An Empirical Study on the Security of SQLite Engines

Xinrong Lin   Baojian Hua
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Study, University of Science and Technology of China
lxr1210@mail.ustc.edu.cn   bjhua@ustc.edu.cn

***Abstract*—SQLite is an embedded, fast, self-contained database widely used in diverse application fields, including web browsers, operating systems, embedded devices, etc. Unfortunately, existing studies on SQLite security mostly concentrate on the security of SQL injection, data encryption, data integrity and access control, but pay little attention to the security of SQLite native code. Without such knowledge of how vulnerabilities are distributed and evolved in the native code of SQLite, SQL engine developers, analysis tool builders, and storage security researchers miss opportunities for further improvements and studies.**

This paper presents the *first* and most *comprehensive* empirical study on memory-related vulnerability detection, rectification and evolution of the SQLite native code. To this end, we first designed and implemented a novel software prototype dubbed SQLSCAN, which leveraged off-the-shelf and state-of-the-art vulnerability analysis tools, as well as syntax-directed heuristics to provide rectification suggestions automatically. We applied SQLSCAN to scan the source code of the latest SQLite (version 3.37) and 9 other versions (3.6 to 3.33), which consists of 2,666,920 lines of source code in total. The empirical results give relevant findings: SQLite is still vulnerable, with 96 vulnerabilities in version 3.37, including 50 null dereferences, 12 uninitialized variables, 16 dead stores, 8 resource leaks and 10 memory leaks. The overall quality of SQLite remained stable during evolution, with vulnerabilities per thousand lines (VPTL) of approximately 0.06. SQLSCAN is effective in automatic vulnerability rectification: 92.00% vulnerabilities from SQLite 3.37 can be rectified successfully. Vulnerabilities in the SQLite native code had persisted for a long time: it takes an average of 1.5 years for a memory-related vulnerability to be discovered and fixed. We suggest that: 1) storage security researchers should pay attention to vulnerabilities in SQLite native code; 2) SQL engine developers can take advantage of the existing popular analysis tools to discover vulnerabilities early; 3) security tool builders should improve their tools to avoid the high rate of false positives which will cause confusion and distrust. We believe these findings would prompt more secure SQL engines and more reliable analysis tools.

***Index Terms*—Empirical study, SQLite, Vulnerability detection, Vulnerability evolution**

I. Introduction

SQLite is an embedded, fast, self-contained database widely used in diverse application fields, including middleware [1] [2], Web browsers [3] [4] [5], operating systems [6] [7], embedded devices [8] [9] [10], data analysis [11] [12], among others. For example, SQLite has been used in more than 4.0 billion smartphones, each of which contains hundreds of SQLite database files. As a result, the usage of SQLite has exceeded the sum of all other database engines [13]. Moreover, according to the latest survey from Stack Overflow [14], over 32% of the survey respondents use SQLite as their primary database. Given SQLite's widespread adoption and use, it should be safe and high-reliable.

However, existing studies have demonstrated the existence of vulnerabilities in SQLite, which may seriously affect storage security [15]. For instance, Magellan [16] [17] [18] and Magellan 2.0 [19] [20] [21] [22] [23] reported a series of vulnerabilities detected in SQLite. These vulnerabilities could cause remote code execution, program memory leaking and even program crashes [24] [25], which affected millions of APPs, PCs and IoT devices with SQLite deployed. For any exploitable vulnerabilities in SQLite, a huge number of users and devices will be affected, which can lead to immeasurable losses.

A significant amount of studies have focused on SQL security recently [26] [27] [28] [29], as TABLE I summarized. Existing research efforts mostly focus on SQL injection, data encryption, data integrity and access control, but just assume the implementations of SQL engines are correct and trustworthy. However, whether such an assumption truly holds is still questionable, for 3 key reasons: 1) SQLite is implemented using C, a low-level language notorious for introducing subtle bugs, which may defeat its security guarantees; 2) it is intrinsically difficult to check memory-related vulnerabilities [30] [31] [32] [33] [34]; and 3) SQLite has no built-in security mechanism to protect the database, but relies on surrounding environments, such as the operating system, to protect the database content, for instance, Android provides a user ID (UID)-based access control policy to isolate SQLite databases from other applications [35]. Thus, the understanding about the security of SQLite native code is still lacking.

To bridge this knowledge gap, we focus on the security of SQLite. Several key questions remain unanswered: *Is the latest SQLite vulnerable? If so, what are these vulnerabilities? How do vulnerabilities evolve in different versions of SQLite? How accurate are existing vulnerability analysis tools for detecting large-scale code such as SQLite? How difficult and costly is it to rectify timely and automatically vulnerabilities in SQLite?* Without such knowledge, SQLite developers cannot benefit from the state-of-the-art security tools due to distrust; security tool builders may miss opportunities to improve their tools

TABLE I
A SUMMARY OF REPRESENTATIVE RESEARCH EFFORTS ON SQL SECURITY IN THE PAST 6 YEARS

Research Fields	Representative Research Efforts
SQL Injection	Kim D. [36], Gu H. [37], Katole R A. [38], Thomé J. [39], Dubey R. [40], Das D. [41], Gupta H. [42], Lawal M A. [43], Sheykhkanloo N M. [44], Joshi P N. [45]
Data Encryption	Wang Y. [46] Obradovic N. [7], Zhao A. [47], Antonopoulos P. [48], Kamara S. [49], Safaryan O A. [50] Liu G. [51], Vinayagamurthy D. [52]
Data Integrity	Oriol Hilari X. [53], Antonopoulos P. [54], Haraty R A. [55], Shahriar H. [56]
Access Control	Mutti S. [57], Dave J. [58], Guarnieri M. [59], Sarfraz M I. [60]

due to incorrect assumptions; and researchers may overlook the potential threats due to a lack of attention to the security of the SQLite native code.

Our work. To answer the above key questions, this paper presents, to the best of our knowledge, the *first* and most *comprehensive* empirical study on vulnerability detection, rectification and evolution of the SQLite native code. We employ both quantitative and qualitative methods to answer the research questions. This study is performed in several steps. First, to investigate vulnerabilities in SQLite, we designed and implemented a novel software prototype dubbed SQLSCAN. SQLSCAN leveraged off-the-shelf and state-of-the-art vulnerability analysis tools, as well as syntax-directed heuristics to provide rectification suggestions automatically.

Second, to investigate common vulnerabilities in SQLites, we first applied SQLSCAN to the latest SQLite (version 3.37), which consists of 352,678 lines of C code. To study the evolution of vulnerabilities, we applied SQLSCAN to 10 different versions of SQLite (from 3.6 to 3.37), with 2,666,920 lines of source code in total.

Finally, to study automatic vulnerability rectification, we utilize a simple, yet effective syntax-directed heuristics to provide minimally intrusive security rectification suggestions automatically.

The empirical results give relevant findings and insights, such as: 1) the latest SQLite contains considerable vulnerabilities, with 96 vulnerabilities being identified (version 3.37), including 50 null dereferences, 12 uninitialized variables, 16 dead stores, 7 resource leaks, and 10 memory leaks; 2) the 6 of 8 analysis tools we used still have many false positives thus need further improvements; 3) overall quality of the code remained relatively stable during the evolution of SQLite: although the SQLite grew significantly, from 125,569 lines of code in version 3.6 to 352,678 lines of code in version 3.37, the vulnerabilities per thousand lines (VPTL) fluctuated around 0.06; 4) automatic vulnerability rectification is effective: 23 out of 25 (92.00%) vulnerabilities can be rectified automatically by a simple, yet effective syntax-directed heuristics, which is similar to the way the SQLite development team fixes vulnerabilities; 5) vulnerabilities in SQLite native code had persisted for a long time: it takes an average of 1.5 years for a vulnerability to be discovered and fixed.

Last but not least, SQLSCAN is efficient in vulnerability rectification: it takes 591.54 milliseconds to automatically fix 22 vulnerabilities (26.89 milliseconds per vulnerability) for the latest SQLite version 3.37.

Our findings and tool can benefit several audiences. Among others, they 1) provide feedback to SQLite developers to improve the security of native code; 2) provide suggestions to analysis tool builders to improve their tools by suppressing duplicate vulnerabilities and becoming more context-sensitive; and 3) significantly expand the scope of existing studies of storage security for researchers.

