Large-Scale Analysis of GitHub and CVEs to Determine Prevalence of SQL Concatenations

Kevin Dennis, Bianca Dehaan, Parisa Momeni, Gabriel Laverghetta and Jay Ligatti
Computer Science and Engineering, University of South Florida, Tampa, Florida, U.S.A.

Keywords: Security Metrics, Web Applications, Structured Query Language, Code Injection Attacks.

Abstract: SQL Injection Attacks (SQLIAs) remain one of the top security risks in modern web applications. Vulnerabilities to SQLIAs 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. If an identifier is not sanitized before concatenation, a kind of SQLIA called a SQL Identifier Injection Attack (SQL-IDIA) is possible.

To investigate the prevalence of SQL concatenations in real code, we conducted, 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 and potential SQL-IDIAs.

Our crawler classified 42% of Java, 91% of PHP, and 56% of C# files as constructing SQL statements via concatenation. It further found that 27% of the Java, 6% of the PHP, and 22% of the C# files of these concatenations contain identifiers. Manual analysis indicates that the automated SQL-IDIA classifier achieved an overall accuracy of 93.4%. Further testing suggests approximately 22.7% of web applications may be exploitable via a SQL-IDIA. PHP applications were particularly exploitable at 38% of applications.

1 INTRODUCTION

Injection attacks remain one of the top security risks in modern web applications. The 2021 Open Worldwide Application Security Project Top Ten list (Open Web Application Security Project, 2021) ranked injection attacks in the top three with the second most recorded occurrences. Injection attacks occur when untrusted and unsanitized input is used to generate an output program (Ray and Ligatti, 2012). One of the most common examples of injection attacks are SQL injection attacks (SQLIAs), where untrusted and unsanitized input gets 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 paper also refers to interpolation unless otherwise noted.

SQLIAs can be mitigated using a variety of techniques, with prepared statements, also known as parameterized queries, being the standard defense (Open Web Application Security Project, 2018; Clarke-Salt, 2012). However, modern prepared-statement implementations are incomplete. SQL Identifier Injection Attacks (SQL-IDIAs) (Cetin et al., 2019) are a subset of SQLIAs where the user data is inserted into a portion of the SQL statement reserved for a SQL identifier, such as a table or column name. To our knowledge, no public implementation of prepared statements supports identifier insertions.

This paper investigates the prevalence of SQL concatenations in real code, performing, to our knowledge, the largest analysis of open-source software to date, relying solely on GitHub's code-search application programming interface (API) to identify program source files for security analysis. Our crawler analyzed a total of 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 prevalence and well-established database libraries/frameworks. We also further classified whether the concatenations are into portions of SQL statements reserved for identifiers.

Dennis, K., Dehaan, B., Momeni, P., Laverghetta, G. and Ligatti, J.
Large-Scale Analysis of GitHub and CVEs to Determine Prevalence of SQL Concatenations.
DOI: 10.5220/0012835200003767

In *Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT 2024)*, pages 286-297
ISBN: 978-989-758-709-2; ISSN: 2184-7711

Our automated GitHub crawler and analyzer classified 42% of Java, 91% of PHP, and 56% of C# web-application files as constructing SQL statements via concatenation. It further 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-IDIA classifier achieved an overall accuracy of 93.4%. After confirming the classifier's accuracy, we determined approximately 22.7% of the web applications meet the additional requirements to be exploitable via an SQL-IDIA. PHP applications were particularly exploitable, with 38% of applications being exploitable. The repository owners of these applications were informed of the vulnerabilities.

We also manually analyzed all 1,775 CVE reports of SQLIAs from 2022. We found that 153 (8.6%) of these 1,775 reports are for SQL-IDIAs, providing further evidence that SQL-IDIAs comprise a nontrivial portion of SQLIAs. We therefore recommend that existing implementations of prepared statements expand to cover insertions of identifiers. Previous work has described and analyzed a non-public proof-of-concept implementation of prepared statements with coverage of identifiers (Cetin et al., 2019).

This paper also presents a modification to the original definition of SQL-IDIAs (Cetin et al., 2019).

The definition is improved to allow SQL identifier lists, enabling our classifier to recognize locations reserved for a comma-separated list of identifiers. This new definition is a strict generalization of the original. An additional 658 Java and 174 C# files were correctly classified due to this updated definition.

This paper makes the following contributions:

- an analysis of concatenation in SQL statements, and of SQL-IDIA vulnerabilities, in millions of GitHub files across multiple languages;
- an improved definition of, and classifier for, SQL-IDIAs, capturing an additional 800 potentially vulnerable files on GitHub;
- a manual classification of all SQLIA CVE reports published in 2022, to investigate the prevalence of SQL-IDIAs.

The remainder of the paper is organized as follows: Section 2 presents the necessary background material on SQLIAs and SQL-IDIAs, Section 3 provides a generalized definition of SQL-IDIAs, Section 4 describes the GitHub crawler and SQL-IDIA classifier experiment, Section 5 describes the analysis of CVE reports for SQL-IDIAs, and Section 6 makes closing remarks.

