SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Empirical Study of Memory Safety Bugs in Database Systems

Lan Ouyang

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Empirical Study of Memory Safety Bugs in Database Systems

Empirische Untersuchung von Speichersicherheitsfehlern in Datenbanksystemen

Author:	Lan Ouyang
Examiner:	Prof. Pramod Bhatotia
Supervisor:	Martin Fink, Ilya Meignan–Masson
Submission Date:	07/02/2025

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 07/02/2025 Lan Ouyang

Acknowledgments

First and foremost, I want to thank the systems research group and my supervisor Pramod Bhatotia for giving me the opportunity to work on this topic of memory safety bug analysis. And I would also like to express gratitude to my advisors Martin Fink, Ilya Meignan–Masson, and Dr. Dimitrios Stavrakakis for their guidance, feedback, and support throughout my research. Lastly, I would like to thank my family and friends for their support and company during my study.

Abstract

Memory safety bugs in databases—implemented predominantly in memory-unsafe languages like C/C++—pose critical risks to reliability and security, yet their characteristics, trends, and impacts remain understudied. Prior empirical work focuses on general software, leaving a gap in understanding database-specific vulnerabilities, their root causes, and mitigation strategies.

This thesis conducts a large-scale empirical study of memory bugs in seven databases (MySQL, SQLite, MariaDB, Redis, LevelDB, DuckDB, RocksDB). We collect and analyze over 150K bug reports and CVEs using automated scraping, keyword-based classification, and multi-dimensional analysis (bug type, severity, component, trends). The work finds out several interesting bug characteristics in modern database systems and shows that memory safety is not a solved problem in databases. This study provides actionable insights for database developers, maintainers, and researchers to take mitigation actions on memory safety problems.

The source code is available at https://github.com/TUM-DSE/db-bug-analysis

Zusammenfassung

Speichersicherheitsfehler in Datenbanken, die überwiegend in speicherunsicheren Sprachen wie C/C++ implementiert sind, stellen ein kritisches Risiko für die Zuverlässigkeit und Sicherheit dar, ihre Eigenschaften, Trends und Auswirkungen sind jedoch noch unzureichend untersucht. Frühere empirische Arbeiten konzentrieren sich auf allgemeine Software, was eine Lücke im Verständnis datenbankspezifischer Schwachstellen, ihrer Ursachen und Minderungsstrategien hinterlässt. In dieser Arbeit wird eine groß angelegte empirische Studie zu Speicherfehlern in sieben Datenbanken (MySQL, SQLite, MariaDB, Redis, LevelDB, DuckDB, RocksDB) durchgeführt. Wir sammeln und analysieren über 150.000 Fehlerberichte und CVEs mithilfe von automatisiertem Scraping, schlüsselwortbasierter Klassifizierung und mehrdimensionaler Analyse (Fehlertyp, Schweregrad, Komponente, Trends). Die Arbeit findet einige interessante Fehlereigenschaften in modernen Datenbanksystemen heraus und zeigt, dass die Speichersicherheit in Datenbanken kein gelöstes Problem ist. Diese Studie bietet umsetzbare Erkenntnisse für Datenbankentwickler, -betreuer und -forscher, um Maßnahmen zur Risikominderung von Speichersicherheitsproblemen zu ergreifen.

Den Quellcode finden Sie unter https://github.com/TUM-DSE/db-bug-analysis

Contents

1. Introduction

1.1. Context and motivation

Memory safety remains a critical challenge in software systems, with vulnerabilities stemming from memory errors accounting for a significant proportion of security breaches. Industry reports, such as those from Google's Chromium project[20] and Microsoft[1], reveal that 70% of severe security vulnerabilities are memory-related. Database systems, foundational to modern applications, are particularly vulnerable due to their widespread use of memory-unsafe languages like C and C++ for performance optimization. Despite rigorous testing, memory bugs persist, leading to crashes, data corruption, and exploitable vulnerabilities. Understanding the nature and prevalence of these bugs is essential to improving database resilience and reducing maintenance costs. This work presents a large-scale empirical study of memory safety bugs in database systems, aiming to uncover their characteristics, trends, and security implications to guide mitigation strategies.

1.2. State-of-the-art and research gap

Prior empirical studies have analyzed bugs in domains like machine learning systems[35], Apache projects[41], and Rust codebases[40]. However, these works lack focus on database systems, which face unique challenges such as ACID compliance, concurrency demands, and complex memory management. Existing studies also suffer from limited scope, analyzing small datasets or generic bug types. For example, Li et al. [16] observed declining memory bugs in general software, but their findings may not apply to databases. This gap leaves developers and researchers without actionable insights into database-specific memory bug patterns.

1.3. Problem statement

This study addresses the following question: What are the characteristics, trends, and impacts of memory safety bugs in modern database systems? We analyze seven widely used databases—MySQL, SQLite, MariaDB, Redis, LevelDB, DuckDB, and RocksDB—to identify patterns in bug frequency, type, severity, and exploitability. The goal is to inform tool development, secure coding practices, and prioritize maintenance efforts.

1.4. High-level approach and implementation overview

Our methodology combines automated data collection, keyword-based classification, and multi-dimensional analysis:

Bug Collection: We develop custom scrapers using the Scrapy framework and REST APIs to gather more than 150K+ bug reports and CVE entries from bug-tracking systems (e.g., Jira, GitHub Issues) and vulnerability databases.

Keyword Identification: A taxonomy of memory bug keywords (e.g., "buffer overflow," "use-after-free") is derived from sanitizer logs and CWE classifications, enabling automated tagging of bug reports.

Elasticsearch Integration: Data are stored and queried using Elasticsearch, leveraging its full-text search and aggregation capabilities for efficient analysis across four dimensions: bug type, temporal trends, severity/priority, and affected components.

