximum size of the file that GitHub might return. The size frame was increased until no results were returned. The cumulative percentage of files by file size can be seen in Figure 4 [12]. The largest file obtained was a one-gigabyte Java file. About 95% of the obtained files for all three languages were under 40 MB in size, with the remaining 5% scattered haphazardly between 40 MB and 1 GB. The graphs presented in this article are restricted to under 40 MB to prevent them from being skewed by large outliers.

The classifier identified that, of the unique files that contain SQL, 144,461 (11.3%) Java files, 63,239 (5.8 %) PHP files, and 66,026 (12.5%) C# files contained at least one instance where an identifier was concatenated or interpolated during the construction of a SQL statement. Table and column names were the two most common identifiers for all three languages. Table 3 [12] presents the number of constructions for non-identifier and identifier locations. For each location, the constructions are further grouped by their type, which can be hardcoded, string concatenation, or string interpolation. The table does not include the number of files that construct SQL statements using prepared statements libraries because such libraries were often called, but misused (i.e., no placeholders were used and user input was concatenated to the query).

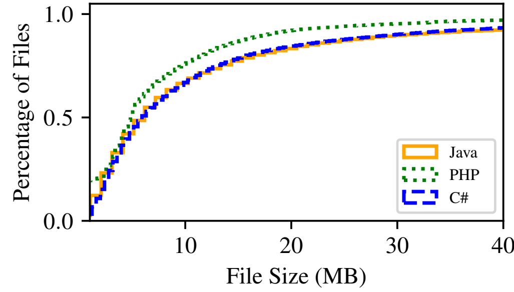

Fig. 4: Cumulative percentage of files by size [12]

Table 3: Concatenation in SQL statements by location in unique files [12]

	Any location		Identifiers	
	Hardcoded	Concatenated + Interpolated	Hardcoded	Concatenated + Interpolated
Java	732k	534k + 6k = 540k	1.1M	143k + 0.7k = 144k
PHP	96k	101k + 904k = 1M	1.0M	21k + 44k = 65k
C#	230k	180k + 117k = 297k	461k	47k + 19k = 66k
Total	**1M**	**815k + 1.0M = 1.8M**	**2.6M**	**211k + 64k = 275k**

Table 4 [12] details the statistics of the unique files analyzed. The crawler and analyzer classified 42% of Java, 91% of PHP, and 56% of C# web-application files as constructing SQL statements via concatenation. Of the files that concatenate to construct SQL statements, 27% of the Java, 6% of the PHP, and 22% of the C# files concatenate identifiers.

We sorted the files classified as containing concatenation by file size to determine if there is a correlation between file size and the likelihood of concatenation occurring; Figure 5 [12] presents the results.

GitHub API Performance While running the crawler to obtain files for classification, we tracked and logged the crawler's interactions with the GitHub API. The experiment lasted 3 months and 3 days. The three languages were split across four different machines, each using a unique GitHub account. Over 100 GB of data was transmitted, with over 1,000 hours dedicated just to networking. The GitHub data was split over 34,706 frames with a total of 128,264 pages.

The networking data can be further broken down into three groups depending on the type of request: code search, GraphQL, and file downloads. The code-search category contains calls to the GitHub code-search API, used to identify interesting code files for classification. GraphQL requests represent the calls made to the GraphQL API, which was used to download other metadata about the projects that contained the code files. The last category represents requests to download the code files after identification

Table 4: Statistics of the unique files analyzed [12]

	% of files with concat.	% of files with identifier concat.	% of files with concat. that have identifier concat.
Java	42.5%	11.3%	26.7%
PHP	91.1%	5.8%	6.4%
C#	56.4%	12.5%	22.2%
Total	**63.3%**	**9.5%**	**15.0%**

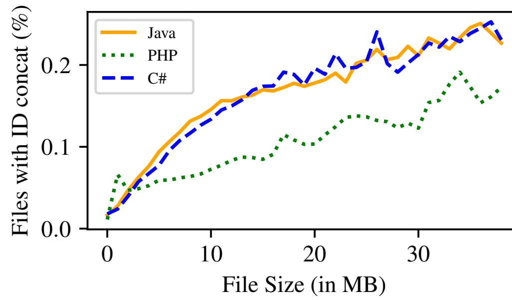

Fig. 5: Percentage of unique files with SQL identifier concatenations by file size [12]

using the GitHub API. Table 5 presents, by language, the total number of bytes, total runtime, and the total number of individual requests for each networking category.

Notably, the crawler program never once hit the regular API limit of ten code searches per minute. However, the program regularly hit the secondary limit, a limit enforced when a user makes too many large requests in a short period. We found waiting a minute was usually sufficient for the secondary limit to be lifted, but the exact length of such a restriction is not predictable by design. We implemented the steps recommended by GitHub [15] to reduce the number of secondary limits, such as caching results using the okhttp3 library [55], but it does not appear code-search results support caching; for more discussion on the unpredictability of the code-search API, see Section 5.3. Figure 6 presents the total number of secondary rate limits that occurred grouped by time of day and day of the week.

The number of files that were reported in each page was also recorded. Due to GitHub's early termination of expensive queries, many pages would not contain the maximum 100 results. The API does allow users to specify the number of results returned in a page, but the authors speculate this value is used when determining the maximum runtime for an individual query. For example, during early testing of the GitHub API, dropping the number of results per page from 100 to 50 did not appear to increase the overall number of results obtained. Over several runs, two pages of 50 results each and a single page of 100 results will both only return about 60 total results on average.