2 BACKGROUND AND RELATED WORK

This section describes previous efforts made to extract data from GitHub, and related work on SQLIAs. Given their prevalence, several papers have focused on SQLIAs, including attempts to classify SQLIAs from GitHub.

2.1 Obtaining Data from GitHub

Several attempts have been made to archive GitHub data, generally with the goal of making the data more accessible. Projects like GHTorrent (Gousios and Spinellis, 2012) and GH Archive (Grigorik, 2023) allow users to download the data set or access the data online. Lean GHTorrent allows users to request data dumps on demand (Gousios et al., 2014) and GH Archive makes their data available as a public data set on Google BigQuery. However, neither of these services offers the data needed to complete the experiment described in this paper; the data available are primarily metadata about the users, projects, and various events. While some useful data can be extracted from commit comments and diffs, the GitHub search-code API provides a larger set of up-to-date files for analysis.

In addition, the GHTorrent service appears to be deprecated, the GHTorrent web page is no longer available, and the once-active GHTorrent Twitter account has not posted since March 2021. The original papers describing the GHTorrent service (Gousios and Spinellis, 2012), however, served as inspiration for automating the crawling process.

An illustration of the general workflow for the GitHub crawler and the classifier is shown in Figure 1. 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.

2.2 SQLIAs and SQL-IDIAs

Applications are vulnerable to SQL Injection Attacks (SQLIAs) when untrusted user input is inserted into SQL statements such that, when passed to the DBMS, the user input is interpreted and executed as SQL code, rather than noncode such as string or numeric literals. In fact, previous work showed that any concatenation of unsanitized input into a SQL statement constitutes a SQLIA vulnerability (Ray and Ligatti, 2012). Typically, such attacks occur when user input is directly concatenated into the query string, but

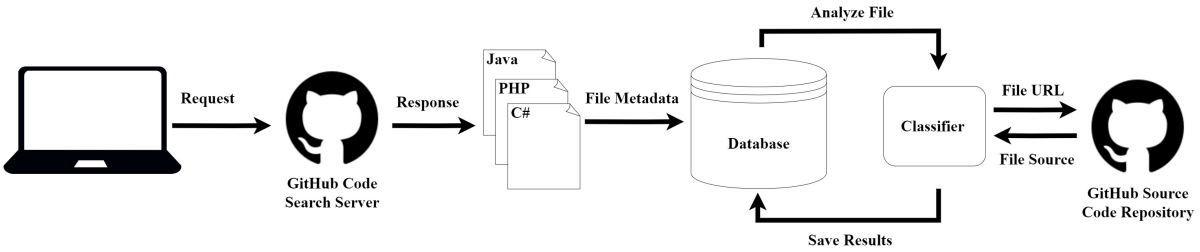

Figure 1: Workflow for the GitHub crawler and classifier.

these attacks are also possible when the SQL statements are built using alternative techniques for concatenating strings, such as with format strings or string interpolation. The basic mechanics of SQLIAs are well understood (Halfond et al., 2006; Ray and Ligatti, 2014).

SQL-IDIAs are a subset of SQLIAs where the untrusted user input is inserted into a location where an identifier is expected (Cetin et al., 2019). Identifiers in SQL include the names of tables, columns, indexes, databases, views, functions, procedures, or triggers. The present paper provides and uses an updated definition of SQL-IDIAs that is based on the original definition of (Cetin et al., 2019). The updated definition, provided in Section 3.2, allows for identifier lists; for example, the query `SELECT a, b FROM c, d`, which retrieves columns `a` and `b` from the Cartesian product of tables `c` and `d`, contains a list of column names and a list of table names. This broader definition of SQL-IDIAs allows for the classification of attacks that would otherwise go unnoticed.

Several solutions have been developed to mitigate SQL injection and code injection attacks in general. Dynamic methods (Ray and Ligatti, 2012) and tools (Halfond and Orso, 2005; Bandhakavi et al., 2007; Son et al., 2013) that attempt to catch injections at runtime have been proposed but may incur large performance overheads and have not seen widespread adoption. Similarly, static methods (Nagy and Cleve, 2017) 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 (Johnson et al., 2013). The two mitigation strategies that have seen the widest acceptance are input sanitization and prepared statements.

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 inserted directly into the output program. The security of the application is now dependent on the filtering process being implemented correctly. Such techniques are difficult to implement as there are a large number of symbols and edge cases that must be accounted for. Using existing well-tested implementations is more reliable, but may introduce their own complexities. For example, the `esc_like` function for escaping a string in a `LIKE` statement in WordPress (WordPress, 2023) might reasonably be assumed to output an escaped string that is safe to insert into SQL code, but the documentation explains it is not and describes how the data must be further sanitized or prepared statements must be used. Other potential downsides include second-order injection attacks, where filtered data is stored in a database and later reused without filtering the data again (Anley, 2002).