1.5. Evaluation overview and impact summary

Our analysis of 5,321 memory bugs across seven databases reveals:

Dominant Bug Types: Overflow/underflow, memory leaks, and dangling pointers are most prevalent.

Security Impact: 70–80% of CVEs in Redis and SQLite are memory-related.

Temporal Trends: Memory bugs in most databases show no decline.

Component Hotspots: Server-side components (e.g., query processors) account for most of the memory bugs in MySQL and MariaDB.

This study provides actionable insights for database developers and maintainers to prioritize actions to high-severity bugs in critical subsystems and for researchers to explore memory-safe alternatives (e.g., Rust) for database internals.

1.6. Key Contributions

This thesis makes the following key contributions:

Large-Scale study: A comprehensive analysis of memory bugs in databases, spanning 20+ years and 150k+ bug reports.

Taxonomy and methodology: A reproducible framework for collecting, classifying, and analyzing memory bugs across diverse systems.

Memory safety insights Critical insights into the nature, trends, and impacts of memory safety bugs.

2. Background

2.1. Bug tracking systems

Bug-tracking systems are a critical component of modern software projects, where users can report bugs or issues with descriptions in detail, including how to reproduce, severity, priority, and attachments such as screenshots and logs. Bug-tracking systems help developers and project managers monitor and manage software bugs and issues throughout the development life cycle, making them essential for maintaining software quality and ensuring timely resolution of issues. Using bug-tracking systems provides insights into common issues and areas for improvement. To discover the impact of memory bugs on databases, we use these bug-tracking systems as the main source of bugs.

There are many popular bug-tracking systems or project management tools that can be customized for bug tracking. We introduce some of the bug-tracking systems in the following subsections.

2.1.1. Jira

Jira[15] is one of the most popular and widely used bug-tracking and project management tools. Jira allows users to create, manage, and track issues with detailed descriptions, including summary, description, reproduction steps, attachments, labels, and tags. Jira issues can be prioritized by low, medium, high, and critical, and severity levels can be assigned to help the team focus on the most crucial issues first and to work as an impact factor in our study of bugs. The workflow tracks the process status of each issue. The issue type field helps categorize issues into bug, epic, task, story, etc. MongoDB and MariaDB, for example, use Jira for project management and issue tracking.

Table 2.1.: Jira issue fields

Field	Description	Example
Project	The project that the issue belongs to.	`MongoDB Server`
Issue Key	Unique identifier combining the project key and number.	`SERVER-50775`
Summary	A short title summarizing the issue.	`Replication lag in sharded cluster under heavy write load`
Description	A detailed explanation of the issue and steps to reproduce.	`Secondaries in the cluster exhibit significant lag when high volume writes occur on the primary node.`
Issue Type	Category of work (e.g., Bug, Story, Task, Epic).	`Bug`
Status	Current stage in the workflow (e.g., To Do, In Progress, Done).	`In Progress`
Priority	Urgency or importance of the issue.	`Critical`
Resolution	How the issue was resolved.	(Unresolved)
Created	When the issue was logged.	`2025-01-05 11:00 AM`
Updated	Most recent modification time.	`2025-01-06 09:30 AM`
Resolved	When the issue was marked as resolved.	(Not set)
Comments	Log of discussion and notes.	`"Initial report by QA; observed replication delay on multiple nodes."`
Components	Subsystems or modules impacted by the issue.	`Replication, Sharding`
Labels	Tags or keywords for categorization and filtering.	`replication, performance`
Environment	Details of the hardware/software environment.	`Ubuntu 20.04, 64-bit server`

2.1.2. GitHub Issues

Another example of a popular bug-tracking system is GitHub Issues[12]. It is integrated with GitHub repositories, can be used to manage tasks, bugs, feature requests, and other work items, and thus becomes a convenient choice for projects already open-sourcing on GitHub. GitHub Issues is a simple and integrated solution. However, its advanced features are not as rich as those of a dedicated bug tracker like Jira in terms of workflow and agile support. Each issue includes a title, description, attachments, and labels (e.g., priority) for categorization, etc. The blend of different work items in GitHub Issues can be a problem for our study as we use automated scripts to collect and analyze bugs, and there is no good way to completely distinguish between bugs and others (e.g., feature and support requests).

Table 2.2.: GitHub issue fields

Field	Description	Example
Repository	The repository in which the issue was created.	`redis/redis`
Issue Number	A unique identifier for the issue within the repository.	`#13697`
Title	A concise summary of the issue.	`[BUG] unstable cannot run redisraft`
Description	A detailed explanation of the issue.	`The current unstable version cannot run redisraft, but it successfully ran after I switched to redis 7.2.5.`
State	The current status of the issue, either `open` or `closed`.	`open`
Security Status	Indicates if the issue is marked as a security vulnerability.	`Not a security issue`
Creation Date	The date and time when the issue was created.	`2024-12-23 10:00 AM`
Last Updated	The date and time of the most recent activity on the issue.	`2024-12-24 09:30 AM`

Continued on next page

Table 2.2 – continued from previous page

Field	Description	Example
Closed Date	The date and time when the issue was closed.	(Not set)
Comments	Discussions and additional information provided by collaborators.	`I encountered this issue on Windows 10. Here's the stack trace:...`
Labels	Tags used to categorize issues, such as `cluster` or `documentation`.	`crash report`

2.1.3. Custom bug-tracking systems

Many large-scale projects customize their bug-tracking systems instead of relying on third-party platforms like Jira or GitHub Issues. With the customization, projects have a bug-tracking system tailored to their specific needs, such as tight integration with internal workflows and development processes. Custom bug-tracking systems give projects full control over their data and security. We introduce only two systems used by MySQL and SQLite because we will collect bugs from these two for our later bug study. But there are more custom bug-tracking systems out there. For example, PostgreSQL [cite website] uses mailing lists and a publicly archived workflow to manage issues, and no web-based bug tracker.