Contributions. This work represents the *first* step toward a comprehensive empirical study of SQLite native code. To summarize, this work makes the following contributions:

- **Empirical study and tools.** To the best of our knowledge, we presented the first and most comprehensive empirical study on vulnerability detection, rectification and evolution of the SQLite native code, with a novel software prototype we created dubbed SQLSCAN.
- **Findings and insights.** We presented empirical results, relevant findings based on the analysis of the empirical results, as well as implications for these results, future challenges and research opportunities.
- **Open source.** We make our tool and empirical data publicly available in the interest of open science at `https://doi.org/10.5281/zenodo.7016753`.

Outline. The rest of this paper is organized as follows. Section II presents an overview of the background and motivation for this work. Section III presents the methodology for the study. Section IV presents the experiments we performed, and the answers to the research questions based on the empirical results. Section V and VI discuss implications for this work and threats to validity, respectively. Section VII discusses the related work and Section VIII concludes.

II. BACKGROUND AND MOTIVATION

To be self-contained, in this section, we present necessary background information on storage security (Section II-A), and SQLite (Section II-B), respectively.

A. Storage Security

Existing research on the local storage security of mobile devices mainly focuses on SQL injection, data encryption, data integrity and access control [36] [61] [62].

SQL injection [63] is one of the most common threats to database engines. Attackers add SQL statements to the application input form, to access resources or change data stored in the database. SQL injection attacks can damage databases by various means, which lead to data loss or misuse by unauthorized parties.

Data encryption is usually used to ensure the confidentiality of data. Some DBMSs simply store data in database files: thus, attackers can access the database files by reading or writing the data. Taking SQLite as an example, although it provides extensions to encrypt database files, vulnerabilities such as improper encryption keys management still leave an opportunity for attackers to steal data, leading to compromised storage security.

Data integrity guarantees validity, reliability and consistency of data. It ensures that data is only modified with explicit user intentions. Attacks on data integrity can lead to data leakage, data modification, among others.

Access control of the DBMS grants different permissions to different users to restrict the user's capability to access data resources. At present, some large DBMSs, such as MySQL and SQL Server, use a role-based access control mechanism where different roles have different access rights. Unfortunately, such access control mechanisms are missing in SQLite.

However, despite all the aforementioned research progress, the study on the security of native code in the database system itself is still lacking, and existing studies just assume that the underlying implementations of database engines are correct and trustworthy. Whether such an assumption truly holds is (sadly) still unknown.

B. *SQLite*

SQLite [13] is a lightweight, fast, self-contained, serverless and full-featured SQL database engine implemented using the C language, with bindings for other mainstream programming languages such as C/C++, Python, or Java, etc. SQLite is the most widely used databases in iOS, Android, Linux and Windows, for its key advantages of small size and ease of use [64].

SQLite is a complex software implemented in the C language. For instance, the latest SQLite 3.37 consists of 352,678 lines of native C code. Moreover, SQLite utilizes a special build system called "amalgamation": over 100 separate source files are concatenated into a single large file `sqlite3.c`, which contains everything an application needs to embed SQLite. This design choice makes it easier to deploy SQLite and to enable better compiler interprocedure and inline optimization for faster execution [65].

As SQLite is implemented in C, it is vulnerable to security issues common to most C programs [30] [31] [32] [33] [34], such as buffer overflows, integer overflows, heap overflows, stack overflows, double free, etc. However, it should be noted that we are not criticizing C language here. Indeed, C is an indispensable and de facto system programming language to develop infrastructures such as SQLite due to its extreme efficiency and flexibility. Unfortunately, C is also (arguably more) flexible in introducing subtle vulnerabilities. Worse yet, for important infrastructures such as SQLite, even a single bug can defeat its guarantee of security and lead to serious consequences, as demonstrated by the recent Log4j [66] exposure.

To this end, it is important to guarantee the security and trustworthiness of SQLite native code by leveraging state-of-the-art security techniques and tools.

III. Methodology

In this section, we present the methodology to conduct the empirical study. It is challenging to perform an empirical study for large projects such as SQLite, for two key reasons: 1) *automation*: the study should be fully automated, otherwise it is difficult if not impossible to study a large code base consisting of up to hundreds of thousands lines of source code; manual code inspection is only required to confirm the vulnerability; and 2) *scalability*: the analysis should be able to work for any SQL engines implemented using C with diverse structures instead of specific ones.

To this end, we designed and implemented a novel software prototype dubbed SQLScan to analyze and rectify vulnerabilities in SQLite, which is supplemented by human efforts to inspect code. We first discuss the architecture of SQLScan (Section III-A), then describe the design and implementation details of the frontend (Section III-B), the vulnerability analysis (Section III-C), the purification (Section III-D), the automatic rectification (Section III-E), the validation (Section III-F), and the rectified program generation (Section III-G), respectively.

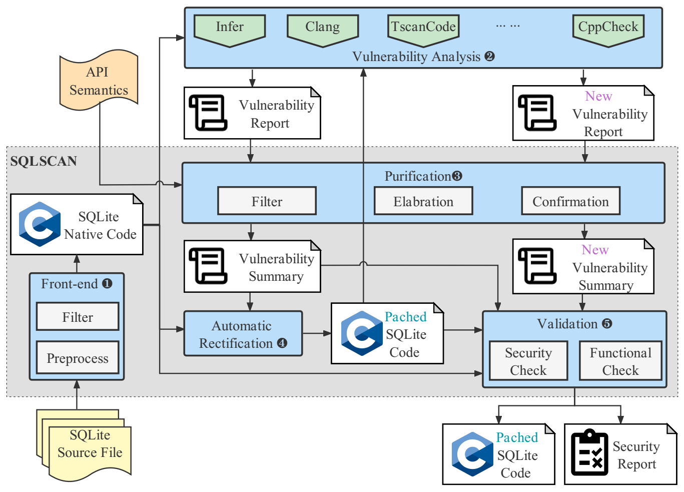

Fig. 1. SQLScan Architecture

A. *The Architecture*

We have two principles guiding the construction of SQLScan framework. First, the architecture of SQLScan should be easily extended to support the analysis and rectification of different versions of SQLite. Second, the architecture of SQLScan should be modular: thus, each module can be easily extended or replaced separately.

Based on these design principles, we present the architecture of the SQLScan in Fig. 1. SQLScan consists of several key modules. First, the frontend module (❶) takes as input the corresponding SQLite source files, then processes the SQLite source code and outputs the SQLite native code. Second, the vulnerability analysis module (❷) takes as input the generated native code from SQLite, analyzes this code, and outputs an original vulnerability report. Third, the purification module (❸) takes as input both the original vulnerability report and the specification of the SQLite C/C++ Interface, purifies the report and generates a confirmed vulnerability summary. Fourth, the automatic rectification module (❹) takes as input both the original SQLite native code and the vulnerability summary, rectifies the native code of SQLite according to the vulnerability summary and generates patched SQLite source code. Finally, the validation module (❺) takes as input the original SQLite source code, the patched SQLite source code, and a new vulnerability summary generated by the vulnerability analysis module for the patched SQLite source code, to validate the effectiveness of rectification and generates final patched code, along with a rectification report.

In the following sections, we discuss the design and implementation of each module in detail.

B. The Frontend

The source organization of different versions of SQLite changes as the software evolves. Furthermore, the source files contain scripts, tests, documents, besides native source code. In addition, recent versions of SQLite utilize a special build system to generate a single large source file `sqlite3.c` which consists of more than 200K lines of code.

To handle these peculiarities and make SQLSCAN both modular and extensible, we have introduced a separate frontend, which normalizes the source code by 1) filtering the source code by removing irrelevant components that are not affected by security flaws such as documents, 2) generating build-time sources and 3) preprocessing C header files to speed up subsequent processing.

Although combining the frontend with other phases is possible, the current design of SQLSCAN, from a software engineering perspective, has two main advantages: 1) it makes SQLSCAN feasible to process different versions of SQLite; and 2) SQLSCAN is more efficient by preprocessing and filtering irrelevant code in an early stage.