Object-Relational Mapping (ORM) libraries abstract the entire process of writing SQL queries into the object-oriented paradigm. For example, a developer using an ORM library might obtain the names of all users by writing code like `Users()->select("name")->execute()`. The ORM library then handles the entire process of constructing the query, connecting to the database, and returning the results. The ORM is then also responsible for constructing the SQL statement securely using input validation or prepared statements, which may not be a reliable assumption (e.g., CVE-2022-4082).

Prepared statements are considered the standard defense for preventing SQLIAs (Open Web Application Security Project, 2018; Clarke-Salt, 2012), preventing injection attacks by providing a clear distinction between code and noncode in the 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, typically a question mark, 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.

However, current implementations do not support placeholders in locations where SQL identifiers

are expected, making them insufficient for preventing SQL-IDIAs (Cetin et al., 2019). If there is a need for dynamic, user-defined identifiers in constructed queries, another mitigation technique must be deployed in addition to prepared statements. A simple scenario where such a need may arise is using a user-provided column to order the returned data by: `"SELECT * FROM users ORDER BY " + orderCol`. The incompleteness of prepared statements is discussed further in Section 3.1.

3 SQL IDENTIFIER INJECTION ATTACKS

This section reviews the theoretical incompleteness of prepared statements and the definition of SQL-IDIAs. Modern prepared-statement implementations, such as the MySQL Java Database Connectivity (JDBC) driver, provide support for only a subset of all potential symbols that may appear in a constructed SQL statement. This section also generalizes the existing definition of SQL-IDIAs (Cetin et al., 2019) to capture strictly more attacks.

3.1 Prepared Statement Incompleteness

In a constructed output program, all symbols fall into exactly one of two categories; a symbol is either a code symbol or a noncode symbol (Ray and Ligatti, 2012). Code symbols are those that define computation. In SQL, code symbols include keywords such as `SELECT`, `FROM`, and `JOIN`, operators such as + and −, and identifiers. Non-code symbols include closed values (Ray and Ligatti, 2012; Ray and Ligatti, 2014), such as string, integer, and date literals.

Prepared-statement implementations such as MySQL JDBC appear to be complete with respect to insertions of complete literals. The JDBC implementation includes support for replacing a placeholder with any of the possible types of SQL literals. However, as discussed in previous sections, only allowing insertions of complete literals is insufficient and limits the expressiveness of programmers. Of particular interest for the present paper is that the JDBC implementation, and all other public DBMS implementations of which we are aware, lack support for defining placeholders for identifiers and replacing placeholders with identifiers. This incompleteness enables SQL-IDIAs.

3.2 SQL-IDIAs with Identifier Lists

The original definition of SQL-IDIAs presented in (Cetin et al., 2019) was limited to applications that concatenate a single identifier into a SQL statement. However, SQL does not have such a limit; some identifiers may appear in a list, including the two most popular identifier types, column and table names. Classifiers based on the original definition would fail to classify such instances as SQL-IDIAs and would instead incorrectly classify them as generic SQLIAs.

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.*

The following items are examples of identifier lists, where ε represents the empty string.

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

The following items are examples of input that would not be considered identifier lists.

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

Definition 2. *An application is vulnerable to a* SQL-IDIA *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.*

Definition 2 has been generalized from (Cetin et al., 2019) to allow for identifier lists rather than just single identifiers. Several vulnerable applications enumerated in the CVE list are not instances of SQL-IDIAs using the narrower, earlier definition but are correctly classified as SQL-IDIAs using this paper's generalized definition.

A SQL-IDIA occurs when a SQL-IDIA-vulnerable application—which would produce a valid SQL statement by concatenating a user-input identifier list into the statement—instead concatenates an input identifier list to produce an invalid SQL statement or concatenates a non-identifier list input.

Definition 3. *A* SQL-IDIA *occurs in a SQL-IDIA-vulnerable 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.*

```
$sql = "SELECT * FROM records ORDER BY
    ↪ id " . userInput;
$stmt = $conn->prepare($sql);
$stmt->execute();
```

Figure 2: A SQL-IDIA-vulnerable application expecting a list of columns as input.

Definition 2 and Definition 3 assume for simplicity that an application accepts a single input. However, both definitions can be straightforwardly generalized to allow for an arbitrary number of inputs.

Figure 2 presents an application program that is vulnerable to a column-name-based SQL-IDIA. The intent is for users to set the sorting order by specifying `ASC` or `DESC`. However, an identifier list may be substituted instead, as described in Definition 2, resulting in a query that orders by multiple columns. The following 3 examples demonstrate how this application may be attacked.

1. An attacker may input `, SLEEP(1000)`. This input is not an identifier list, but the resulting SQL statement is valid and causes the database to sleep for 1000 seconds, a 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.
2. An attacker may input `, SELECT`, which is also not an identifier list but in this case produces a statically invalid SQL statement. Depending on the environment, this attack might leak metadata (e.g., through error messages) or 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 2, 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.

Definition 3 considers all of these examples to be SQL-IDIAs.