Table 5: Networking performance by language and type

	Search Code			GraphQL			File Downloads		
	Bytes (GB)	Time (h)	Requests	Bytes (GB)	Time (h)	Requests	Bytes (GB)	Time (h)	Requests
Java	15.9	53.3	53,873	0.28	40.3	16,492	39.4	402.7	2,568,364
PHP	12.4	52.9	52,449	0.18	10.9	7,203	13.7	334.9	1,482,427
C#	5.6	15.2	17,999	0.15	10.6	3,174	17.0	149.2	1,068,642
Total	**33.9**	**121.4**	**124,321**	**0.61**	**61.8**	**26,869**	**70.1**	**886.8**	**5,119,433**

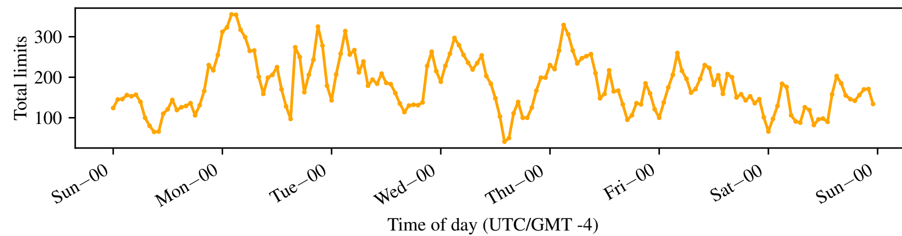

Fig. 6: Total number of secondary rate limits encountered by time of week

Figure 7 presents the average number of results in a page grouped by time of day and day of the week.

Although we hypothesized that GraphQL would be faster than the code-search API, we found that the GraphQL queries were instead slower given the number of bytes transmitted. Instead, the GraphQL performance stands out for the low overhead of the transmitted data and the API's reliability. The code-search API only returned about 50-60 results on average for each request. Every single GraphQL query, on the other hand, returned 100% of the requested data. Every project that was requested via the GraphQL API was returned. In addition, every byte included in the GraphQL data was useful, desired data as it was specifically requested. Of the almost 34 GB of code-search data, only about 10% of that was actually desired. The search code response data is filled with many long links to other API endpoints to retrieve potentially related data that was not useful for this study.

The code-search API's performance appears highly correlated to the typical work week and is superior during off-peak hours. As shown in Figures 6 and 7, the API was more reliable during the weekend and around 6:00 PM US Eastern Time. This time of day is at the end of typical working hours in most of the U.S and outside of typical working hours in many other places of the world; 6:00 PM US Eastern time is equivalent to 10 PM in London, England, 3:30 AM in New Delhi, India, and 7:00 AM in Tokyo, Japan. In addition to returning a larger number of results, the number of times the API hit a secondary rate limit was also lower outside of these hours. There appears to be a strong correlation between the rate-limiting and the number of returned results, with the peak rate-limiting occurring simultaneously with the low number of results.

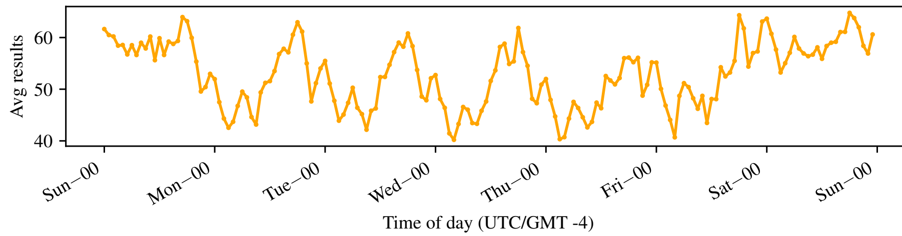

Fig. 7: Average number of results returned in a page by time of week

5.3 Discussion

Addressing RQ1, our results show that 63% of the obtained files construct at least one SQL statement using concatenation. Of these files, 15% concatenate SQL identifiers. Table names were the most commonly concatenated identifier, followed by column names.

String interpolation was almost nonexistent in Java programs but was common in PHP, with over 83% of PHP files utilizing interpolation to construct their SQL statements. PHP also had the highest concatenation rate, which is reflected in the CVE analysis in Section 6, where the majority of vulnerability reports were observed to be in PHP web applications, commonly built using WordPress.

As shown in Figure 5 [12], file size is correlated with the likelihood of identifier concatenation. We hypothesize that this is likely a result of large code bases serving a more complex purpose, with a larger number of queries that must be dynamic in nature.

An additional 658 Java files and 174 C# files were classified correctly due to Definition 2. All of these files concatenated values into a location reserved for a SQL identifier list and would be incorrectly classified according to the definition in [6].

Limitations The amount of files that can be obtained is restricted by the somewhat unpredictable results of the GitHub code-search API. This behavior can be seen even using the code-search feature available on the GitHub website. When searching for a string and viewing the code results, GitHub will report the number of code results at the top of the page. Refreshing the page repeatedly will show various different numbers due to the run time limits placed on the query. An accurate estimate of the number can be obtained by taking the maximum value seen over a long period of time, particularly during non-peak hours.

This issue is also present when retrieving the results. Querying the results multiple times will very likely return a different subset of data each time, which can not be corrected due to the inability to order results. Combined with pages often returning only a partial set of data, obtaining all the files in a given frame may require requesting the same pages repeatedly, even during non-peak hours. This limitation is offset by the large amount of available data; successfully downloading only 25% of the data may give millions of files, depending on the search query and target language.

Table 6: Results of manual analysis of randomly sampled GitHub files [12]

	Total Files	True Pos. (TP)	False Pos. (FP)	True Neg. (TN)	False Neg. (FN)	FP Rate $\left(\frac{FP}{FP+TN}\right)$	FN Rate $\left(\frac{FN}{FN+TP}\right)$	Precision $\left(\frac{TP}{TP+FP}\right)$	Accuracy $\left(\frac{TP+TN}{Total}\right)$
Java	385	319	12	45	9	0.21	0.027	0.964	0.945
PHP	385	332	14	24	15	0.368	0.043	0.960	0.925
C#	385	290	18	69	8	0.207	0.027	0.942	0.932
Total	**1,155**	**941**	**44**	**138**	**32**	**0.242**	**0.033**	**0.955**	**0.934**

The use of regular expressions to identify concatenation may be insufficient if developers construct queries in particularly creative ways. However, given the results of the manual classifier verification in Section 5.4, this issue does not seem to be significant in the context of SQL-IDIVs.