C. Vulnerability Detection

To conduct an empirical study on the security of SQLite, we need to detect vulnerabilities in the native code of SQLite. To achieve this goal, SQLSCAN leverages state-of-the-art and off-the-shelf vulnerability analysis tools to generate a detailed vulnerability report for subsequent processing.

A key criterion to select analysis tools is that the selected tools can generate, for the detected vulnerabilities, complete and precise information. Such information includes but is not limited to vulnerability type, source files, line numbers, etc., as one of our research goals is to investigate to what extent the detected vulnerabilities can be rectified automatically (Section III-E). Without the detailed vulnerability information, it is difficult if not impossible to rectify it in an automated manner. For instance, it is difficult to rectify a vulnerability without source locations.

To this end, we selected 8 popular state-of-the-art analysis tools as presented in TABLE II. All tools are actively maintained, with 4 of them were updated this year whereas others were updated last year. All but two (Fortify and Coverity) are open source. Except for two proprietary tools, 4 of the tools are licensed under GNU GPL.

TABLE II
ANALYSIS TOOLS USED IN THIS STUDY

Tools	Memory Vulnerability	Last Updated	Open-source	License
Infer [67]	✔	2022.03	✔	MIT
Clang [68]	✔	2021.11	✔	Apache v2.0
TscanCode [69]	✔	2022.01	✔	GNU GPL v3.0
Fortify [70]	✔	2022.01	✖	Proprietary
CppCheck [71]	✔	2022.02	✔	GNU GPL v3.0
Coverity [72]	✔	2021.12	✖	Proprietary
Valgrind [73]	✔	2021.10	✔	GNU GPL v2.0
FlawFinder [74]	✔	2021.07	✔	GNU GPL v2.0

Our work mainly studies memory-related vulnerabilities for two key reasons: 1) existing studies [75] [76] have demonstrated such vulnerabilities are very serious; and 2) state-of-the-art security analysis tools are good at detecting such vulnerabilities (see TABLE II). Specifically, this study focuses on 5 types of memory vulnerabilities: null dereferences, dead stores, uninitialized variables, resource leaks, and memory leaks. Nevertheless, it should be noted that our infrastructure SQLSCAN can be easily extended to detect other types of vulnerabilities as well.

It should be noted that the efficiency of the analysis tools used is of no significance, as SQLSCAN makes use of the analysis tools in an offline manner.

D. Purification

The purification module takes as the input both original vulnerability reports generated by the vulnerability analysis tools along with a SQLite C/C++ API semantics specification, and generates a vulnerability summary to be used by the subsequent rectification module.

The purification module has two key submodules: 1) filter: by stripping the original vulnerability report to retain only necessary information, this submodule not only simplifies but also speeds up subsequent modules; and 2) elaboration: by elaborating the original vulnerability report against the SQLite C/C++ API semantics specification, this submodule generates a more precise description of vulnerabilities.

The elaboration module is indispensable because although vulnerability analysis tools are effective in detecting vulnerabilities, they lack domain-specific knowledge about SQLite C/C++ API semantics. Such a lack of knowledge may lead to two problems: 1) false positives, which can lead to redundant rectifications; and 2) false negatives, which may lead to ignoring potential vulnerabilities. For instance, consider the following code from SQLite version 3.37:

```
1  // SQLite-3.37\shell.c #line 15726
2  static void
3  open_db(ShellState *p, int openFlags){
4  ...
5  aData = (unsigned char *)readFile(
       zDbFilename, &nData);
6  ...
7  rc = sqlite3_deserialize(p->db, "main",
       aData, nData, nData,
8  SQLITE_DESERIALIZE_RESIZEABLE |
9  SQLITE_DESERIALIZE_FREEONCLOSE);
10 }
```

the SQLite API function `sqlite3_deserialize()` accepts as the third argument a pointer `aData` at line 7. The flag `SQLITE _DESERIALIZE_FREEONCLOSE` at line 9 indicates that the memory pointed to by `aData` will be reclaimed by `sqlite3_deserialize()` automatically before return. If an analysis tool wrongly reports that `aData` is not reclaimed, it will mislead the rectification module to add an extra reclamation statement, leading to double frees.

As prior studies [77] demonstrated, static analysis may have a high false positive rate. Hence, in order to further exclude false positives and avoid redundant rectifications, we formed a group with three graduate students who are familiar with C/C++ languages and vulnerability analysis tools to conduct a manual inspection of the original vulnerability reports by analyzing the code logic and the information in reports. To ensure the reliability of the inspection results, we adopted the Fleiss' Kappa score, which is frequently used to test interrater reliability [78]. The inspection resulted in a Fleiss' Kappa score of 0.854 between the three students, which indicates an "Almost Perfect" agreement. In the rare cases where the students disagreed on the understanding of reported information, we conservatively judged the corresponding reports as false positives, because they cannot help people understand and fix bugs.

It should be noted that the presence of a purification module does not demonstrate the limitation of any analysis tools, as the programs analysis of nontrivial program properties on large software projects is conservative [79].

E. Automatic Rectification

The automatic rectification module takes as input the vulnerability summary generated by the purification module as well as the original SQLite native code, automatically generates the rectification suggestions according to the vulnerability summary. If the developers approve of the rectification suggestion generated by SQLScan, the patched SQLite code will be further processed by SQLScan's subsequent phases. It should be noted that the key research goal of this part of the study is to investigate to what extent the detected vulnerabilities can be rectified in an automated manner, thus providing deeper insights into the nature of the vulnerabilities of SQLite. It does not intend to substitute developer code reviews or existing code quality testing infrastructures such as CI/DI.

Technically, existing program rectification techniques [80] [81] [82] [83] [84] consist of either one type or a combination of two types of primitive operations: addition and removal of code segments. Our key insight here is that to rectify security instead of functionality vulnerabilities, it is generally enough to add extra code segments without modifying existing ones, because memory-related vulnerabilities are often caused by a lack of certain security checking such as releasing resources, freeing memory, or null checking [80].

Based on this key insight, SQLScan employs a simple, yet effective syntax-directed heuristics. To be specific, our current proof-of-concept implementation supports automatic rectifications of the following four types of vulnerabilities (with more still being added): 1) null dereference: SQLScan adds guard statements, which performs runtime checking to guarantee nullable variables are not dereferenced; 2) uninitialized variables: SQLScan inserts necessary initialization statements, based on variable types; 3) dead stores: SQLScan removes the corresponding dead statements conservatively; and 4) resource/memory leaks: SQLScan adds explicit reclaiming statements to reclaim resources or memories.

F. Validation

We have two primary goals for validation: 1) *security check*: the detected vulnerabilities have indeed been rectified; and 2) *functional check*: the normal functionality of SQLite is not changed.

To achieve the first goal, the patched source code is fed to the vulnerability analysis module and purification module for a second time, to obtain a new vulnerability summary. Then, the validation module compares the new vulnerability summary against the corresponding old one, to check that the rectified vulnerabilities no longer exist and no new vulnerabilities introduced in the new vulnerability summary.

To achieve the second goal, we leverage a regression strategy by using the testing suits distributed with SQLite. Although in theory regression testing is incomplete, it is a well established practice for program testing, and our experimental results with SQLScan demonstrated this strategy is effective in practice.

G. Rectified Program Generation

After rectifying the target program, SQLScan generates as outputs the rectified code, along with a security report to the developers.

IV. Empirical Results

In this section, we present the empirical results. We first give the research questions (Section IV-A), the experimental setup (Section IV-B), the data sets used (Section IV-C), and the evaluation metrics (Section IV-D. Then we present empirical results by answering research questions (Sections IV-E, IV-F, and IV-G).