Proposition (SQL-IDIA Definition Generalization). *Definition 3 strictly generalizes the definition of SQL-IDIAs in (Cetin et al., 2019).*

Proof. Consider an arbitrary application, A, that is vulnerable to SQL-IDIAs using the definition of (Cetin et al., 2019). 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 (Cetin et al., 2019) also satisfies Definition 3. On the other hand, the application in Figure 2 allows injections of identifier lists but not single identifiers, so it exhibits SQL-IDIAs according to Definition 3 but not according to (Cetin et al., 2019).

4 CONCATENATION ON GitHub

Over 4,762,175 files uploaded to GitHub were analyzed to investigate the prevalence of SQL concatenations in real code. The process starts by finding source files to analyze by querying the GitHub API. The identified source files are then passed to the classifier program, which classifies instances in the source files where concatenation is used to construct SQL queries.

4.1 Crawling GitHub's API

GitHub, as the largest public code hosting service with 94 million users and 85.7 million repositories (State of the Octoverse, 2023), provides an enormous set of data for analysis. GitHub grants all authenticated users the ability to quickly search for specific strings in source files across the uploaded repositories, an impressive feat given the data size. The GitHub API does limit code searches to the first 1000 results, requiring a workaround; other limitations with the API are described in Section 4.4.1.

For each target programming language (Java, PHP, and C#), the most popular database library was selected for analysis, and the GitHub API was used to locate files with calls to the function in that library that executes a SQL command. For example, the Java Database Connectivity (JDBC) API was chosen for Java, and the GitHub API was queried for Java files containing the string `executeQuery`. The popularity of each library was determined by checking the total number of results reported by the GitHub API. GitHub reported about 3.6 million entries for JDBC.

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".

4.2 SQL Classifier

After identifying files for classification using the GitHub API, the classifier program downloads and analyzes the files to find potential misuse of concatenation in the construction of SQL statements. The classifier sources relevant code files from GitHub and determines the usage of prepared statements or concatenation in each file using a number of regular expressions. For example, the following PHP code contains string interpolation in a SQL statement via the `$table` variable.

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

As the classifier is primarily focused on identifying SQL statements inside a source file's string literals, the classifier has been designed to support new languages without changing the underlying classifier program. The classifier has abstracted language-level identifiers or symbols from the regular expressions, allowing for these to be dynamically changed depending on the source file's language. Some examples of the abstracted features include identifier naming requirements (e.g., PHP requires variables to start with a dollar sign) and the various concatenation symbols used by different languages.

The classifier program first identifies all instances where the file constructs SQL code and then classifies the file into one of four categories: none, hardcoded, string concatenation, or string interpolation. The "none" classification means that the file contained no SQL statements, "hardcoded" means all SQL statements were hard coded 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 based on Definition 3 (and not single identifiers). 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).

4.3 GitHub 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 1. These files

Table 1: Files and projects reviewed per language.

	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**

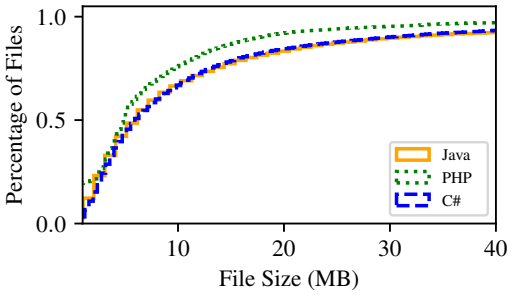

Figure 3: Cumulative percentage of files by size.

were spread across a total of 944,316 projects on GitHub. Not all of the files obtained were unique. By comparing the hashes of the obtained files, duplicate files were identified and ignored. To avoid skewing the analysis, all of the results presented in this paper are based on the data set of unique files containing SQL.

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 3. The largest file obtained was a one-gigabyte Java file. However, the graph shows that the vast majority of files obtained were under 40 MB in size, with about 95% of files appearing below 40 MB for each language. The remaining 5% was scattered haphazardly between 40 MB and 1 GB. Thus, the graphs presented in this paper are restricted to under 40 MB to prevent them from being skewed by these 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 incidence where an identifier was concatenated or interpolated during the construction of a SQL statement. Column and table names were the most common identifiers. Table 2 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. Ideally, this table would include the number of files that use a prepared-statement implementation, however, we found that it was typical for such libraries to be called, but not utilized (i.e., no placeholders were

Table 2: Concatenation in SQL statements by location in unique files.

	Any non-identifier 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 3: Statistics of the unique files analyzed.

	% 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%**

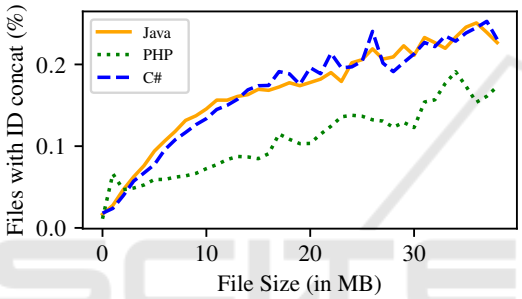

Figure 4: Percentage of unique files with SQL identifier concatenations by file size.

used and data was appended using concatenation).