5.4 Classifier Verification

We manually verified a random sampling of the files from each language to determine the accuracy of the classifier's regular expressions. The ideal sample size for each language subset was determined to be 385 for a precision level of 95% using Cochran's formula [64]. MySQL's `RAND` function was used to randomly select the files for analysis. As no other classifier for SQL-IDIVs exists that would enable an automated comparison, the verification was instead performed by downloading the file, reviewing the source code, and verifying that the construction of SQL statements in the file corresponded with the classifier's results. For example, a false positive occurred if the classifier reported that a file contained string interpolation of a column identifier but no such string interpolation existed in the file. The classifier exhibited a false negative when it failed to detect concatenation in an output SQL program that was located in the file.

The classifier had an overall precision of 95.5% and an overall accuracy of 93.4%. False positives, where a file was marked as constructing SQL with a concatenated value but did not, generally arose due to SQL-like statements in comments, logging, or error messages. False negatives, on the other hand, came from programmers constructing or formatting their SQL output in an unusual or unpredicted fashion. The results of the manual verification are shown in Table 6 [12].

5.5 Vulnerability Exploitation

The classifier was shown to be reliable in identifying files where potentially unsafe concatenations had occurred, but the unsafe code is only exploitable if the output SQL code can be manipulated by the attacker and is not dead code. The concatenated values must be derived from user input without proper sanitization and the code must be reachable during normal program execution. While both static and dynamic tools exist to reliably detect SQLIVs, these tools cannot reliably detect SQL-IDIVs; an example of sqlmap [54] (an automated SQLIV detection and exploitation tool) failing to exploit a SQL-IDIV-containing application is shown later in this section. In order to determine

how many of the identified applications are exploitable in practice, we manually installed and tested a subset of the manually verified applications. For all repositories determined to be exploitable, their owners were notified of the vulnerabilities.

The exploitation process is similar to a standard code audit. First, the code and GitHub project is reviewed to determine if the project is a web application and if the program dynamically generates a SQL output program using user input. If the code is determined to likely implement a SQL-IDIV-containing application, the repository is cloned and the application installed. In general, installing the applications involves setting up the database schema for the application, filling the database with sample data, and deploying the application on a web server. The database steps are similar for all three languages, but the deployment process varies for each language and, for Java, varies greatly depending on the build tools chosen by the developer (e.g., Maven, Ant, Gradle). Once running, the application is navigated in the intended fashion until the vulnerable code is executed; this process may be eased using standard debugging tools or adding a line to print the output SQL program (the programs were otherwise unaltered and any debugging print statements were removed for final confirmation of the vulnerability). Finally, if the application is truly vulnerable, code is injected via the identified vulnerable input to cause observable malicious behavior, typically a call to a sleep function.

During the exploitation process, we assumed the following: (1) all databases and tables that are referenced in the code exist and are populated with at least one entry, (2) the application code is unmodified, (3) the application runs with the standard configuration provided (if applicable), and (4) only the single flagged file chosen as part of the random analysis is considered. Projects that failed to compile or could otherwise not be installed and exploited within two hours were recorded as “Not Exploitable”, but these applications may still be exploitable if the compilation issues were corrected or more time was allocated. Programs were otherwise marked “Not Exploitable” if the vulnerable code was dead code or if the concatenated values were not derived from user input, statically compared to an allow list, or dynamically verified (that is, the database schema was queried to ensure that the identifier existed). The application was marked as “Exploitable” only if SQL code could be injected and malicious behavior observed.

Only web applications were considered relevant and exploited; while a serverless GUI or text application may be vulnerable, there is generally little value in such exploits as the exploited database would be on the user’s machine and could be modified in simpler ways. The breakdown of applications by language, interface type, and purpose is shown in Table 7 [12]. An application falls into one of three interface types: web application, standalone application, or other (e.g., library/framework, client-server, build system). All the PHP applications were web applications, so only a total row is shown for PHP. The purpose category is used to differentiate applications that were clearly student projects, tutorials, and sample/beginner code from other applications. Relevant markers for being classified as a student application included referencing a course directly, including or referencing a grade or rubric, or an assignment directory structure (e.g., a folder named “Test1”). Tutorial/beginner code consisted of “Hello world” programs or other obvious references to a tutorial (such as the repository owner having the name of a tutorial site).

Table 7: Types of applications analyzed [12]

Interface	Purpose			Total
	Student	Tutorial	Other	
Web	1	2	18	21
Standalone	5	2	14	21
Other	1	1	6	8
Java Total	**7**	**5**	**38**	**50**
PHP Total	**4**	**2**	**34**	**40**
Web	1	0	26	27
Standalone	4	2	21	27
Other	0	1	7	8
C# Total	**5**	**3**	**54**	**62**

Table 8: SQL-IDIV-containing applications by Identifier Type [12]

	Table	Column	Column (ORDER BY)
	# / Total	# / Total	# / Total
Java	1 / 14	0 / 3	3 / 6
PHP	4 / 24	2 / 14	9 / 11
C#	0 / 14	0 / 21	1 / 5
Total	**5 / 52**	**2 / 38**	**13 / 22**

We inspected 50 Java, 40 PHP, and 62 C# applications (152 total). Twenty-six applications were student projects or tutorials, and 21 Java and 27 C# projects were web applications. Twenty of the 88 (22.7%) web applications were confirmed to be exploitable via SQL-identifier injection: 4 out of 21 Java (19%), 15 out of 40 PHP (38%), and 1 out of 27 C# (4%). Only 2 of the exploitable web applications were student programs (1 PHP and 1 C#); all the others appeared to serve a more professional purpose. A summary of the vulnerable applications grouped by the vulnerable identifier type is shown in Table 8 [12]. Note that the total numbers will not sum to the number of applications as a single application may include a combination of all three possible identifiers. Multiple instances of a single type in an application were only counted once. These numbers are a lower bound as only the randomly selected file in each project was considered; several applications were noticeably vulnerable in other files, but the examination was limited to the randomly chosen subset of files.