A. Research Questions

By presenting the empirical results, we mainly investigate the following research questions:

TABLE III
VULNERABILITIES DETECTED BY SQLSCAN

Tool	ND		UV		DS		RL		ML		Total				
	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Precision	Recall	F1
TscanCode	176	26	29	1	0	0	8	6	10	6	223	39	17.49%	41.05%	24.53%
Infer	18	6	85	7	33	11	1	1	0	0	137	25	18.25%	26.04%	21.46%
CppCheck	7	3	12	4	0	0	3	1	7	3	29	11	37.93%	11.58%	17.74%
Clang	70	8	3	0	8	3	0	0	4	0	85	11	12.94%	11.58%	12.22%
Fortify	15	6	0	0	2	2	20	0	20	1	57	9	15.79%	9.47%	11.84%
Coverity	6	1	0	0	0	0	2	0	0	0	8	1	12.50%	1.05%	1.94%
Valgrind	0	0	0	0	0	0	0	0	0	0	0	0	NA	0.00%	NA
FlawFinder	0	0	0	0	0	0	0	0	0	0	0	0	NA	0.00%	NA
Total	292	50	129	12	43	16	34	8	41	10	539	96	NA	NA	NA

RQ1: Effectiveness. Is SQLSCAN effective in detecting vulnerabilities in SQLite native code? How about precision and recall of SQLSCAN?

RQ2: Evolution. How do vulnerabilities evolve in different versions of SQLite?

RQ3: Rectification. Is SQLSCAN effective in rectifying vulnerabilities in SQLite naive code automatically?

RQ4: Usefulness. Are empirical study and rectification results useful to the SQLite?

B. Experimental Setup

All the experiments and measurements are performed on a server with one 4 physical Intel i7 core (8 hyperthread) CPU and 12 GB of RAM running Ubuntu 21.04.

C. Data Sets

We selected 10 different versions of SQLite (from 3.6 to 3.37) as our data sets. We have two principles guiding our selection of data sets: 1) in order to investigate the evolution of vulnerabilities in SQLite, we selected 10 different major versions of SQLite according to the release time, spanning 13 years from 2008 to 2021. SQLite has grown significantly: with number of files grew from 117 to 311, and lines of source code grew from 125,569 to 352,678; 2) we only selected SQLite versions 3.x but not versions 2.x, because SQLite 3.x introduced a new format for database files which is incompatible with version 2.x [85]. Moreover, the official SQLite website specifies that SQLite 2.x has not been updated since 2005 [86], thus is obsolete. However, it should be noted that SQLSCAN can also be easily extended to process SQLite 2.x.

D. Evaluation Metrics

In this paper, we use TP, FP, P to denote true positives, false positives and ground truth, respectively. A tool's TPs are defined as the vulnerabilities reported by the tool and confirmed by the aforementioned group of three graduates, and the FPs are given as the inverse of the TP process. We take the union of TPs from all analysis tools as our ground truth P.

We use $precision$ and $recall$ to measure the accuracy (or effectiveness) of the tools we use. The definition of these two metrics is in equation (1).

$$Precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{P} \tag{1}$$

Recall measures the ratio of true positives to the ground truth. An analysis tool with high false negatives may have low recall. Precision measures the ratio of true positives to the result of a tool. An analysis tool with high false positives may have low precision. Given the importance of both recall and precision, we also compute the $F1$ score according to the equation (2).

$$F1\,score = \frac{2 \times precision \times recall}{precision + recall} \tag{2}$$

The F1 score can reflect the overall accuracy of an analysis tool.

E. RQ1: Effectiveness

To answer **RQ1** by investigating the effectiveness of SQLSCAN in detecting vulnerabilities, we first take the latest SQLite version 3.37, and apply SQLSCAN on it.

Table III presents the empirical results: the first column gives the names of the analysis tools. The next 5 columns present the numbers of vulnerabilities detected by each tool according to vulnerability category: null dereferences (ND), uninitialized variables (UV), dead stores (DS), resource leaks (RL) and memory leaks (ML). For each category, we present the number of vulnerabilities each tool reported, as well as true positives (TPs) among them. The last 5 columns present the total vulnerabilities detected with true positives, as well as the metrics of precision, recall and F1 score.

The empirical results provide interesting findings and insights. First, after scanning 352,678 lines of code in 311 native files in SQLite 3.37 by the 8 analysis tools and manual inspection, we found a total of 96 vulnerabilities, including 50 null dereferences (52.63%), 12 uninitialized variables (12.63%), 16 dead stores (16.84%), 7 resource leaks (7.37%) and 10 memory leaks (10.53%). This empirical result shows that the native codes of SQLite are still vulnerable, which should be brought to the attention of storage security researchers and SQLite developers.

TABLE IV
EVOLUTIONS OF VULNERABILITIES, IN 10 VERSIONS OF SQLITE

Version	ND		UV		DS		RL		ML		Total				
	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Reports	TPs	Files	LoC	VPTL
3.6	14	3	92	12	27	6	0	0	0	0	133	21	117	125,569	0.17
3.7	25	3	29	3	33	3	0	0	0	0	87	9	142	156,282	0.06
3.8	24	6	30	3	12	3	0	0	0	0	66	12	173	198,163	0.06
3.9	22	3	65	5	16	5	0	0	0	0	103	13	211	246,613	0.05
3.15	25	2	66	6	26	9	1	1	0	0	118	18	224	268,786	0.07
3.21	21	2	64	6	29	9	1	1	0	0	115	18	268	309,588	0.06
3.25	38	4	72	5	31	8	1	1	0	0	142	18	291	326,807	0.06
3.30	42	5	75	6	34	9	1	0	0	0	152	20	302	337,643	0.06
3.33	15	4	75	6	37	11	1	0	0	0	128	21	310	344,791	0.06
3.37	18	6	85	7	33	11	1	1	0	0	137	25	311	352,678	0.07

Second, except for CppCheck, other security analysis tools have a low precision (under 20%). Although CppCheck has the highest precision (37.93%), its recall is rather low (11.58%), thus the F1 score is only 17.74%. Both Infer and TscanCode have not only high recalls (26.04% and 41.05%, respectively), but also high F1 scores (21.46% and 24.53%, respectively), which results in their relatively high accuracy. However, the overall F1 scores of the 8 analysis tools are low and unsatisfactory. For instance, only 39 of the 223 vulnerabilities reported by TscanCode are true positives. This result demonstrated that these popular analysis tools have considerable room for further improvements.

Third, to further investigate false positives and false negatives, the aforementioned group of graduate students conducted a manual analysis of the code fragments corresponding to these FPs and FNs. This inspection reveals two key reasons for false positives: 1) *duplicate reports*: for example, we found 41 out of the 85 (48.24%) vulnerabilities reported by Clang were duplicate reported, including 32 null dereferences, 5 dead stores, 2 uninitialized variables, and 2 memory leaks. We believe the reason is that the detection techniques for some types of vulnerabilities are path-sensitive, and different testing paths lead to the same program point, resulting in duplicate reports. If these duplicate reports were not filtered in advance, they will mislead the automatic rectification module to add redundant and incorrect patches to the same vulnerable code segment, which may introduce new vulnerabilities, such as freeing the same pointer multiple times while rectifying a memory leak. 2) *incomplete context information*: security analysis tools will report false positives, when the context information is incomplete or even missing. For example, the Infer reported a null dereference vulnerability for the variable pointer `pCExpr` at line 4 in the following code snippet:

```
1 //SQLite-3.37\build.c #line 1860
2 Expr *pCExpr=sqlite3ExprSkipCollate(...);
3 assert( pCExpr!=0 );
4 sqlite3StringToId(pCExpr);
```

However, in fact, the `assert( pCExpr!=0 )` at line 3 guarantees the `pCExpr` cannot be nullable. Such false positives are also commonly reported by Clang or TscanCode, which should arouse the attention of the security tool builders. Fortunately, most of analysis tools are still under active development and future improvements may reduce such false positives. On the other hand, we found that all 8 tools had false negatives. For instance, a known SQLite security issue was fixed by a commit [87]. This issue is demonstrated by the following code snippet:

```
//SQLite-3.37\src\select.c #line 7447
if( eDist!=WHERE_DISTINCT_NOOP ){
struct AggInfo_func *pF=&pAggInfo->aFunc[0];
fixDistinctOpenEph(pParse, eDist, pF->
    iDistinct, pF->iDistAddr); }
```

where the pointer `pF` is nullable. Unfortunately, none of the 8 tools detected it. We argue that one key reason for such false negatives is that these tools do not cover all possible execution paths due to the nature of static analysis, hence path-dependent vulnerabilities are difficult to detect under some complex situations.