MySQL Bugs system

MySQL, one of the well-known open-source relational database management systems, has a dedicated bug-tracking system[22] to manage and track issues, bugs, and feature requests. This system is web-based, publicly accessible, and works as an important platform for collaboration between users, developers, and the MySQL community.

Table 2.3.: MySQL bug report fields

Field	Explanation	Example
Title	A concise summary of the bug.	`Segmentation fault when using JOIN on large tables`

Continued on next page

Table 2.3 – continued from previous page

Field	Description	Example
Description	Detailed information about the bug.	The server crashes with a segmentation fault...
Status	The current status of the bug report, such as `Open`, `In Progress`, `Resolved`, `Closed`.	Open
Priority	The urgency assigned to resolving the bug, often correlated with severity.	High
Severity	The impact level of the bug on the system. Common levels include S1 (Critical), S2 (Major), S3 (Minor), S4 (Trivial), S5 (Feature request).	S1 (Critical)
Category	The component or area of MySQL affected by the bug.	InnoDB
Operating System	The operating system and version on which the bug occurred.	Ubuntu 20.04 LTS
CPU Arch	Details about the hardware environment, if relevant.	x86_64 architecture
Date Reported	The date when the bug was reported.	2025-02-01
Comments	Additional notes or discussions related to the bug.	A user noted that the issue does not occur in version 8.0.21.

SQLite Bug Reports system

SQLite, another widely used relational database management system, also maintains its own bug-tracking system [34]. The SQLite team developed a version control system, Fossil [10], to support the SQLite project itself. Aside from version control, the Fossil system also includes an integrated bug-tracking system to handle issues, feature requests, and bugs. A bug report in Fossil is called a "ticket." The creation and modification of a ticket are recorded as Ticket Change Artifacts.

The SQLite team takes a different approach to bug reporting. They realize that allowing anonymous users to enter new bug reports into their bug tracking system introduces a "high volume" of low-quality reports, including duplicates, poorly written bug reports that cannot be reproduced, and support requests disguised as "bug reports." SQLite users are asked to report bugs to the SQLite Forum. Registered developers will filter bug reports on the forum and transfer actual new bugs into the bug-tracking system. This approach significantly reduces the level of background noise and facilitates studies of bug density and software quality.

Table 2.4.: SQLite ticket fields

Field	Explanation	Example
Title	A concise summary of the issue or feature request.	`Assertion fault using indices with redundant columns`
Description	A detailed explanation of the issue.	`The following script causes an assertion fault:...`
Type	The nature of the ticket, such as `Bug`, `Feature Request`, or `Documentation`.	`Bug`
Status	The current state of the ticket, e.g., `Open`, `Closed`, `Fixed`, `Duplicate`.	`Open`
Priority	The urgency assigned to the ticket, indicating how soon it should be addressed.	`High`
Severity	The impact level of the issue, often categorized as `Critical`, `Major`, `Minor`, etc.	`Major`
Resolution	The outcome of the ticket, such as `Fixed`, `Won't Fix`, `Invalid`.	`Fixed`

Continued on next page

Table 2.4 – Continued from previous page

Field	Explanation	Example
Subsystem	The specific component or module of SQLite affected by the issue.	`Code_Generator`
Last Modified	The most recent date when the ticket was updated.	`2025-02-03`

2.2. CVE program

The Common Vulnerabilities and Exposures (CVE) program aims to "identify, define, and catalog publicly disclosed cybersecurity vulnerabilities." [5] When a researcher, vendor, or user finds a vulnerability, they submit the issue to a CVE Numbering Authority (CNA) who is responsible for assigning a unique CVE and publishing it. CVE provides a universal reference for vulnerabilities, enabling consistent communication of vulnerability data across organizations, tools, and databases.

Each CVE entry is published with a brief description of the vulnerability and references to advisories, patches, technical reports, etc. Vendors and databases like National Vulnerability Database (NVD)[23] add Common Vulnerability Scoring System (CVSS) [9] scores, patch links, and other metadata. The severity scores (e.g., CVSS) help prioritize fixes based on severity and exposure. For a detailed explanation of a CVE entry, see Table 2.5. The CVE program is the backbone for vulnerability management, yet it is limited in some aspects. It does not cover all the vulnerabilities; for example, some vulnerabilities are silently patched. Second, there is a lag between vulnerability discovery and CVE publishment to wait for CNA to act. And CVE entries lack context, so organizations must assess the relevance to their systems. To our bug study, the CVE database is a great source to study the impact of memory safety bugs.

GitHub Security advisories

As one of the GitHub Security features, GitHub builds Security Advisories[2] based on the CVE list and allows repository maintainers to publish security advisories to publicly disclose security vulnerabilities to the project's community after privately discussing and fixing them, making it easier for their community to research the impact of the security vulnerabilities. Unlike GitHub Issues, only repository administrators have permission to create a security advisory, which means everything there has been reviewed carefully and makes the security advisories a more reliable source for bug study than GitHub Issues. According to the vulnerability disclosure process, reported vulnerabilities are usually kept private until a patch is available. Therefore, we are not guaranteed to have a complete view of the serious bugs from security advisories for our study.

Table 2.5.: CVE entry core fields and enhancements

Field	Description	Example
Core Fields		
CVE ID	Unique identifier (format: `CVE-YYYY-NNNNN`)	`CVE-2014-0160`
Status	Entry state: `PUBLISHED`, `REJECTED`, etc.	`PUBLISHED`
Description	Neutral vulnerability summary	"OpenSSL TLS heartbeat extension permits..."
References	Links to advisories/patches	`https://www.openssl.org/news/secadv/...`
Assigning CNA	CVE Numbering Authority	`cve@mitre.org`
Enhanced Fields		
CVSS Severity	Criticality score (0–10)	`CVSS v3.1: 9.8 (Critical)`
Vulnerability Type	CWE classification	`CWE-119: Buffer Overflow`
Affected Products	CPE identifiers	`cpe:2.3:a:openssl:openssl:1.0.1`
Exploitability	Attack complexity	`Attack Vector: Network`
Patch Links	Fix URLs	`https://httpd.apache.org/patches/...`
Known Exploits	Exploit database IDs	`Exploit-DB: 31337`
Mitigation & Fixes	Mitigation steps	"Upgrade to OpenSSL 1.0.1g"

2.3. Memory safety bugs

When a program allows statement execution that reads or writes memory and the program is in a state where the memory access results in undefined behavior, memory safety bugs arise [26].