Column names used in `ORDER BY` statements were the most likely to be vulnerable, with 12 (60%) of the 20 injections occurring due to user input being concatenated into an `ORDER BY` statement. Three of these exploitable applications would not have been detected without the use of Definition 2.

While the focus was on exploiting SQL-IDIV-containing applications, a number of other vulnerable applications were observed during the process. A total of 25 other applications were exploitable but not via identifiers (7 Java, 14 PHP, and 4 C#). This number is a very conservative minimum; since SQLIVs were not a focus, these applications were only discovered passively and because they were very obvious.

In addition to these SQLIV-containing applications, 2 out of 3 Java libraries analyzed may cause an application to be vulnerable to a SQL-identifier injection as they implemented no mitigation techniques. Since these are not applications themselves, they are not exploitable; however, if a developer assumes the library sanitizes input and passes user input without their own validation, the program would be exploitable. The third library implemented dynamic validation by querying the database for valid table names and comparing the input to those values.

A total of 13 applications were not exploitable because the identified concatenation occurred in dead code. These applications contained functions that took an argument and directly concatenated it into a SQL output program, but the functions were not called and thus the applications were not exploitable. Although these projects were not exploitable at the time of our manual analysis, they may be prone to exploitation in the future. If a project involves a long-lived code base regularly updated by developers new to the project, it is easy to imagine a scenario where feature creep results in the misuse of functions not originally meant to be invoked.

Combining all of these different categories together, a total of 60 exploitable and problematic applications were identified out of 152 (20 SQL-IDIVs, 25 other SQLIVs, 13 dead-code concatenations, and 2 non-sanitizing libraries).

The results of the crawler allowed for the exploitation process to be sped up dramatically. Using the crawler data, the vulnerable SQL statement in a given application could be almost immediately identified and understood. After growing accustomed to the process, a given application could be analyzed for vulnerability and documented within 5-10 minutes (although installing the application and performing the exploit generally took far longer). This process could be further optimized by taking advantage of the full data set, which was not possible for sampling a random subset of the data. For example, a non-random analysis could prioritize applications that use `ORDER BY` statements or batch all files in a project (rather than analyzing a single file).

Figure 8 [12] demonstrates a SQL output program from one of the exploitable PHP applications. The $c variable is user input interpolated directly into the output SQL program. The intended value of this $c variable is either “users” or “crew”, allowing the web page to show either customers or employees using the same code. Notably, sqlmap could not detect any vulnerabilities in this code, regardless of the options used. Any subquery injected into this location would not be syntactically valid without a table alias, and sqlmap does not include this technique. Two other instances were not detectable using sqlmap and were also exploited using a minor syntactic change: the first used a column alias and the second modified an `INSERT` statement by injecting a `SELECT` statement to specify the data (instead of the `VALUES` keyword).

6 Experiment II: SQL-IDIVs in CVEs

MITRE’s Common Vulnerabilities and Exposures (CVE) List [38] tracks publicly known cybersecurity vulnerabilities. A total of 161,955 CVE records were published from 2014 to 2023, as listed in the CVE Details site [8]. Of these 161,955 vulnerabilities, 7,327 (4.5%) were SQLIVs [9]. In 2022, 1,789 SQLIV entries were reported, making

```
$sql="SELECT * FROM $c WHERE ...";
```

(a) Truncated PHP code from one of the exploited programs.

```
    (SELECT SLEEP(10000)) as t --
```

(b) The malicious input; the table alias is necessary to be syntactically valid.

Fig. 8: One of the exploited applications that could not be detected using sqlmap [12]

up 7.1% of the vulnerabilities reported that year and more than doubling the number of 2021 SQLIV CVEs (741) [9].

A CVE record's date of publication is not necessarily equal to its reporting date. For example, of the 1,775 SQLIV CVEs published in 2022, 1,507 were also reported in 2022; the remaining 268 were published in 2022 but reported earlier. We chose to analyze all SQLIV CVEs published (not reported) in 2022 or in 2023 to ensure that our dataset is static.

To address RQ2, we analyzed the 1,775 and 1,982 SQLIV CVE records published in 2022 and in 2023, respectively, to determine the number of these 3,757 SQLIV CVEs that are for SQL-IDIVs. This section discusses the methodology and results of this CVE analysis.

6.1 Experiment II Methodology

The CVE experiment consisted of two phases: (1) downloading relevant CVE records and (2) classifying each record as pertaining to a SQLIV or a SQL-IDIV. Details of each phase are given in the following subsections.

Downloading CVE Records To obtain SQLIV CVE records for analysis, we queried the National Vulnerability Database (NVD) [42] maintained by the National Institute of Standards and Technology. The NVD can be queried using various search parameters. We used the following search parameters to obtain our dataset:

- Category (CWE): CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- Published Start Date: 01/01/2022
- Published End Date: 12/31/2023
- Ordered By: Publish Date Ascending

We wrote a Python script to automate this process. The script accepts a starting date, an ending date, and an output file name. During execution, the script populates the given file with the SQLIV CVE records published during the given timeframe. Our 2022 CVE data file, 2023 CVE data file, and the CVE retrieval script are available online [30].