Finally, both Valgrind and FlawFinder detected no possible vulnerabilities. For Valgrind, the SQLite developers acknowledged us that Valgrind is used with SQLite, while running the Tcl-driven test suite, prior to every SQLite release [88]. And for FlawFinder, its manual [74] specifies that it makes use simple pattern-matching strategy, which may be too weak to analyze large code bases such as SQLite.

Summary: the native code of SQLite is still vulnerable, with 96 vulnerabilities detected. The 8 analysis tools have a high false positive rate and still need further improvements.

F. RQ2: Evolution

To answer **RQ2** by investigating the evolution of vulnerabilities in different versions of SQLite, we applied SQLSCAN to scan 10 versions of SQLite, from 3.6 to the latest 3.37. Due to both space limitations and clarity, in TABLE IV we only present vulnerability reports reported by Infer. The reproduction package contains all original data.

Several interesting findings and insights can be obtained from these empirical results. First, in order to gain an understanding of how the overall code quality of SQLite evolves, we calculate the numbers of vulnerabilities per thousand lines of code (VPTL), which is a well-established metrics for measuring code quality [89] [90] [91]. Although the SQLite has grown significantly, from 125,569 lines of code in version 3.6 to 352,678 lines of code in version 3.37, the VPTL always fluctuated around 0.06 (that is, about 1 vulnerability in 20000 lines of code). This finding indicates that the overall quality of SQLite remained relatively stable for these different versions.

Second, the empirical result shows that among all vulnerabilities, the 3 types of vulnerabilities: null dereferences, dead stores, and uninitialized variables continue to account for the majority. Few memory leak or resource leak vulnerabilities have been identified during the evolution process. We have identified three reasons: 1) these types of vulnerabilities do not exist; 2) the analysis tools have limited capabilities; 3) the complex code logic makes it difficult to find vulnerabilities. Moreover, we found that some vulnerabilities had persisted in multiple versions for a long time before they were detected and fixed, which will be discussed in Section IV-H.

Summary: the code quality of SQLite remains stable overall in 10 versions spanning 13 years, even though its code grows significantly from 125,569 to 352,678 lines of code.

G. Automatic Rectification

To answer **RQ3** by investigating the effectiveness of SQLScan to rectify vulnerabilities automatically, we present in TABLE V the rectification results for SQLite 3.37.

These empirical results give several relevant findings. First, the automatic rectification of SQLScan is effective. As the 3rd column of TABLE V shows that SQLScan successfully rectified 23 out of 25 (92.00%) vulnerabilities, including 6 null dereferences (100%), 7 uninitialized variables (100%), 9 dead stores (81.82%) and 1 resource leak (100.00%) in SQLite 3.37. Moreover, these results proved that simple syntax-directed heuristics are sufficient to fix these vulnerabilities.

TABLE V
Vulnerabilities SQLScan Rectified in SQLite 3.37

Vulnerability Category	No.	Rectified
Null Dereference	6	6 (100.00%)
Uninitialized Variable	7	7 (100.00%)
Dead Store	11	9 (81.82%)
Resource Leak	1	1 (100.00%)
Total	25	23 (92.00%)

Second, in order to gain an understanding of why SQLScan failed to rectify 2 vulnerabilities (especially the DS category), our group further conducted a manual inspection of all the vulnerable code segments that SQLScan failed to rectify. This inspection reveals a key reason: as prior studies [92] [93] have demonstrated that aggressive dead store elimination may remove seemingly useless but actually useful memory stores. For instance, in multithreaded programming, developers often add dead stores into specific threads to reduce thread contention. Hence, to avoid removing useful dead stores, SQLScan employed heuristics to perform conservative removal, which leads to a low DS rectification ratio.

Finally, to test the performance of automatic rectification, we conducted experiments to measure the time SQLScan spent in rectifying 10 different versions (from 3.6 to 3.37). We ran SQLScan 30 rounds on each version of SQLite, respectively. Then, we calculated the time spent for each run, and the time spent to rectify each vulnerability. TABLE VI presents the number of total vulnerabilities (#No.), the number of rectified vulnerabilities (#Rectified), the total time

TABLE VI
Performance of SQLScan on 10 Versions of SQLite

Versions	#No.	#Rectified	Total (ms)	Average (ms)
3.6	21	21 (100.00%)	228.86	10.90
3.7	9	7 (77.78%)	236.05	33.72
3.8	12	10 (83.33%)	362.76	36.28
3.9	13	11 (84.62%)	320.28	29.12
3.15	18	17 (94.44%)	598.55	35.21
3.21	18	17 (94.44%)	497.54	29.27
3.25	18	18 (100.00%)	619.41	34.41
3.30	20	18 (90.00%)	633.94	35.22
3.33	21	21 (100.00%)	582.64	27.74
3.37	25	23 (92.00%)	591.54	26.89

to rectify all vulnerabilities, and the average time to rectify one vulnerability, all in milliseconds. This empirical result demonstrated that SQLScan is efficient: it takes only 591.54 milliseconds to rectify all 22 vulnerabilities in SQLite 3.37. On average, it takes about 10 to 40 milliseconds to rectify one vulnerability.

Summary: most vulnerabilities in SQLite are straightforward to rectify by syntax-directed heuristics: SQLScan successfully rectified 92.00% vulnerabilities in SQLite 3.37. SQLScan is efficient and practical to rectify vulnerabilities, taking less than 40 milliseconds to rectify one vulnerability.

H. Usefulness

To answer **RQ4** by investigating the usefulness of SQLScan, we investigate whether vulnerabilities found by SQLScan in old versions of SQLite have been fixed.

The empirical results provide interesting findings and insights. First, after tracing the vulnerabilities from 10 versions of SQLite presented in TABLE IV, we discover that 9 vulnerabilities reported by SQLScan were fixed by the SQLite development team, which is presented in TABLE VII. The average latency of these vulnerabilities is 17.2 months (nearly 1.5 years) and the longest duration is 42 months (nearly 3.5 years). This empirical result shows that vulnerabilities in SQLite native code had persisted in multiple versions for a long time, before they were discovered and fixed, which leave security risks to SQLite. For instance, SQLScan found a resource leak vulnerability that first appeared in 2016, persisted in subsequent versions, and was not fixed until 2018 [99], which is illustrated by the following code snippet:

TABLE VII
VULNERABILITIES FIXED IN OLD VERSIONS OF SQLITE AND FOUND BY SQLSCAN

No.[1]	Vulnerability Category	Location	Variable Name	LoC	Rectification Method			Latent Period in Months
					Description	Commit ID	Same[2]	
1	ND	src/btree.c	`pPage`	7558	Check before using nullable variable	0c5a7d11 [94]	✔	25 mo.
2	UV	src/btree.c	`rc`	7509	Insert necessary initialization	ea01d437 [95]	✔	20 mo.
3	ND	tool/lemon.c	`types`	4932	Check before using nullable variable	efb20b9d [96]	✔	52 mo.
4	UV	src/vdbe.c	`pIn2`	5156	Insert necessary initialization	14e6d19c [97]	✔	11 mo.
5	UV	src/vdbe.c	`pOut`	5156	Insert necessary initialization	14e6d19c [97]	✔	11 mo.
6	UV	src/insert.c	`regEof`	7509	Insert necessary initialization	ea01d437 [95]	✔	6 mo.
7	UV	src/select.c	`addrOutB`	5156	Insert necessary initialization	14e6d19c [97]	✔	6 mo.
8	ND	src/expr.c	`pE2`	4266	Check before returning nullable variable	036fc37a [98]	✖	9 mo.
9	RL	src/shell.c	`in`	8918	Release resources before exiting	ec36d15a [99]	✔	25 mo.

[1] Informal number.
[2] Is the official rectification method same as ours?

```
1 //SQLite-3.15\shell.c #line 2253
2 static char *readFile(const char *zName){
3 FILE *in = fopen(zName, "rb"); ...
4 if( in==0 ) return 0; ...
5 if( pBuf==0 ) return 0;
6 //patch: if(pBuf==0){fclose(in);return 0;}
7 ...
8 fclose(in);}
```

the file pointed to by `in` was not closed at line 5, when the function `readFile` exited early.