2.3.1. Classes of memory safety bugs

Memory safety bugs can be distinguished based on their possible solutions, their impact on performance, developer experience, etc. The primary classifications are spatial memory safety bugs and temporal memory safety bugs. Google [26] and Apple [4] have introduced more classes like type safety bugs, initialization safety bugs, and data-race/thread safety bugs.

Spatial memory safety bugs

This kind of bug occurs when a memory access refers to a memory location outside the intended bounds (e.g., an object's allocated region). Examples are buffer overflows, which access data beyond the allocated buffer limits, and array index out-of-bounds using indices that exceed the array's defined range.

Temporal memory safety bugs

These bugs arise when a memory access to an object happens outside of the object's lifetime. Common examples include use-after-free accessing memory after it has been deallocated and double-free attempting to free the same memory location multiple times.

Type safety bugs

Type safety bugs refer to cases when reading a value of a given type from memory that does not contain a member of this type, e.g., reading memory after an invalid pointer cast.

Initialization safety bugs

Initialization safety bugs arise when memory is read before being initialized. This can cause information disclosure and type/temporal safety bugs.

Data-race safety bugs

Unsynchronized reads and writes by different threads may lead to an object in an inconsistent state. When a bug arises from improper or missing synchronization but not anything mentioned above, we can classify it as a data-race safety bug.

2.3.2. CWE

Common Weakness Enumeration (CWE)[6] lists common software and hardware weaknesses with the help of the community. It identifies, categorizes, and describes vulnerabilities, including memory-related bugs, in a standardized way. CWE organizes weaknesses in a hierarchical structure, with parent categories (i.e., class weakness) and specific child entries (i.e., base weakness). But this relationship can vary in different "views." So, in the Table 2.6, we list memory-related class and base weaknesses together and roughly group them into 5 parts based on experience. These CWE entries can help us better understand existing memory bugs.

Table 2.6.: Memory-related CWE weaknesses (Abridged)

CWE ID	Weakness Name	Description
	Buffer Overflows and Out-of-Bounds Access	
CWE-119	Improper Restriction of Operations within Bounds	General category for memory access outside intended bounds.
CWE-120	Buffer Copy without Checking Size	Copying data to a buffer without verifying the source fits.
CWE-121	Stack-based Buffer Overflow	Overflowing a buffer allocated on the stack.
CWE-122	Heap-based Buffer Overflow	Overflowing a buffer allocated on the heap.
CWE-124	Buffer Underwrite	Writing to a buffer before its start location.
CWE-125	Out-of-bounds Read	Reading memory outside the bounds of a buffer.
CWE-787	Out-of-bounds Write	Writing memory outside the bounds of a buffer.
CWE-129	Improper Validation of Array Index	Using untrusted input as an array index without validation.
	Memory Allocation/Deallocation Errors	
CWE-400	Uncontrolled Resource Consumption	General category for resource exhaustion (e.g., memory leaks).
CWE-401	Memory Leak	Failing to release allocated memory.

Continued on next page

Table 2.6 – continued from previous page

CWE ID	Weakness Name	Description
CWE-415	Double Free	Freeing the same memory block multiple times.
CWE-416	Use After Free	Using a pointer after its memory has been freed.
CWE-789	Stack Exhaustion	Allocating memory without limits, leading to exhaustion.
CWE-590	Free of Memory not on the Heap	Calling ‘free()‘ on non-heap memory (e.g., stack or global variables).
CWE-908	Use of Uninitialized Resource	Using memory before it is initialized.
CWE-457	Use of Uninitialized Variable	Using a variable before initializing it.
CWE-824	Access of Uninitialized Pointer	Dereferencing a pointer before it is initialized.
	Pointer/Address Errors	
CWE-476	NULL Pointer Dereference	Dereferencing a NULL pointer.
CWE-468	Incorrect Pointer Scaling	Incorrect pointer arithmetic (e.g., wrong size calculations).
CWE-587	Assignment of Fixed Address	Assigning a fixed memory address to a pointer.
CWE-823	Use of Out-of-range Pointer Offset	Using an invalid offset for a pointer.
	Integer-Related Memory Issues	
CWE-190	Integer Overflow or Wraparound	Integer operations produce values outside expected bounds.
CWE-194	Unexpected Sign Extension	Sign extension errors during integer conversions.
CWE-131	Incorrect Calculation of Buffer Size	Miscalculating buffer sizes, leading to overflows.
	Other Memory Corruption Issues	
CWE-123	Write-what-where Condition	Attacker-controlled arbitrary memory write.

3. Methodology

This chapter describes the databases used in our bug study and our research questions. To answer these research questions, we should analyze thousands of bugs. To reduce the manual effort, we have developed bug trackers to collect bug reports from different bug-tracking systems and a keyword-matching approach to identify and classify memory bugs.

3.1. Problem statement

To understand the impact of memory safety bugs on database systems, we investigate all the bugs reported on the bug tracking system of 7 databases, identify those memory safety-related ones by keywords, and perform statistical analysis to determine memory bug characteristics, including bug distribution over time and in different software components.

To be more specific, we are interested in evaluating the following research questions:

RQ1: How often do memory bugs appear in database systems?