Classifying CVE Records A CVE entry consists of (1) a CVE-ID, (2) a description of the vulnerability, (3) a list of external references, (4) the assigning CVE Numbering Authority, and (5) the entry creation date. The vulnerability description generally consists of only a few sentences and, while useful for classifying vulnerabilities into broad vulnerability types, it lacks the detail necessary to finely classify vulnerabilities. To overcome this limitation, we analyzed the list of external references. Examples of reference materials include a link to the advisory or patch notes from the software authors, a GitHub repository with the source code for open-source projects, and a proof of concept (PoC) attack.

In our analysis, SQLIVs are considered SQL-IDIVs when they satisfy Definition 2. This classification cannot always be made from the vulnerability description alone due to a lack of technical detail. To determine whether the CVE represents a SQL-IDIV, the CVE must reference either a PoC or freely available, online source code that pertains to the same version of the software referenced in the CVE report. We therefore excluded the 15% of 2022 SQLIV CVEs and 32.5% of 2023 SQLIV CVEs that lacked usable source code and a PoC.

To provide clarity on the classification process, we present two example SQL-IDIVs taken from our CVE dataset. The first example showcases the classification of a CVE report that includes both a PoC and source code, and the second example demonstrates the classification a CVE that provides only a PoC.

The first example is CVE-2023-7157, which details a vulnerability found in an inventory management application written in PHP [44]. The CVE record includes a link to a PoC, and the PoC includes a link to the project's source code. The vulnerable code, located at line 47 of the /ample/app/sell_return_data.php file, is shown below:

```
$stmt = $pdo->prepare("SELECT * FROM sell_return WHERE 1
    ↪ ".$searchQuery." ORDER BY ".$columnName." ".
    ↪ $columnSortOrder." LIMIT :limit,:offset");
```

Although the code makes use of PDO's `prepare` function, no placeholder is used for `columnName` (PDO does not support identifier placeholders in prepared statements regardless, as discussed in Section 3). In addition, the `columnName` variable is obtained directly from user input without any sanitization via the following code:

```
$columnName = $_POST['columns'][$columnIndex]['data']; //
    ↪ Column name
```

As noted in the CVE record, an attacker can manipulate the `columns[0][data]` argument, leading to SQL injection. Since the untrusted user input is concatenated after an `ORDER BY` statement, and `ORDER BY` statements must be followed by an identifier list, this CVE pertains to a SQL-IDIV.

CVE-2023-2592 details a SQLIV present in an older version of the FormCraft WordPress plugin [43]. The CVE record includes a link to a PoC. Although the current version of the FormCraft code may be downloaded from the FormCraft web page [45], this CVE pertains to an older version of the code. The code for this older version is no longer accessible. Thus, a PoC is available for this CVE but not source code.

The PoC explains that, in FormCraft versions below 3.9.7, there is a lack of sanitization of the `sortOrder` query parameter in HTTP requests. SQL code can be injected into the parameter, as shown in the following payload:

```
‘sortOrder=ASC%2c(select*from(select(sleep(20)))a‘
```

It appears that the intended value for the `sortOrder` parameter is either `ASC` or `DESC`. This example attack uses an invalid identifier list to perform a sleep-based injection. Note that `%2c` is the URL encoding for a comma. Since the vulnerable code appears to concatenate user input into a location that would accept an identifier list, but not a single identifier, the code constitutes a SQL-IDIV.

6.2 Experiment II Results

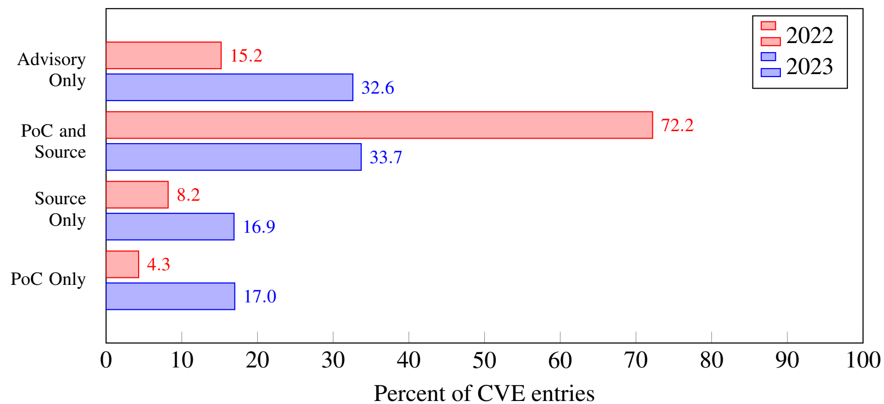

Fig. 9: Resources available for classification of SQLIV CVE entries from 2022-2023

Of the 1,775 CVEs published in 2022, 1282 (72.2%) had both a PoC and source code available, 146 (8.2%) had only source code available, 77 (4.3%) had only a PoC, and 270 (15.2%) had only an advisory. In 2023, 1,982 SQLIV CVEs were published, of which 668 (33.7%) had both a PoC and source code available, 334 (16.9%) had only source code available, 336 (17%) had only a PoC, and 646 (32.6%) had only an advisory. Figure 9 shows the classification resources available for the 2022-2023 SQLIV CVEs. We documented a total of 300 SQL-IDIVs: 130 from 2022 and 170 from 2023. Figure 10 presents the results of the 2,841 classifiable CVEs, classifying them as SQLIVs or SQL-IDIVs.