Second, these vulnerabilities were rectified by the SQLite development team in simple ways, which are similar to the one we proposed in Section III-E. As the 7th column of TABLE VII shows, the SQLite development team inserted necessary initialization statements to fix uninitialized variables [95] [97], added runtime checking for nullable pointers to fix null dereferences [94] [96] [98] and added reclaiming statements to fix leaked resources [99]. For instance, the resource leak vulnerability presented in the above snippet was rectified by simply adding a resource recovery statement (as line 4 presented). Although these vulnerabilities are not difficult to fix, their long latent period does bring security risks to SQLite. If SQLite developers can detect such vulnerabilities early by applying analysis tools and fix them in a timely manner, such risks will be largely reduced thus SQLite will be more robust and secure.

Summary: 9 vulnerabilities reported by SQLSCAN had been discovered and fixed by the SQLite development team. Vulnerabilities in the SQLite native code had persisted for a long time before they were detected and fixed, which leave security risks to SQLite. And the strategies SQLite developers use to rectify vulnerabilities are similar to those used by SQLSCAN automatic rectification module.

V. IMPLICATIONS

This paper presents the first and largest empirical study on vulnerability detection, rectification and evolution of the SQLite native code. In this section, we discuss some implications of this work, along with some important directions for future research.

For SQLite Developers. The results in this work provide SQLite developers with important insights into improving the security of the SQLite native code. On the one hand, security analysis tools already deployed have improved the security of SQLite considerably. For example, Valgrind does not report any vulnerabilities in SQLite 3.37, because the SQLite developers acknowledged that they already make use of Valgrind internally to scan the source code before release [88]. On the other hand, the latest progress in security analysis techniques and tools will benefit the SQLite developers. Another direction for exploration is to integrate tools such as SQLSCAN into the production systems (e.g., CI/DI), which can further improve the security of SQLite.

Although SQLite is not open to free contribution, which limited our access to SQLite developers, we made efforts to address this limitation by using widely recognized analytical tools and detailing the study methodology and results. We also received feedback and suggestions from other researchers in communities and forums to ensure the usefulness of our findings. Moreover, in order to make our work really beneficial to the SQLite developers, we have sent emails to the SQLite developers, according to the official guidelines for vulnerability reporting, Our report not only contains the vulnerability reports in this work, but also our suggestion to integrate state-of-the-art security tools into the CI/DI process of SQLite. Our report is being analyzed and evaluated by the SQLite developers.

For Security Analysis Tool Builders. Given the importance of software infrastructures such as SQLite, the results in this work provide insights to security tool builders to develop more effective analysis techniques. First, security tool builders should put more research efforts into improving precision by suppressing duplicate reports and incorporating more path-sensitive and context-sensitive analysis algorithms.

Second, automatic rectification of vulnerabilities reported by analysis tools is a promising field for future research. Our experimental results show that many memory-related vulnerabilities can be rectified by a simple, yet effective syntax-directed heuristics; unfortunately, none of the tools we investigated have such capabilities.

Finally, we (sadly) observed that, none of the security tools we investigated has incorporated any database engines such as SQLite into their standard test suits. We believe that, by incorporating such benchmarks, the accuracy of these tools can be further improved.

For Storage Security Researchers. The results in this work provide researchers with new research directions into improving the storage security. To be specific, researchers can conduct more in-depth studies on the security of the native code, in addition to the well-studied SQL injection, data encryption, data integrity, and access control, as our empirical results demonstrated that the assumption that *SQLite implementation is just secure and trustworthy* does not truly hold.

VI. Threats to Validity

As any empirical study, our work is subject to threats to validity. We attempt to remove these threats where possible, and mitigate the effect when removal is not possible.

Data sets. In this work, we used SQLite as our data set to evaluate SQLScan, as it is the most widely deployed database system and its usage has exceeded the sum of all other database engines, thus any bug in SQLite can lead to serious damage. Meanwhile, other DBMSs are also implemented in C/C++, such as MySQL [100]. Fortunately, the modular design of SQLScan brings extensibility and makes it possible to study other DBMS as well. As we have made our tool open source and publicly available, in the short term, we are planning to conduct experiments on MySQL, a widely used open source database management system, which has also been reported to be vulnerable recently [101] [102].

Other Vulnerabilities. In this work, we have concentrated on detecting and rectifying memory-related vulnerabilities in SQLite, and the empirical results demonstrated that SQLScan is effective in achieving this research goal. Although memory-related vulnerabilities are among one of the most severe vulnerabilities [34], other types of vulnerabilities exist in SQLite, such as thread safety issues, information flow bugs, etc. Fortunately, the modular architecture of SQLScan (Fig. 1) can be easily extended to support the detection and rectification of other types of vulnerabilities without difficulty. We believe the main technical challenge here is the detection capability of security tools. For instance, although most state-of-the-art tools can detect memory-related vulnerabilities, few of them can handle concurrent vulnerabilities well.

Analysis Tools. In this work, we used 8 analysis tools to obtain the empirical results, which demonstrate the effectiveness of these tools. and formed a group with 3 graduate students to conduct a manual inspection of the original defect report for each checker and identify all true positives (TPs). Although the knowledge level of each student may be different, we calculated Fleiss' kappa score [78] to evaluate the reliability of their manual inspection results. In the meanwhile, proprietary analysis tools such as CxSAST [103] may detect other vulnerabilities, but we do not have the necessary resources to buy or explore. On the one hand, the 8 analysis tools we used in our work are widely used thus the results are trustworthy. On the other hand, fortunately, the architecture of SQLScan (Fig. 1) is neutral to the specific analysis tools used. The modular design of SQLScan makes it easy to incorporate other analysis tools.

VII. Related Work

In recent years, a significant number of studies have focused on the security of DBMS. However, the work in this paper represents a novel contribution to this field.

A. SQL Security

Considerable studies have been done on SQL security. Dubey R et al. [40] proposed SQL filtering to prevent SQL injection attacks and maintain the confidentiality, integrity and authenticity of data. Katole R A [38] proposed an SQL injection attack detection method by removing the parameter values of the SQL query. Wang Y et al. [46] proposed CryptSQLite, a high security SQLite database system, which protects both the confidentiality and integrity of the data. StealthDB [52] is an encrypted database system which has a small trusted computing base and provides a relatively strong security guarantee.

However, a major limitation of existing efforts is that they only focus on SQL injection, data encryption, data integrity, and access control, but assume the implementations of SQL engines are safe and trustworthy. Moreover, despite the numerous studies on the security of native code, [104] [105] [106], they did not focus on the vulnerabilities in SQLite.

B. Automatic Program Rectification

Recently, automatic program rectification has become one of the most important subjects. Tonder [75] presented Semantic Crash Bucketing, a generic method for precise crash bucketing using program transformation, to automatically generate and apply approximate fixes for general bug classes. Kim et al. [107] proposed Par, a novel patch generation approach, which uses fix patterns learned from existing human-written patches to generate program patches automatically. GenProg [108] is an automated method for repairing defects in off-the-shelf programs by using an extended form of genetic programming to evolve a program variant.

However, a major limitation of existing work is that most of them are test-driven and regression-based and focus on the automatic rectification of vulnerabilities in common programs. They cannot be used directly to rectify the vulnerabilities in SQLite. Another key difference between SQLScan and these works is that SQLScan is extensible to allow developers to provide custom patch templates.

VIII. Conclusion

In this work, we present the first empirical study on vulnerability detection, rectification and evolution of the SQLite native code. By utilizing a novel tool SQLScan, we performed a comprehensive study of SQLite. The empirical results show that the SQLite is still vulnerable. Most vulnerabilities in SQLite can be easily rectified by SQLScan.

References

[1] L. Jun-feng, "Application of sqlite in embedded middleware system," *Computer and Modernization*, vol. 1, no. 5, p. 184, 2010.

[2] "Sqlite odbc driver," http://www.ch-werner.de/sqliteodbc/.

[3] H. HARIANI, "Eksplorasi web browser dalam pencarian bukti digital menggunakan sqlite," *Jurnal INSTEK (Informatika Sains dan Teknologi)*, vol. 6, no. 1, pp. 66–74, 2021.