RQ2: What kind of memory bugs appear in database systems?

RQ3: Are memory bugs diminishing in database systems in the past years?

RQ4: How severe are various kinds of memory bugs?

RQ5: How urgent are memory bugs in database systems?

RQ6: How are memory bugs distributed in different components in the database system?

RQ7: How many memory bugs in database systems can be exploited to compromise security?

3.2. Subject databases

We study bugs from seven databases. MySQL[21] is a relational database that is well known for its speed, scalability, and strong support for ACID transactions and powers many web applications. SQLite[33] is a lightweight, serverless relational database that is stored in a single file, an ideal option for embedded systems, mobile applications, and browsers. MariaDB[19] is a 100% open-source drop-in replacement for MySQL with improved performance and enhanced security. Redis[27] is an in-memory key-value store that enables ultra-fast caching, message queues, and real-time analytics and supports persistence and advanced data structures. LevelDB[11] is a high-performance, key-value store designed for fast writes and reads. DuckDB[7] is a columnar relational data store optimized for Online Analytical Processing (OLAP) and fast analytical queries. RocksDB[30] is a key-value store that is built on LevelDB but optimized for low-latency, high-throughput workloads. All the databases are open-source and implemented in C or C++, which grants low-level control and high performance, especially memory efficiency. Still, they are also memory-unsafe languages and, therefore, very prone to memory safety bugs. We select these databases, so that we have a rich source of memory bugs to study.

3.3. Bug collection

All the seven databases have a bug-tracking system. We develop 4 bug trackers for 4 types of bug-tracking systems that are used by the databases to collect bugs through either Representational State Transfer (REST) Application Programming Interface (API) or web scraping technique and store them in Elasticsearch for reproduction.

Collection of security vulnerabilities. We collect all the security vulnerabilities that are related to the seven databases in the Mitre CVE database[5].

3.4. Memory safety bug identification

Bug reports do not explicitly indicate what type a bug is, and checking them one by one is not feasible due to the massive amount of bugs. Bug reporters describe the bug they encountered in the title and body of the report; other users discuss the bug in the comments. Although bug reports usually do not have a field dedicated to the bug type, if the bug type is known, users will mention it in the title, description, or comments. via keyword matching, a reported bug is determined as a memory bug if any of the title, description, and comments field contains a memory safety keyword(see Table 3.1). The heuristic is simple, and some previous studies (e.g., [41]) used the same technique to extract interested documents. Elasticsearch provides a broad range of queries that allow us to do keyword matching against the stored bug reports.

The problem with this approach is that the keyword appearing in a bug report only shows that this report discusses a memory safety-related topic but does not exactly "define" a memory safety bug, so we may identify some non-memory bug as a memory bug. On the other hand, if we do not have an exhaustive collection of all the possible keywords, we may leave out some memory bug reports. So, false positives and false negatives are both possible with the keyword-matching approach. However, after checking some sample bug reports, we assume only a small margin of error.

Another possible approach is using AI models, such as meta Llama[18]. We can feed the model with bug reports and ask for the type of bug. Again, it does not assure 100% accuracy.

3.5. Memory safety keywords

We derive a taxonomy of memory safety bug keywords from sanitizer logs and CWE classifications focusing on the most common memory bug types, enabling automated tagging of bug reports as shown in Table 3.1.

If we find one of the keywords in the title, description, or comments in a bug report, we label it as a memory safety bug and classify it as the bug type that the keyword belongs to.

3.6. Irrelevant bug reports identification

Not every bug report in the bug-tracking system is valid. Invalid bug reports include duplicates, non-reproducible bugs, not bugs, external bugs, issues unsupported by the team, incomplete reports, etc. Among all kinds of information attached to the bug reports, the resolution/status/type field helps identify and exclude invalid bug reports.

In addition, we can exclude those reports for documentation instead of real software by the component/subsystem/project field, as those are not bugs for sure.

As a supplement to bug reports that do not have those fields to determine the relevance or the fields are not updated to reflect the real case of the bug reports, we also match phrases such as "this is duplicate of," "this is not a bug," and "feature request" in the title, description and comments of bug reports.

3.7. Analysis dimensions

We analyze memory bugs in 4 dimensions:

Bug type analysis gives us the distribution of memory bugs with different types and the change of distribution over the years, with different impacts, and in different components.

Date of creation analysis shows the trend of different categories of memory bugs.

Impact(including severity and priority) analysis produces the distribution of different impacts with the corresponding bug types.

Table 3.1.: Memory bug types and keywords

Type	**Keyword**
Memory Leak	memory leak
Over-/underflow	overflow
	buffer overflow
	buffer overrun
	buffer underflow
	buffer underrun
	out of bounds
	array bounds
	stack overflow
	stack smashing
	integer overflow
Null Pointer Dereference	null pointer dereference
	null pointer exception
Double Free	double free
	multiple free
Uninitialized Memory	uninitialized memory
	use of uninitialized
	wild pointer
Dangling Pointer	dangling pointer
	dangling reference
	use after free
	UAF
	stack use after scope
	stack use after return
Others	memory safety
	memory corruption
	heap corruption
	stack corruption
	memory error
	invalid memory access
	illegal memory access
	segmentation fault
	SIGSEGV
	address sanitizer
	ASLR

Software component analysis offers insights into where the dominant memory bugs are located in software components and how different components are affected by memory bugs.

4. Implementation

4.1. Web crawling and scraping

MySQL and SQLite use their dedicated bug-tracking systems that do not provide any interface other than web pages for people to get their bug data. So, we use web crawling and scraping techniques to collect the bug data from these two databases.

There are some differences between web crawling and web scraping. Simply speaking, web crawling is to iteratively find and fetch web links starting from some seed URLs; web scraping is to extract data from a web page. In our implementation, web crawling gets us from a bug list page to bug detail pages and, after finishing processing one bug list page, to the next bug list page. Web scraping lets us extract the bug information we need from each bug detail page.