6.3 Discussion

Addressing RQ2, our results show that SQL-IDIVs comprise at least 300 (8%) of the 3,757 SQLIV CVEs published in 2022-2023. We could not classify 916 CVEs because they lack both a PoC and relevant source code, so the SQL-IDIV percentages

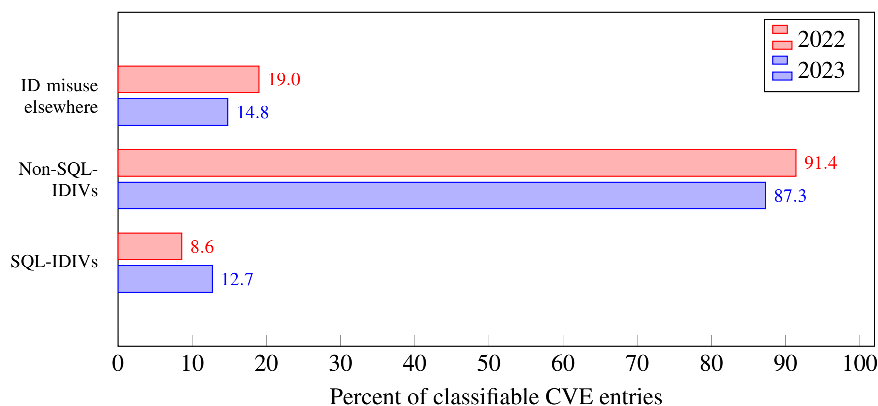

Fig. 10: Classified SQLIV CVE entries for 2022-2023

are lower bounds. Although only 1,336 of the 2023 CVEs were classifiable compared to the 1,505 from 2022, we documented more SQL-IDIVs from 2023 than in 2022. A greater percentage of CVEs in 2023 were found to be for SQL-IDIVs than in 2022, indicating that the prevalence of SQL-IDIVs appears to be increasing.

Approximately 17% of the classifiable CVEs pertain to an application that concatenates at least one identifier elsewhere in its source code (excluding the vulnerable location referenced in the CVE record). Vulnerabilities that were observed in large code bases were often caused by a single missing use of prepared statements.

The percentage of SQL-IDIV vulnerabilities found in the universe of classifiable 2022-2023 SQLIV vulnerabilities (10.6%) is less than the percentage of identifier concatenations found in the universe of GitHub SQL concatenations (15%) as described in Section 5.2. Future work might explore this gap further, to try to make statistical inferences and conclusions about how accurately classifiable CVE reports represent the vulnerabilities present in large open-source data sets.

7 Conclusions and Future Work

SQL concatenations, which form the basis for SQL injection attacks, are prevalent in web applications. In total, 63% of web applications analyzed contained SQL concatenations. SQL identifier concatenations comprised approximately 15% of SQL concatenations. Given that our automated GitHub crawler and code analyzer classified approximately 275K files as containing SQL identifier concatenations, with a precision rate of 95.5%, we estimate our automated framework found approximately 262K web-application files that contain at least one SQL-IDIV. Our manual vulnerability exploitation indicates that, of these 262K files, 22.7% (62K) are likely to meet all of the additional requirements to be exploited in practice. Furthermore, SQL-IDIVs comprise at least 300 (10.6%) of the 3,757 SQLIV CVE records published in 2022-2023. We observed that a greater percentage of 2023 CVEs were for SQL-IDIVs than in 2022,

signaling an increase in SQL-IDIV prevalence. These results indicate that SQL-IDIVs are nontrivially represented in open-source code and in public vulnerability reports.

There are multiple directions for future work. Previous work has described and analyzed a non-public proof-of-concept implementation of prepared statements with coverage of identifiers [6]. This work may serve as a model for expanding a large-scale open-source DBMS such as MySQL or Postgres to include support for identifiers in prepared statements. Incorporating these additions into front-end APIs for commonly used languages would provide additional mitigation of SQL-IDIVs. Future work could also examine input sanitization functions such as `prepare` in WordPress (discussed in Section 2.2), studying their effectiveness compared to prepared statements. This avenue could include an analysis of published CVEs for the wpdb class, to identify any known vulnerabilities in the `prepare` function. In addition, our GitHub crawler can be modified to perform additional types of measurement analysis, such as an analysis of a wider selection of languages or an investigation into the prevalence of SQLIV mitigation techniques (e.g., ORM libraries or input sanitization functions). Another possible avenue for future work is to investigate the use of large language models (LLMs) to classify cases of SQL-IDIVs in input code. The LLM's training and test sets might consist of the CVE records analyzed in the present article. This SQL-IDIV-detection LLM could automate future CVE analysis and assist in detecting SQL-IDIVs in code.

References

1. Anley, C.: Advanced SQL injection in SQL server applications. Tech. rep. (2002), https://crypto.stanford.edu/cs155old/cs155-spring11/papers/sql_injection.pdf
2. Bandhakavi, S., Bisht, P., Madhusudan, P., Venkatakrishnan, V.N.: CANDID: Preventing SQL injection attacks using dynamic candidate evaluations. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2007). https://doi.org/10.1145/1315245.1315249, https://doi.org/10.1145/1315245.1315249
3. B.C. Institute of Technology: CodeIgniter framework 3. https://github.com/bcit-ci/CodeIgniter (2023), retrieved April 26, 2023
4. Boyter, B.: searchcode - source code search engine. https://searchcode.com (2023), retrieved April 26, 2023
5. Carlson, B.: Queries - node-postgres. https://node-postgres.com/features/queries (2024), retrieved November 8, 2024
6. Cetin, C., Goldgof, D., Ligatti, J.: SQL-identifier injection attacks. In: IEEE Conference on Communications and Network Security (CNS) (2019), https://doi.org/10.1109/CNS.2019.8802743
7. Clarke-Salt, J.: SQL Injection Attacks and Defense. Elsevier, 2nd edn. (Jun 2012)
8. CVE Details: Browse vulnerabilities by date. https://www.cvedetails.com/browse-by-date.php (2024), retrieved November 24, 2024
9. CVE Details: Vulnerability distribution of CVE security vulnerabilities by types. https://www.cvedetails.com/vulnerabilities-by-types.php (2024), retrieved November 24, 2024
10. Dabic, O., Aghajani, E., Bavota, G.: Sampling projects in github for msr studies. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). pp. 560–564 (2021). https://doi.org/10.1109/MSR52588.2021.00074
11. Datalanche: node-pg-format. https://github.com/datalanche/node-pg-format (2024), retrieved November 9, 2024