Table 3 details the statistics of the unique files analyzed. The classifier classified 42% of Java, 91% of PHP, and 56% of C# web-application files as constructing SQL statements via concatenation. It further found that 27% of the Java, 6% of the PHP, and 22% of the C# files that concatenate to construct SQL statements concatenate identifiers.

Files classified as having concatenation were also sorted by file size to determine whether there is a correlation between file size and the likelihood of concatenation occurring. Figure 4 presents the results.

4.4 Discussion

The obtained results demonstrate that SQL identifiers make up a small, but significant, portion of all SQL misconstructions using concatenation. While the majority of identifiers are hardcoded into the string, the number of concatenated identifiers still presents a potential risk for SQL-IDIAs. Table identifiers were the most commonly concatenated, followed by column identifiers.

String interpolation was almost nonexistent in Java programs as it is not supported natively. C# and PHP had a larger number of instances of string interpolation, as both languages support it natively. String interpolation is common in PHP, with over 83% of files utilizing it to construct their SQL statements. PHP had the highest concatenation rate which is reflected in the CVE analysis in Section 5, where the majority of vulnerability reports were observed to be WordPress applications.

There seems to be a strong correlation between the size of files and the percentage of files that concatenate an identifier. This is likely a result of larger 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 the updated SQL definition in Section 3.2 (Definition 3). All of these files concatenated values into a location reserved for a SQL identifier list. These files would not have been classified correctly without the updated definition.

4.4.1 Limitations

The amount of files that can be obtained is limited 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, but is offset by the large amount of available data.

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 analysis in Section 4.5, this issue does not seem to be significant in the context of SQL-IDIAs, as developers appear to largely follow predictable coding patterns and behaviors.

Table 4: Results of manual analysis of randomly sampled GitHub files.

	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
Aggregate	**1,155**	**941**	**44**	**138**	**32**	**0.242**	**0.033**	**0.955**	**0.934**

4.5 Classifier Verification

As with all other static analyzers, the classifier is neither sound nor complete, as programmers can be quite creative in how they construct their SQL statements. A random sampling of the data was manually verified to determine the accuracy of the classifier. Each language was verified independently and the ideal sample size for each language subset was determined to be 385 for a precision level of 95% using Cochran's formula (Woolson et al., 1986). MySQL's `RAND` function was used to randomly select the files for analysis. As no other classifier for SQL-IDIAs 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 output programs in the file corresponded with the flagged results. For example, if the classifier reported that a file contained string interpolation of a column identifier, but no string interpolation had occurred, this would be a false positive. The classifier exhibited a false negative when it failed to detect concatenation in an output SQL program that was located in the file.

While observing the accuracy of the classifier, the files were also reviewed to determine whether the application was vulnerable to a SQL-IDIA. In order for the application to be exploited, the concatenated value must be sourced from user input. This was evaluated separately from the classifier accuracy. From this single-file analysis, about half of the files can be determined to use obfuscated, hardcoded values or employ input sanitization. The remaining files concatenate values that originate from other source files or from user input in the analyzed file and thus may be vulnerable.

Based on this analysis, the classifier had an overall precision of 95.5% and an overall accuracy of 93.4%. False positives mostly arose due to SQL-like statements in comments, logging, or error messages. False negatives 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 4.

During analysis, we observed several programs with comments about their inability to use identifiers with prepared statements. They attempted to overcome this limitation by escaping special characters in the identifiers manually. This practice is often not sufficient for preventing injection attacks (Cetin et al., 2019), and programmers who use prepared statements may not be familiar with sanitization APIs.

4.6 Vulnerability Exploitation

With the classifier having been verified to exhibit satisfactory accuracy, we tested whether the potentially unsafe code could really be exploited; the unsafe code is only exploitable if the output SQL code can be manipulated by the attacker and is not dead code. That is, the concatenated values must be derived from user input without proper validation, and the code must be reachable during normal execution. While both static and dynamic tools exist to detect SQLIAs more reliably, these tools cannot reliably detect SQL-IDIAs; an example of sqlmap (sqlmapproject, 2023) (an automated SQLIA detection and exploitation tool) failing to exploit a SQL-IDIA vulnerable application is shown later in this section. To determine how many of these identified applications may be exploitable, a subset of the applications manually verified were installed and tested. For all repositories determined to be exploitable, their owners were notified of the vulnerabilities.

The following assumptions were made: 1) all databases/tables that are referenced in the code exist and contain 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 file chosen as part of the random analysis is considered. Projects that did not 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 these 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. The application was marked as "Exploitable" if SQL code

Table 5: Types of applications analyzed.

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**
Web	4	2	34	40
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**

could be injected and malicious behavior observed.