MariaDB uses Jira as a project management and issue-tracking system; Redis, LevelDB, DuckDB, and RocksDB use GitHub's built-in issue-tracking system GitHub issues. Both Jira and GitHub provide API to download issue data.

4.1.1. Techniques and frameworks

Naive solution

HTTP library for downloading web pages coupled with HTML parsing library for parsing HTML and extracting data. Python offers various kinds of libraries for sending HTTP requests and receiving responses to load target web pages, such as Requests[28] and urllib3[36], and other libraries to extract data from HTML documents, such as pyquery[16] and BeautifulSoup[29].

This solution gives us the most straightforward and most flexible web scraper. We can gain better performance by reusing the underlying TCP connection. The downside is that it offers only the basic functionality that scrapes one page, and we need to take care of following all the necessary links and implement advanced features if we need any, for example asynchronous requests and concurrency. It only handles static web pages and is not suitable for websites that require JavaScript to load.

Web crawling frameworks

Frameworks like Scrapy[31] and PySpider[42] contain a rich set of functionalities, including web crawling, data extraction, and processing. They also often do some optimizations to extract a large amount of data in a short time.

This solution is still flexible and much more powerful than the naive one. Scrapy even has middleware that we can take and "plugin" to fit our needs. So, it can deal with complex web scraping tasks easily. However, this can be redundant if we are not doing that complex task and may slow down the program unnecessarily.

Headless browsing

For websites that heavily use JavaScript and Ajax, scraping tools like Scrapy and BeautifulSoup that only retrieve static pages on their own are in trouble. In this case, headless browsing tools like Selenium[32] can be very helpful. Headless browsing allows us to render a web page the same way a normal browser would do, including executing JavaScript code and Ajax calls.

Although Selenium combined with Chrome in headless mode can do any kind of scraping we would like, it takes plenty of memory and CPU resources, especially when several instances are run concurrently.

Cloud-based solutions

Cloud-based web scraping tools are no-code or low-code solutions for web scraping, easy to use, and save time to learn. By leveraging the cloud infrastructure, they offer scalability and speed without resource concerns.

However, these benefits come with good costs, limited customization, and flexibility to tailor the scraping and later data processing to specific needs.

Using REST APIs

Some websites provide APIs for developers and administrators to interact with their resources via Uniform Resource Identifier (URI) paths. By looking into the API documentation and finding the corresponding API for querying the specific resources, we can easily extract the data we want. Using APIs is often more efficient than scraping HTML pages and works well if the website changes its layout. A downside of this approach is that there are usually rate limits imposed on API requests.

4.1.2. Scrapy framework

We choose Scrapy to scrape bug report data on MySQL and SQLite bug tracking platforms. Scrapy framework has powerful supporting features that save us effort in implementing our own and reserve enough flexibility for developers to define customized scrapers, especially if we want to use ElasticSearch[13] as data storage.

Scrapy architecture

The Scrapy system has 7 main components: engine, scheduler, downloader, spider, item pipeline, downloader middleware, and spider middleware. Data, including requests, responses, and scraped items, are driven by the engine and flow among these components.

The engine gets initial requests from the spider, schedules them with the help of a scheduler, and sends scheduled requests to the downloader. After downloading the web page from the internet, the downloader generates a response and sends it to the engine, which sends it further to the spider for processing and producing scraped items and new requests. Scraped items and new requests pass through the engine and go to the item pipeline and scheduler. Middlewares sitting between the downloader (spider) and engine process data passes around.

In our implementation, we customize the spider to perform our scraping tasks and an item pipeline to store scraped items in the ElasticSearch database.

4.1.3. Scrapy-based web spider

We implement 2 spiders to scrape bug reports from MySQL and SQLite. The custom spider inherits from the `scrapy.Spider` class and starts crawling a specified Uniform Resource Locator (URL) which retrieves all the MySQL/SQLite bug reports. After performing the initial requests to the start URL, a search results page is received as a response and parsed in the callback function `parse`. We define the behavior of `parse` function to extract all the bug report links from the search result page, follow each bug report link to scrape detailed information from the bug report page using Cascading Style Sheets (CSS) selector and XPath, and generate items with the parsed data. Items passing through the engine reach the item pipeline; here, we use the Scrapy-Elasticsearch pipeline[39] to store items in Elasticsearch.

4.1.4. Scraping using REST API

Jira and GitHub both provide a library for easy use of Jira/GitHub REST API from Python[3, 24]. After being authenticated by Personal Access Token (PAT) and connected to a Jira server or GitHub repository, we get all the issues through the corresponding call and store relevant issues fields in a dictionary.

To index and store the issues in Elasticsearch, we implement a helper class `ElasticSearchStore` based on Elasticsearch Python client[8] to index issues in bulks.

4.1.5. Scraping rate limiting

Without any restriction, Scrapy Spider can easily crawl about 20 web pages per second due to asynchronous and non-blocking I/O and multi-threading. Many websites are equipped with anti-bot techniques. Keeping sending requests at this speed from the same IP address, our spider can be easily detected by bug tracking systems, gets blocked, and only receives "403 Forbidden". To slow down the Scrapy Spider and avoid being banned, we set `DOWNLOAD_DELAY` to a small number(0.6s-1s) to introduce a delay between requests or enable `AUTOTHROTTLE` so that Scrapy slows itself down based on website response times.

To prevent abuse and ensure the API availability, GitHub and Jira also impose rate limits on API requests. Jira API does not apply strict rate limiting, but when the endpoint is under great pressure, it returns a rate limit response with status code 429 to indicate the applications to delay the requests. GitHub API, on the other hand, has a strict rate limit of 5000 requests per hour[25]. To not exceed the rate limit, the API-based bug trackers sleep for 0.3 seconds between requests.