12. Dennis, K., Dehaan, B., Momeni, P., Laverghetta, G., Ligatti, J.: Large-scale analysis of github and cves to determine prevalence of sql concatenations. In: International Conference on Security and Cryptography (SECRYPT) (2024), https://www.scitepress.org/Papers/2024/128352/128352.pdf
13. Eder, L.: Class defaultpreparedstatement. https://www.jooq.org/javadoc/dev/org.jooq/org/jooq/tools/jdbc/DefaultPreparedStatement.html (2024), retrieved November 13, 2024
14. @ghtorrent: GHTorrent. https://x.com/ghtorrent (2024), retrieved November 19, 2024
15. GitHub: Rate limit - GitHub docs. https://docs.github.com/en/rest/rate-limit (2022), retrieved April 26, 2023
16. GitHub: About github. https://github.com/about (2024), retrieved November 26, 2024
17. Gousios, G., Spinellis, D.: GHTorrent: GitHub's data from a firehose. In: IEEE Working Conference on Mining Software Repositories (MSR) (2012). https://doi.org/10.1109/MSR.2012.6224294, https://doi.org/10.1109/MSR.2012.6224294
18. Gousios, G., Spinellis, D.: ghtorrent.org. http://ghtorrent.org/ (2020), retrieved November 19, 2024
19. Gousios, G., Vasilescu, B., Serebrenik, A., Zaidman, A.: Lean GHTorrent: GitHub data on demand. In: Proceedings of the Working Conference on Mining Software Repositories (MSR) (2014). https://doi.org/10.1145/2597073.2597126, https://doi.org/10.1145/2597073.2597126
20. GraphQL Foundation: A query language for your API. https://graphql.org (2023), retrieved April 27, 2023
21. Grigorik, I.: GH Archive. https://www.gharchive.org (2023), retrieved April 26, 2023
22. Halfond, W.G.J., Orso, A.: AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE) (2005). https://doi.org/10.1145/1101908.1101935, https://doi.org/10.1145/1101908.1101935
23. Halfond, W.G.J., Viegas, J., Orso, A.: A classification of SQL-injection attacks and countermeasures. In: Proceedings of the IEEE international symposium on secure software engineering. vol. 1 (2006)
24. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: Proceedings of the International Conference on Software Engineering (ICSE) (2013), https://doi.org/10.1109/ICSE.2013.6606613
25. jOOQ: The jooq user manual. https://www.jooq.org/doc/dev/manual-pdf/jOOQ-manual-3.20.pdf (2024), retrieved November 13, 2024
26. Kim, M.Y., Lee, D.H.: Data-mining based sql injection attack detection using internal query trees. Expert Systems with Applications **41**(11), 5416–5430 (2014). https://doi.org/https://doi.org/10.1016/j.eswa.2014.02.041, https://www.sciencedirect.com/science/article/pii/S0957417414001171
27. Kubelet, K.: The vital role of $wpdb->prepare() in wordpress. https://blog.koddr.io/blog/2023/05/15/importance-wpdb-prepare-wordpress/ (2023), retrieved November 17, 2024
28. Kurth, P.: A beginner's guide to using sql to query the wordpress database. https://hookturn.io/custom-wordpress-sql-queries-for-beginners/, retrieved November 17, 2024
29. Laverghetta, G.: SQLcrawler. https://github.com/glaverghetta/SQLcrawler (2023), retrieved November 18, 2024
30. Laverghetta, G.: SQL-IDIV-Manual-Analysis. https://github.com/glaverghetta/SQL-IDIV-Manual-Analysis/ (2024), retrieved November 18, 2024
31. Laverghetta, G.: SQL-regex. https://github.com/glaverghetta/SQL-regex/ (2024), retrieved November 18, 2024
32. Maheswari, K.G., Anita, R.: An intelligent detection system for sql attacks on web ids in a real-time application. In: Vijayakumar, V., Neelanarayanan, V. (eds.) Proceedings of the 3rd