[4] R. Bagley, R. I. Ferguson, and P. Leimich, "On the digital forensic analysis of the firefox browser via recovery of sqlite artifacts from unallocated space," in *6th International Conference on Cybercrime Forensics Education & Training,*. Canterbury Christ Church University, 2012.

[5] S. Jeon, J. Bang, K. Byun, and S. Lee, "A recovery method of deleted record for sqlite database," *Personal and Ubiquitous Computing*, vol. 16, no. 6, pp. 707–715, 2012.

[6] K. Dhokale, N. Bange, S. Pradip, and S. Malave, "Implementation of sql server based on sqlite engine on android platform," *International Journal of Research in Engineering and Technology*, vol. 3, no. 4, pp. 8–14, 2014.

[7] N. Obradovic, A. Kelec, and I. Dujlovic, "Performance analysis on android sqlite database," in *2019 18th International Symposium INFOTEH-JAHORINA (INFOTEH)*. IEEE, 2019, pp. 1–4.

[8] C. Bi, "Research and application of sqlite embedded database technology," *WSEAS Trans. Comput*, vol. 8, no. 1, pp. 83–92, 2009.

[9] "Well-known users of sqlite," https://www.sqlite.org/famous.html.

[10] L. Junyan, X. Shiguo, and L. Yijie, "Application research of embedded database sqlite," in *2009 International Forum on Information Technology and Applications*, vol. 2. IEEE, 2009, pp. 539–543.

[11] C. Meng and H. Baier, "bring2lite: a structural concept and tool for forensic data analysis and recovery of deleted sqlite records," *Digital Investigation*, vol. 29, pp. S31–S41, 2019.

[12] L. Yang, H. Pang, L. Jiang, and K. Yue, "The research of embedded linux and sqlite database application in the intelligent monitoring system," in *2010 International Conference on Intelligent Computation Technology and Automation*, vol. 3. IEEE, 2010, pp. 910–914.

[13] "Most widely deployed sql database engine," https://www.sqlite.org/mostdeployed.html.

[14] "Stack overflow developer survey 2021," https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021.

[15] V. Jain, M. S. Gaur, V. Laxmi, and M. Mosbah, "Detection of sqlite database vulnerabilities in android apps," in *International Conference on Information Systems Security*. Springer, 2016, pp. 521–531.

[16] "Cve-2018-20346," https://nvd.nist.gov/vuln/detail/cve-2018-20346.

[17] "Cve-2018-20505," https://nvd.nist.gov/vuln/detail/cve-2018-20505.

[18] "Cve-2018-20506," https://nvd.nist.gov/vuln/detail/cve-2018-20506.

[19] "Cve-2019-13734," https://nvd.nist.gov/vuln/detail/cve-2019-13734.

[20] "Cve-2019-13750," https://nvd.nist.gov/vuln/detail/cve-2019-13750.

[21] "Cve-2019-13751," https://nvd.nist.gov/vuln/detail/cve-2019-13751.

[22] "Cve-2019-13752," https://nvd.nist.gov/vuln/detail/cve-2019-13752.

[23] "Cve-2019-13753," https://nvd.nist.gov/vuln/detail/cve-2019-13753.

[24] "Magellan: Sqlite remote code execution vulnerability," https://blade.tencent.com/en/advisories/sqlite//.

[25] "Magellan 2.0: Sqlite remote code execution vulnerability," https://blade.tencent.com/en/advisories/sqlite_v2//.

[26] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov, "Inputs of coma: Static detection of denial-of-service vulnerabilities," in *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 2009, pp. 186–199.

[27] A. V. Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, "Variability-aware static analysis at scale: An empirical study," vol. 27, no. 4, pp. 1–33. [Online]. Available: https://dl.acm.org/doi/10.1145/3280986

[28] J. Taneja, Z. Liu, and J. Regehr, "Testing static analyses for precision and soundness," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. ACM, pp. 81–93. [Online]. Available: https://dl.acm.org/doi/10.1145/3368826.3377927

[29] Y. Lyu, S. Volokh, W. G. J. Halfond, and O. Tripp, "Sand: A static analysis approach for detecting sql antipatterns," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, pp. 270–282. [Online]. Available: https://dl.acm.org/doi/10.1145/3460319.3464818

[30] J. C. Foster, V. Osipov, N. Bhalla, N. Heinen, and D. Aitel, "Buffer overflow attacks," *Syngress, Rockland, USA*, 2005.

[31] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, pp. 1–29, 2015.

[32] E. D. Berger, "Heapshield: Library-based heap overflow protection for free," *UMass CS TR*, pp. 06–28, 2006.

[33] H. Zhang, S. Wang, H. Li, T.-H. P. Chen, and A. E. Hassan, "A study of c/c++ code weaknesses on stack overflow," *IEEE Transactions on Software Engineering*, 2021.

[34] Y. Younan, W. Joosen, F. Piessens, and H. den Eynden, "Security of memory allocators for c and c++," Technical Report CW 419, Department of Computer Science, Katholieke ..., Tech. Rep., 2005.

[35] F. Akowuah and A. Ahlawat, "Protecting sensitive data in android sqlite databases using trustzone," in *2018 International Conference on Security & Management*, vol. 2018, 2018.

[36] D. Kim, S. Kim, and J. Ryou, "Vulnerability assessment of android sqlite database for information exposure detection," *International Information Institute (Tokyo). Information*, vol. 19, no. 2, p. 491, 2016.

[37] H. Gu, J. Zhang, T. Liu, M. Hu, J. Zhou, T. Wei, and M. Chen, "Diava: a traffic-based framework for detection of sql injection attacks and vulnerability analysis of leaked data," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 188–202, 2019.

[38] R. A. Katole, S. S. Sherekar, and V. M. Thakare, "Detection of sql injection attacks by removing the parameter values of sql query," in *2018 2nd International Conference on Inventive Systems and Control (ICISC)*. IEEE, 2018, pp. 736–741.

[39] J. Thomé, L. K. Shar, and L. Briand, "Security slicing for auditing xml, xpath, and sql injection vulnerabilities," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 553–564.

[40] R. Dubey and H. Gupta, "Sql filtering: An effective technique to prevent sql injection attack," in *2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*. IEEE, 2016, pp. 312–317.

[41] D. Das, U. Sharma, and D. Bhattacharyya, "Defeating sql injection attack in authentication security: an experimental study," *International Journal of Information Security*, vol. 18, no. 1, pp. 1–22, 2019.

[42] H. Gupta, S. Mondal, S. Ray, B. Giri, R. Majumdar, and V. P. Mishra, "Impact of sql injection in database security," in *2019 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE)*. IEEE, 2019, pp. 296–299.

[43] M. Lawal, A. B. M. Sultan, and A. O. Shakiru, "Systematic literature review on sql injection attack," *International Journal of Soft Computing*, vol. 11, no. 1, pp. 26–35, 2016.

[44] N. M. Sheykhkanloo, "A learning-based neural network model for the detection and classification of sql injection attacks," *International Journal of Cyber Warfare and Terrorism (IJCWT)*, vol. 7, no. 2, pp. 16–41, 2017.

[45] P. N. Joshi, N. Ravishankar, M. Raju, and N. C. Ravi, "Contemplating security of http from sql injection and cross script," in *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*. IEEE, 2017, pp. 1–5.

[46] Y. Wang, Y. Shen, C. Su, J. Ma, L. Liu, and X. Dong, "Cryptsqlite: Sqlite with high data security," *IEEE Transactions on Computers*, vol. 69, no. 5, pp. 666–678, 2019.

[47] A. Zhao, Z. Wei, and Y. Yang, "Research on sqlite database encryption technology in instant messaging based on android platform," in *International Conference on Intelligent Science and Big Data Engineering*. Springer, 2015, pp. 316–325.

[48] P. Antonopoulos, A. Arasu, K. D. Singh, K. Eguro, N. Gupta, R. Jain, R. Kaushik, H. Kodavalla, D. Kossmann, N. Ogg *et al.*, "Azure sql database always encrypted," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1511–1525.

[49] S. Kamara and T. Moataz, "Sql on structurally-encrypted databases," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2018, pp. 149–180.