Only web applications were considered; a vulnerable serverless GUI or text application has less value because the exploited database is on the user's machine. The breakdown of applications by language, interface type, and purpose is shown in Table 5. The other interface category includes libraries/frameworks, client-server apps, or build systems. All PHP applications were web applications. The purpose category differentiates applications that were student projects or tutorials. Relevant markers for being classified as a student application included referencing a course, grade, or rubric directly or an assignment directory structure (e.g., a folder named "Assignment1"). Tutorial/beginner code consisted of hello-world type programs or other obvious references (such as the repository owner being a tutorial site).

A total of 152 applications were inspected for exploitation. Of the 152, 50 were Java applications, 40 were PHP applications, and 62 were C# applications. Only a small number of applications were student projects or tutorials. All of the PHP applications were web applications, while only 21 Java and 27 C# projects were web applications. Of the web applications, there were a total of 20 SQL-IDIA-vulnerable applications that were confirmed to be exploitable: 4 out of 21 Java (19%), 15 out of 40 PHP (38%), and 1 out of 27 C# (4%). Only 2 of the exploitable applications were student programs (1 PHP and 1 C#); the others all appeared to serve a more professional purpose. A summary of the vulnerable applications grouped by the vulnerable identifier type is shown in Table 6. Note that the total numbers will not sum to the number of applications because an application may include a combination of identifiers. Multiple instances of a single type in an application were counted once. These numbers are a lower bound, because only the randomly-chosen file was considered; several applications were vulnerable in other files.

Table 6: SQL-IDIA-exploitable applications by Identifier Type.

	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**

Column identifiers used in `ORDER BY` statements were the most likely to be vulnerable, with 12 of the 20 SQL-IDIA-vulnerable applications specifically containing an `ORDER BY` concatenation vulnerability.

While the focus was on exploiting SQL-IDIA-vulnerable 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 SQLIAs were not a focus, these applications were only discovered passively and because they were very obvious.

A total of 13 applications were not exploitable because the identified concatenation occurred in dead code. The functions concatenated an argument into a SQL output program but were not called. Most of these functions were alternative queries sorting data using the `ORDER BY` statement. For example, a forum application allowed finer user sorting, but the search interface was not yet implemented. If used without a mitigation technique, the functions would be exploitable. Furthermore, 2 applications exported non-sanitizing string libraries that client applications could use incorrectly (by assuming the libraries sanitize).

Combining these categories, 60 exploitable and problematic applications were identified out of 152 (20 SQL-IDIAs, 25 other SQLIAs, 13 dead-code concatenations, and 2 non-sanitizing libraries).

Figure 5 demonstrates a SQL output program for an exploitable PHP application that could not be detected using sqlmap (sqlmapproject, 2023). The `$c` variable is user input interpolated directly into the output SQL program. The intended content of this `$c` variable should be "users" or "crew", querying either the customers or employees table using the same code. Any subquery injected into this location would not be syntactically valid without a table alias, and sqlmap does not include this technique in a scan. 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).

```
$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.

Figure 5: One of the exploited applications that could not be detected using sqlmap (sqlmapproject, 2023).

5 SQL-IDIAs IN CVEs

MITRE's Common Vulnerabilities and Exposures (CVE) List (MITRE Corporation, 2020) tracks publicly known cybersecurity vulnerabilities. Of the 200,946 CVE entries added from 1999 to 2023, 11,766 (5.9%) were SQLIAs, making it the sixth most prevalent vulnerability type as ranked on the CVE Details site after code execution (23.1%), denial of service (14.9%), overflow (11.8%), cross-site scripting (12.9%), and information gain (6.8%) (CVE Details, 2019). In 2022, 1,789 SQLIA entries were added, making up 7.1% of the 25,227 vulnerabilities reported that year. This is the largest recorded number of SQLIAs in one year, beating the previous record of 1,101 in 2008 by a large margin. It also more than doubles the 741 reported in 2021.

To determine the prevalence of SQL-IDIAs, we analyzed 1,775 SQLIA CVEs by hand. Note this number slightly differs from the overall mentioned previously, as we analyzed all CVEs published (not reported) in 2022. Of the 1,775 SQLIAs published in 2022, 1,507 were also reported in 2022; the remaining 268 were published in 2022 but reported earlier. The publication date was chosen as it is a static set of CVEs. More CVEs first reported in 2022 may be published much later, making that number unreliable. For example, one of the vulnerabilities published in 2022 has a CVE label from 2013.

In our analysis, SQLIAs are considered SQL-IDIAs when they satisfy Definition 3. Such a classification cannot always be made from the vulnerability description alone as they rarely provide sufficient technical detail. To determine whether the CVE represents a SQL-IDIA, the CVE must reference source code or a proof of concept (PoC) attack. We therefore excluded the 15% of 2022 SQLIA CVEs that lacked reference source code or a PoC. If source code is available, the classification can be determined by finding the injection point and reviewing the query.

5.1 SQL-IDIAs in CVEs

To demonstrate how real SQL-IDIAs have appeared in applications and to demonstrate how they can be classified, this subsection describes two example SQL-IDIAs found in vulnerabilities reported to the CVE List. To demonstrate the difference between classifying with source code and with a PoC, the first example, CVE-2020-8520, contains a PoC and the source code. The second example, CVE-2020-9268, only references a PoC. These examples are from 2020, and not part of the data set; the two were part of our training set for the researcher performing the manual analysis.

CVE-2020-8520 is for a jQuery Datatables tutorial by PHPZag using PHP and MySQL. All of the source code is available for download and described in detail in a blog post (PHPZag Team, 2023). Three SQLIAs against this program have been discovered and included in the CVE list, one of which is a SQL-IDIA. The application, uses MySQLi, a PHP extension for interfacing with MySQL databases with prepared statements, but these features are not used.

The application creates a table named `live_records`. Line 29 of the file `Records.php`, which retrieves the data stored in the `live_records` table based on the user's request, contains the following PHP code:

```
$sqlQuery.='ORDER BY ' . $_POST['order
    ↪ ']['0']['column'] . ' ' . $_POST
    ↪ ['order']['0']['dir'] . ' ';
```

This PHP statement appends user input (via the global variable `$_POST`) directly to an `ORDER BY` statement. Clearly, untrusted input can be injected into this query, and there exists an identifier list (specifically any combination of `id`, `name`, `skills`, `address`, `designation`, or `age`) such that concatenating that list into the query creates a valid SQL statement. The CVE can thus be classified as a SQL-IDIA.

CVE-2020-9268 details a vulnerability found in an online tool. This app, SoPlanning, provides services for planning teamwork periods. For this example, only the PoC attack linked directly in the CVE is considered. An automated tool, sqlmap (sqlmapproject, 2023), was used to discover and exploit the vulnerability. Sqlmap is used to analyze the `by` GET parameter in the following URL (which is partly in French):

```
/soplanning/www/projets.php?order=
    ↪ nom_createur&by=ASC
```

Based on the names of these parameters, it appears that injection on the `by` GET parameter is a SQL-IDIA

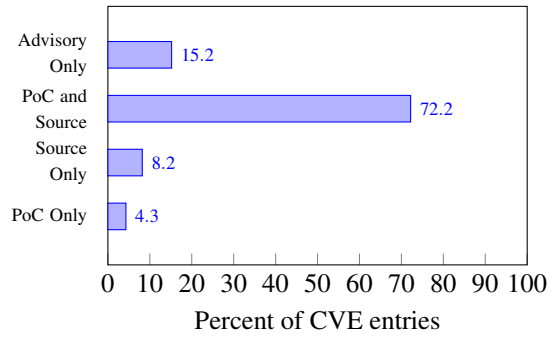

Figure 6: Resources available for classification in 1775 SQLIA CVE entries from 2022.

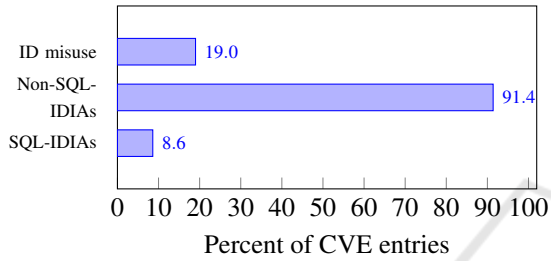

Figure 7: Classified SQLIA CVE entries for 2022.

in an `ORDER BY` statement. In addition, the valid value `ASC` appears in `ORDER BY` statements, and one of the sqlmap suggestions is for an `ORDER BY` statement. As `ORDER BY` statements must be followed by an identifier list, this CVE is vulnerable to a SQL-IDIA.

5.2 CVE Results

Of the 1,775 SQLIA CVEs published in 2022, 72.2% had a PoC and source code available, 8.2% had only the source code available, 4.3% had a PoC only, and 15.2% had only an advisory. Figure 6 shows the available references in SQLIA CVEs from 2022. Figure 7 presents the results of the 1505 classifiable CVEs, classifying them as SQL-IDIAs or SQLIAs. SQL-IDIAs make up 8.6% of classifiable SQLIAs and at least 6.8% of all the SQLIA CVEs published. About 19% of those vulnerable projects constructed SQL statements incorrectly using identifiers.

5.3 Discussion

Although SQL-IDIAs are not the majority of reports, all of the other CVE SQLIAs analyzed could have been prevented using prepared statements. In many cases, the vulnerable application used a library providing prepared statements but did not employ them. CVE-2020-8520, the training example described in detail, does not take advantage of prepared statements using MySQLi despite passing the query to the prepare function that takes a parameterized SQL statement. Thus, the reported SQLIAs that are not SQL-IDIAs are preventable using existing technologies that are readily available.

Developers not employing readily available prepared statements was a common issue across CVEs. About 19.0% of the CVEs had source code that concatenated an identifier elsewhere in the code (excluding the vulnerable location reported in the CVE), with the majority using prepared statements correctly in other locations. Vulnerabilities that were observed in large code bases were often caused by a single missing use of prepared statements; making concatenation poor practice by supporting SQL identifiers in prepared statements may help reduce such occurrences.

The percentage of SQL-IDIA vulnerabilities found in the universe of classifiable SQLIA vulnerabilities (8.6%) is less than the percentage of identifier concatenations found in the universe of GitHub SQL concatenations (14%) as described in Section 4.3. Future work might explore such gaps 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.

6 CONCLUSIONS

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—over a quarter of a million—web-application files vulnerable to SQL-IDIAs. Of these 262K files, 62K are likely to meet all of the additional requirements to be exploited in practice.

These results were not equally distributed across the three analyzed languages. PHP applications were particularly exploitable at 38% of applications but also had the overall lowest percentage of identifier concatenations at 6%. While this is likely partially due to the very high number of overall concatenations in PHP, with 91% of files concatenating SQL values, SQL-IDIAs are quite a concern for PHP due to the relatively high opportunities for exploiting concatenations in practice. Compared to Java and C#, we hypothesize PHP's vulnerability is largely due to the common use of string interpolation. However, all