International Symposium on Big Data and Cloud Computing Challenges (ISBCC – 16'). pp. 93–99. Springer International Publishing, Cham (2016)
33. Make WordPress Core: Escaping table and field names with wpdb::prepare() in wordpress. https://make.wordpress.org/core/2022/10/08/escaping-table-and-field-names-with-wpdbprepare-in-wordpress-6-1/ (2022), retrieved November 17, 2024
34. McWhirter, P.R., Kifayat, K., Shi, Q., Askwith, B.: Sql injection attack classification through the feature extraction of sql query strings using a gap-weighted string subsequence kernel. Journal of Information Security and Applications **40**, 199–216 (2018). https://doi.org/https://doi.org/10.1016/j.jisa.2018.04.001, https://www.sciencedirect.com/science/article/pii/S2214212617303691
35. Microsoft Learn: Dbtype enum. https://learn.microsoft.com/en-us/dotnet/api/system.data.dbtype?view=net-8.0 (2024), retrieved November 3, 2024
36. Microsoft Learn: Sqldbtype enum. https://learn.microsoft.com/en-us/dotnet/api/system.data.sqldbtype?view=net-8.0 (2024), retrieved November 3, 2024
37. MITRE Corporation: CVE-2022-40824. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-40824 (October 2022), retrieved October 15, 2023
38. MITRE Corporation: Cve program mission. https://www.cve.org/ (2024), retrieved November 24, 2024
39. MySQL2: Prepared statements. https://sidorares.github.io/node-mysql2/docs/documentation/prepared-statements# (2024), retrieved November 8, 2024
40. mysqljs: mysql. https://github.com/mysqljs/mysql#readme (2022), retrieved November 8, 2024
41. Nagy, C., Cleve, A.: A static code smell detector for SQL queries embedded in Java code. In: IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM) (2017), https://doi.org/10.1109/SCAM.2017.19
42. National Institute of Standards and Technology: National vulnerability database. https://nvd.nist.gov/ (2024), retrieved November 25, 2024
43. National Vulnerability Database: CVE-2023-2592 detail. https://nvd.nist.gov/vuln/detail/CVE-2023-2592 (June 2023), retrieved November 28, 2024
44. National Vulnerability Database: CVE-2023-7157 detail. https://nvd.nist.gov/vuln/detail/CVE-2023-7157 (December 2023), retrieved November 28, 2024
45. nCrafts: FormCraft - Form Builder. https://wordpress.org/plugins/formcraft-form-builder/ (July 2024), retrieved November 28, 2024
46. Open Web Application Security Project: SQL injection prevention - OWASP cheat sheet series. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet (2018), retrieved October 15, 2023
47. Open Web Application Security Project: OWASP top ten – 2021. https://owasp.org/www-project-top-ten/ (2021), retrieved April 26, 2023
48. Oracle: Interface preparedstatement. https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html (2020), retrieved November 3, 2024
49. Oracle: Enum mysqldbtype. https://dev.mysql.com/doc/dev/connector-net/latest/api/data_api/MySql.Data.MySqlClient.MySqlDbType.html (2024), retrieved November 3, 2024
50. Ray, D., Ligatti, J.: Defining code-injection attacks. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2012), https://doi.org/10.1145/2103656.2103678
51. Rudrastyh, M.: Sanitize early, escape late, always validate. https://rudrastyh.com/wordpress/sanitize-escape-validate.html (2023), retrieved November 17, 2024
52. Son, S., McKinley, K.S., Shmatikov, V.: Diglossia: Detecting code injection attacks with precision and efficiency. In: Proceedings of the ACM SIGSAC Conference on Computer &

Communications Security (CCS) (2013). https://doi.org/10.1145/2508859.2516696, https://doi.org/10.1145/2508859.2516696
53. Spadini, D., Aniche, M., Bacchelli, A.: Pydriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 908–911. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3264598, https://doi.org/10.1145/3236024.3264598
54. sqlmapproject: sqlmap. https://github.com/sqlmapproject/sqlmap (2023), retrieved October 15, 2020
55. Square, Inc.: Overview - OkHTTP. https://square.github.io/okhttp/ (2022), retrieved April 26, 2023
56. Stanciu, A.M., Ciocârlie, H.: Analyzing code security: Approaches and tools for effective review and analysis. In: 2023 International Conference on Electrical, Computer and Energy Technologies (ICECET). pp. 1–6 (2023). https://doi.org/10.1109/ICECET58911.2023.10389326
57. Tang, P., Qiu, W., Huang, Z., Lian, H., Liu, G.: Detection of sql injection based on artificial neural network. Knowledge-Based Systems **190**, 105528 (2020). https://doi.org/https://doi.org/10.1016/j.knosys.2020.105528, https://www.sciencedirect.com/science/article/pii/S0950705120300332
58. The PHP Documentation Group: Mysql improved extension. https://www.php.net/manual/en/book.mysqli.php (2024), retrieved November 7, 2024
59. The PHP Documentation Group: mysqli_stmt::bind_param. https://www.php.net/manual/en/mysqli-stmt.bind-param.php (2024), retrieved November 9, 2024
60. The PHP Documentation Group: Pdo::prepare. https://www.php.net/manual/en/pdo.prepare.php (2024), retrieved November 7, 2024
61. The PHP Documentation Group: Predefined constants. https://www.php.net/manual/en/pdo.constants.php (2024), retrieved November 12, 2024
62. Unruh, T., Shastry, B., Skoruppa, M., Maggi, F., Rieck, K., Seifert, J.P., Yamaguchi, F.: Leveraging flawed tutorials for seeding Large-Scale web vulnerability discovery. In: 11th USENIX Workshop on Offensive Technologies (WOOT 17) (Aug 2017), https://www.usenix.org/conference/woot17/workshop-program/presentation/unruh
63. Verdi, M., Sami, A., Akhondali, J., Khomh, F., Uddin, G., Motlagh, A.K.: An empirical study of c++ vulnerabilities in crowd-sourced code examples. IEEE Transactions on Software Engineering **48**(5), 1497–1514 (2022). https://doi.org/10.1109/TSE.2020.3023664
64. Woolson, R.F., Bean, J.A., Rojas, P.B.: Sample size for case-control studies using Cochran's statistic. Biometrics **42**(4), 927–932 (1986), https://doi.org/10.2307/2530706
65. WordPress: wpdb::esc_like. https://developer.wordpress.org/reference/classes/wpdb/esc_like/ (2023), retrieved April 26, 2023
66. WordPress Developer Resources: class wpdb. https://developer.wordpress.org/reference/classes/wpdb/ (2024), retrieved November 17, 2024
67. WordPress Developer Resources: wpdb::prepare. https://developer.wordpress.org/reference/classes/wpdb/prepare/ (2024), retrieved November 17, 2024
68. WordPress Developer Resources: wpdb::query. https://developer.wordpress.org/reference/classes/wpdb/query/ (2024), retrieved November 26, 2024
69. Yasar, H.: Experiment: Sizing exposed credentials in GitHub public repositories for CI/CD. In: 2018 IEEE Cybersecurity Development (SecDev) (2018), https://doi.org/10.1109/SecDev.2018.00039
70. Zhang, B., Ren, R., Liu, J., Jiang, M., Ren, J., Li, J.: Sqlpsdem: A proxy-based mechanism towards detecting, locating and preventing second-order sql injections. IEEE Transactions on Software Engineering **50**(7), 1807–1826 (2024). https://doi.org/10.1109/TSE.2024.3400404