[50] O. A. Safaryan, E. V. Roshchina, L. V. Cherckesova, E. V. Pinevich, A. G. Lobodenko, and B. A. Akishin, "Cryptographic algorithm implementation for data encryption in dbms ms sql server," in *2020 IEEE East-West Design & Test Symposium (EWDTS)*. IEEE, 2020, pp. 1–6.

[51] G. Liu, G. Yang, H. Wang, Y. Xiang, and H. Dai, "A novel secure scheme for supporting complex sql queries over encrypted databases in

cloud computing," *Security and Communication Networks*, vol. 2018, 2018.

[52] D. Vinayagamurthy, A. Gribov, and S. Gorbunov, "Stealthdb: a scalable encrypted database with full sql query support." *Proc. Priv. Enhancing Technol.*, vol. 2019, no. 3, pp. 370–388, 2019.

[53] X. Oriol Hilari, E. Teniente López, and G. Rull, "Tintin: a tool for incremental integrity checking of assertions in sql server," in *Advances in Database Technology-EDBT 2016, 19th International Conference on Extending Database Technology, Bordeaux, France, March 15-16, Proceedings*, 2016, pp. 632–635.

[54] P. Antonopoulos, R. Kaushik, H. Kodavalla, S. Rosales Aceves, R. Wong, J. Anderson, and J. Szymaszek, "Sql ledger: Cryptographically verifiable data in azure sql database," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2437–2449.

[55] R. A. Haraty and B. Zantout, "A collaborative-based approach for avoiding traffic analysis and assuring data integrity in anonymous systems," *Computers in Human Behavior*, vol. 51, pp. 780–791, 2015.

[56] H. Shahriar and H. M. Haddad, "Security vulnerabilities of nosql and sql databases for mooc applications," *International Journal of Digital Society (IJDS)*, vol. 8, no. 1, pp. 1244–1250, 2017.

[57] S. Mutti, E. Bacis, and S. Paraboschi, "Sesqlite: Security enhanced sqlite: Mandatory access control for android databases," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 411–420.

[58] J. Dave and M. L. Das, "Securing sql with access control for database as a service model," in *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, 2016, pp. 1–6.

[59] M. Guarnieri, S. Marinovic, and D. Basin, "Strong and provably secure database access control," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 163–178.

[60] M. I. Sarfraz, M. Nabeel, J. Cao, and E. Bertino, "Dbmask: Fine-grained access control on encrypted relational databases," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015, pp. 1–11.

[61] C. Zhang and J. Yin, "Research on security mechanism and forensics of sqlite database," in *International Conference on Artificial Intelligence and Security*. Springer, 2021, pp. 614–629.

[62] L. Haiyan and G. Yaowan, "Analysis and design on security of sqlite," in *International Conference on Computer, Networks and Communication Engineering (ICCNCE 2013)*, 2013.

[63] Z. S. Alwan and M. F. Younis, "Detection and prevention of sql injection attack: A survey," *International Journal of Computer Science and Mobile Computing*, vol. 6, no. 8, pp. 5–17, 2017.

[64] S. Bhosale, M. T. Patil, and M. P. Patil, "Sqlite: Light database system," *Int. J. Comput. Sci. Mob. Comput*, vol. 44, no. 4, pp. 882–885, 2015.

[65] "The sqlite amalgamation," https://sqlite.org/amalgamation.html.

[66] "Apache log4j security vulnerabilities," https://logging.apache.org/log4j/2.x/security.html.

[67] "Infer static analyzer — infer — infer," https://fbinfer.com/.

[68] "Clang static analyzer," https://clang-analyzer.llvm.org/.

[69] "Tscancode," Tencent, Aug. 2022.

[70] "Fortify," https://www.joinfortify.com/.

[71] "Cppcheck," https://cppcheck.sourceforge.io/.

[72] "Coverity scan," https://scan.coverity.com/.

[73] "Valgrind home," https://valgrind.org/.

[74] "Flawfinder home page," https://dwheeler.com/flawfinder/.

[75] R. van Tonder, J. Kotheimer, and C. Le Goues, "Semantic crash bucketing," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 612–622.

[76] F. Medeiros, M. Ribeiro, R. Gheyi, L. Braz, C. Kästner, S. Apel, and K. Santos, "An empirical study on configuration-related code weaknesses," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 2020, pp. 193–202.

[77] G. Tan and J. Croft, "An empirical security study of the native code in the jdk." in *Usenix Security Symposium*, 2008, pp. 365–378.

[78] M. L. McHugh, "Interrater reliability: The kappa statistic," *Biochemia Medica*, pp. 276–282, 2012.

[79] P. Cousot and R. Cousot, "Modular static program analysis," in *International Conference on Compiler Construction*. Springer, 2002, pp. 159–179.

[80] R. van Tonder and C. L. Goues, "Static automated program repair for heap properties," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 151–162.

[81] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[82] H. Yan, Y. Sui, S. Chen, and J. Xue, "Automated memory leak fixing on value-flow slices for c programs," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 1386–1393.

[83] X. Cheng, M. Zhou, X. Song, M. Gu, and J. Sun, "Automatic fix for c integer errors by precision improvement," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 2–11.

[84] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 295–306.

[85] "Sqlite version 3 overview," https://www.sqlite.org/version3.html.

[86] "History of sqlite releases," https://www.sqlite.org/chronology.html.

[87] "Fix a null-pointer dereference," https://github.com/sqlite/sqlite/commit/3117b7b081d782c5da91da94a03cf135ed05533d.

[88] "How sqlite is tested," https://www.sqlite.org/testing.html.

[89] S. C. Misra and V. C. Bhavsar, "Relationships between selected software measures and latent bug-density: Guidelines for improving quality," in *International Conference on Computational Science and Its Applications*. Springer, 2003, pp. 724–732.

[90] T. Bach, A. Andrzejak, R. Pannemans, and D. Lo, "The impact of coverage on bug density in a large industrial software project," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 307–313.

[91] Z. Shams and S. H. Edwards, "An experiment to test bug density in students' code," in *Proceeding of the 44th ACM technical symposium on Computer science education*, 2013, pp. 742–742.

[92] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Security and Privacy Workshops*. IEEE, 2015, pp. 73–87.

[93] Z. Yang, B. Johannesmeyer, A. T. Olesen, S. Lerner, and K. Levchenko, "Dead store elimination (still) considered harmful," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1025–1040.

[94] "Sqlite: Check-in [0c5a7d11]," https://www.sqlite.org/src/info/0c5a7d1117cfb322.

[95] "Sqlite: Check-in [ea01d437]," https://www.sqlite.org/src/info/ea01d43788a75e39.

[96] "Sqlite: Check-in [efb20b9d]," https://www.sqlite.org/src/info/efb20b9da6c7cb31.

[97] "Sqlite: Check-in [14e6d19c]," https://www.sqlite.org/src/info/14e6d19c3157ccdc.

[98] "Sqlite: Check-in [036fc37a]," https://www.sqlite.org/src/info/036fc37a034093a4.

[99] "Sqlite: Check-in [ec36d15a]," https://www.sqlite.org/src/info/ec36d15a9e349f42.

[100] "Mysql," https://www.mysql.com/.

[101] A. S. Kaya and A. A. Sclçuk, "Post connection attacks in mysql," in *2020 5th International Conference on Computer Science and Engineering (UBMK)*. IEEE, 2020, pp. 1–6.

[102] B. Altintaş, "A security comparison of oracle, sql server and mysql database management systems against sql injection attack vulnerabilities," Master's thesis, Fen Bilimleri Enstitüsü, 2019.

[103] "Checkmarx," https://checkmarx.com/.

[104] G. J. Duck and R. H. Yap, "Effectivesan: type and memory error detection using dynamically typed c/c++," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 181–195.

[105] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "{HeapHopper}: Bringing bounded model checking to heap implementation security," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 99–116.

[106] H. Xu, W. Ren, Z. Liu, J. Chen, and J. Zhu, "Memory error detection based on dynamic binary translation," in *2020 IEEE 20th International Conference on Communication Technology (ICCT)*. IEEE, 2020, pp. 1059–1064.

[107] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 802–811.

[108] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 356–366.