4.2. Elasticsearch storage and query

We store the scraped bug reports using Elasticsearch for reproduction and analysis purposes. Elasticsearch is a No-SQL data store that stores data as JSON documents and allows flexible schema[14]. And Elasticsearch as a search and analytics engine, allows us to leverage its powerful full-text search capability and robust data aggregations to do nearly real-time analysis. All interactions with Elasticsearch document data can be done via a RESTful API or through language-specific client libraries (e.g., Elasticsearch Python Client).

Elasticsearch adopts a distributed architecture, with clusters made up of nodes and indices divided into shards, allowing for parallel processing and distributed storage, thus ensuring scalability and availability. As a core component of Elastic Stack, Elasticsearch, combined with Logstash, Kibana, and Beats, is a popular and powerful tool for logging, monitoring, and security analytics. However in our implementation, we are not using these advanced features of Elasticsearch but just basic storage and analytics.

4.2.1. Indices and mappings

An Elasticsearch index is a logical namespace that stores a collection of documents. A document is a collection of fields. Each field is a key-value pair holding the data.

Mapping defines how a document and its fields are stored and indexed. Each field has a field data type that indicates the kind of data the field contains and affects the search behavior, space usage, and performance characteristics. A mapping definition contains a list of fields pertinent to the document and some metadata fields related to document identity and source.

Without any mapping definition, Elasticsearch uses dynamic mapping to automatically detect the data type of fields based on their JSON type following the default mapping rules. For example, strings matching any of the specified date patterns are mapped to date type, numeric strings are mapped to float or long type, and other strings are mapped to text type. Dynamic mapping also allows customization of mapping rules and default date formats.

Instead of dynamic mapping, we use explicit mapping in our Elasticsearch indices to define exactly how data types are mapped to fields. Otherwise, all of the fields are mapped to text type by default in our case. The text type is for indexing full-text values, such as the body of an email and the description of a bug. A text field is analyzed by an analyzer to convert the string into a list of individual terms before being indexed. This analysis enables individual word searches within each full-text field. So, text type is best suited for unstructured content and cases needing full-text search. While we do need full-text searches for bug report titles and comments, we also want to do exact term matches and aggregations on many fields. That requires keyword type, which is used for structured content like IDs, status, and subsystems and supports term-level queries to find documents based on precise values in structured data. Besides, we also need to define custom date formats using explicit mapping to match the dates we have scraped from the web.

We build up 7 indices, each storing bug reports of a database. For mappings that we define for each index, see Table 4.1.

Table 4.1.: Data type mappings of bug reports fields for 7 databases

Field	MySQL	SQLite	MariaDB	Redis, RocksDB, LevelDB, DuckDB
id	keyword	keyword		
issue_key			keyword	keyword
title	text	text	text	text
comments	text	text	text	text
submitted	date			
created_at			date	date
last_modified		date		
severity	keyword	keyword		
priority		keyword	keyword	
resolution		keyword	keyword	
state				keyword
status	keyword	keyword	keyword	
type		keyword		
labels				keyword
component	keyword			
subsystem		keyword		
project			keyword	
cpu_arch	keyword			
os	keyword			

4.2.2. Elasticsearch query

We use full-text queries to search for memory bug keywords as listed in Table 3.1 in bug report titles and comments(including the description of the bug and responses and discussions of users). Full-text queries search analyzed text field with a query string that is processed using the same analyzer that was used on the field when indexing.

Match query is the standard query for performing a full-text search. The provided query text is analyzed and broken into tokens before matching. The match query then searches for one or more matches of these tokens in the documents.

Match-phrase query is similar to match query and also analyzes the query text into tokens. However, unlike match queries, tokens must appear in the exact order and consecutively in documents to be seen as a match. Match-phrase query allows us to search for memory bug keywords as a phrase instead of looking for words one by one in documents.

Multi-match query performs match query or match-phrase query on multiple fields and returns documents that can find a match on any of the specified fields running the match query. Using multi-match, we can search for memory bug keywords in the title and comments fields at the same time.

To analyze the collected data, we need Elasticsearch aggregations to summarize our data as metrics, statistics, or other analytics to gain some insights from the data. Elasticsearch provides 3 categories of aggregations: metric aggregations, bucket aggregations, and pipeline aggregation. We only use the bucket aggregations, which group documents into "buckets" based on some criterion.

Terms aggregation groups documents based on unique values of a field. With terms aggregation, we can, for example, aggregate bug reports by their priority and count how many bugs have been reported with each priority level.

Date histogram aggregation groups documents into time-based intervals based on a date field. By specifying a calendar-based interval of year, we have bug reports grouped year by year and count bugs reported every year to see the yearly trend.

4.3. Security vulnerabilities

We download all the CVE records in JSON format from 1999 to 2024 from the CVE website. We scan over the CVE records to find records that are related to the 7 databases we study. There are no decisive fields in CVE records that we can use to tell the relevance. So, we search for the database names in CVE JSON documents and store each matching record's CVE ID, descriptions, CVSS scores, and references. For all databases except MySQL, we find a moderate number of matching CVE records and manually inspect those records. The number of records for MySQL is over 2000, so we only use a script to obtain the analytics.

Moreover, we find that using the "MySQL" keyword to search CVE records returns many irrelevant records, and it is not feasible to remove them manually. To partly address this problem, we make use of the product status information, which states the affected vendors and products. The automated script checks if the product status information is provided and if the "MySQL" keyword can be found in this information. If the product status information is provided but the "MySQL" keyword is not found in this information, we drop the CVE record.

We tried to use the URLs in the reference list attached with each CVE record to find out how many security vulnerabilities recorded in the CVE database are also reported in the database bug tracking system.

After obtaining a collection of database-related CVE records, we investigate the distribution of memory safety issues in the CVE records in the same way as we do to the bug reports scraped from each database's bug tracking system.