three languages remain susceptible to SQL-IDIAs and would benefit from support for identifiers in prepared statements.

Based on these analyses, we recommend that existing prepared-statement implementations expand to cover insertions of identifiers. For example, previous work has described and analyzed a non-public proof-of-concept implementation of prepared statements with coverage of identifiers (Cetin et al., 2019).

Potential directions for future work include expanding a large-scale open-source DBMS such as MySQL to include support for identifiers in prepared statements, and incorporating these additions into front-end APIs for commonly used languages.

REFERENCES

Anley, C. (2002). Advanced SQL injection in SQL server applications. Technical report. https://crypto.stanford.edu/cs155old/cs155-spring11/papers/sql_injection.pdf.

Bandhakavi, S., Bisht, P., Madhusudan, P., and Venkatakrishnan, V. N. (2007). CANDID: Preventing SQL injection attacks using dynamic candidate evaluations. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/1315245.1315249.

Cetin, C., Goldgof, D., and Ligatti, J. (2019). SQL-identifier injection attacks. In *IEEE Conference on Communications and Network Security (CNS)*. https://doi.org/10.1109/CNS.2019.8802743.

Clarke-Salt, J. (2012). *SQL Injection Attacks and Defense*. Elsevier, 2nd edition.

CVE Details (2019). Vulnerability distribution of CVE security vulnerabilities by types. https://www.cvedetails.com/vulnerabilities-by-types.php. Retrieved October 15, 2023.

Gousios, G. and Spinellis, D. (2012). GHTorrent: GitHub's data from a firehose. In *IEEE Working Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1109/MSR.2012.6224294.

Gousios, G., Vasilescu, B., Serebrenik, A., and Zaidman, A. (2014). Lean GHTorrent: GitHub data on demand. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1145/2597073.2597126.

Grigorik, I. (2023). GH Archive. https://www.gharchive.org. Retrieved April 26, 2023.

Halfond, W. G. J. and Orso, A. (2005). AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1145/1101908.1101935.

Halfond, W. G. J., Viegas, J., and Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*, volume 1.

Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In *Proceedings of the International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2013.6606613.

MITRE Corporation (2020). CVE - common vulnerabilities and exposures. https://cve.mitre.org/. Retrieved October 15, 2023.

Nagy, C. and Cleve, A. (2017). A static code smell detector for SQL queries embedded in Java code. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. https://doi.org/10.1109/SCAM.2017.19.

Open Web Application Security Project (2018). SQL injection prevention - OWASP cheat sheet series. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet. Retrieved October 15, 2023.

Open Web Application Security Project (2021). OWASP top ten – 2021. https://owasp.org/www-project-top-ten/. Retrieved April 26, 2023.

PHPZag Team (2023). Live add edit delete datatables records with Ajax, PHP and MySQL. https://www.phpzag.com/live-add-edit-delete-datatables-records-with-ajax-php-mysql/. Retrieved October 15, 2023.

Ray, D. and Ligatti, J. (2012). Defining code-injection attacks. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2103656.2103678.

Ray, D. and Ligatti, J. (2014). Defining injection attacks. In *Proceedings of the International Information Security Conference*. https://doi.org/10.1007/978-3-319-13257-0_26.

Son, S., McKinley, K. S., and Shmatikov, V. (2013). Diglossia: Detecting code injection attacks with precision and efficiency. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*. https://doi.org/10.1145/2508859.2516696.

sqlmapproject (2023). sqlmap. https://github.com/sqlmapproject/sqlmap. Retrieved October 15, 2020.

State of the Octoverse (2023). The global developer community. https://octoverse.github.com/2022/developer-community. Retrieved April 26, 2023.

Woolson, R. F., Bean, J. A., and Rojas, P. B. (1986). Sample size for case-control studies using Cochran's statistic. *Biometrics*, 42(4):927–932. https://doi.org/10.2307/2530706.

WordPress (2023). wpdb::esc_like. https://developer.wordpress.org/reference/classes/wpdb/esc_like/. Retrieved April 26, 2023.