We want to study the severity of the database security vulnerabilities by analyzing the CVSS scores assigned to the CVE records. But because CVSS score is not a core field of CVE record and needs to be assigned by CVE partners after the vulnerability is published and carefully checked, most of the mined CVE records do not have a CVSS score. So, we can not get a very representative analysis result out of it.

Besides the CVE program, Redis lists reported security vulnerabilities in its GitHub repository, and SQLite also provides a table of recent CVEs affecting SQLite, each containing a few dozens of CVEs, which can be easily manually checked.

5. Evaluation

This chapter presents our experimental setup, analysis results, and the answers our analysis gives to the questions.

5.1. Experimental setup

We install and run Elasticsearch 7.17.16 on the local machine `irene`. Bug reports collected from 7 databases are stored in 7 indices separately. Table 5.1 lists the number of collected bug reports from each database and the date ranges. The analysis is performed on these bug reports stored as Elasticsearch documents with the help of Elasticsearch query and analytics capability. Although we collect all the available bug reports into storage, we identify and remove irrelevant reports using the methods mentioned in Chapter 3 when running queries. Hence, the number of total reports in each database index returned in query results is smaller than the number in Table 5.1.

Table 5.1.: Number of collected bug reports and their date ranges

Database	Index	Number of documents	start date	end date
MySQL	mysql-issues	92683	12/09/2002	31/12/2024
SQLite	sqlite-issues	794	18/08/2009	16/09/2024
MariaDB	mariadb-issues	34200	17/02/2009	05/11/2024
Redis	redis-issues	13248	23/01/2010	10/07/2024
LevelDB	leveldb-issues	1160	28/08/2014	10/07/2024
DuckDB	duckdb-issues	500	09/08/2024	10/09/2024
RocksDB	rocksdb-issues	12814	15/11/2013	10/09/2024

5.2. RQ1: Memory bug frequencies

We calculate the number of different types of memory bugs from seven databases as shown in Table 5.2. We use acronyms for bug types in tables: Memory Leak (ML), Over-/underflow (ODF), Null pointer dereference (NPD), Double Free (DF), Uninitialized

Table 5.2.: Number of bugs per type in 7 databases

Database	ML	ODF	NPD	DF	UM	DP	Others	All memory bugs	Total
MySQL	322	536	41	52	67	49	1026	1907	43706
SQLite	2	11	1	0	0	4	5	23	624
MariaDB	228	810	18	62	113	596	872	2076	24527
Redis	143	235	25	22	3	28	480	827	11838
LevelDB	11	11	1	0	0	2	18	42	1056
DuckDB	1	3	0	1	0	1	10	16	488
RocksDB	81	85	1	12	3	57	209	413	12266

Memory (UM), and Dangling pointer (DP). In terms of absolute numbers of memory bugs, MariaDB and MySQL, with numbers around 2000, are the highest, way more than the others. By dividing the number of memory bugs in a database by the total number of bugs in the database, we find that MariaDB has the highest frequency of memory bugs (8.46 memory bugs/100 bugs), followed by Redis(6.88 memory bugs/100 bugs) and MySQL(4.36 memory bugs/100 bugs). SQLite, LevelDB, DuckDB, and RocksDB have very similar numbers(3 memory bugs/100 bugs). These numbers seem to be lower than those shown in previous studies like [17]. A possible reason may be that we do not have an exhaustive list of keywords to identify as many memory bugs as possible, and irrelevant bug reports can not be excluded completely.

5.3. RQ2: Memory bug types

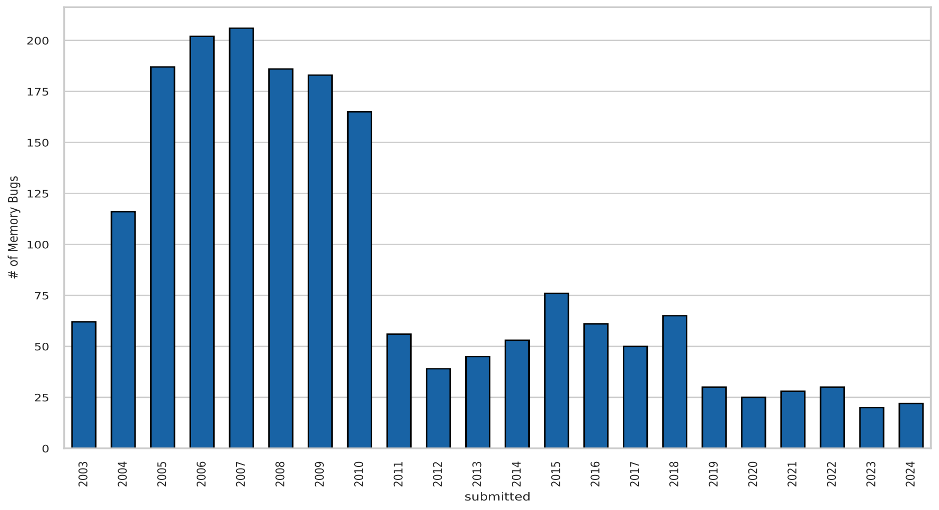
We automatically classify memory bugs into different types based on the matching keyword found in the bug reports. The distribution of memory bugs based on the 8 categories is also shown in Table 5.2. Among the 7 specific categories, most memory bugs are categorized as over-/underflow (i.e., out-of-bounds access), followed by memory leak and dangling pointer. Nearly half of the memory bugs fall under the others category, which means general phrases are widely used to describe memory safety problems.

5.4. RQ3: Memory bug trends

Based on the year of creation of bug reports (the year of last modification for SQLite bug reports), we count the number of reported memory bugs every year for each database. Figures 5.1 to 5.7 show the yearly trends of memory bug reporting for seven databases. MySQL has fewer memory bugs reported in recent years than before. Redis and RocksDB do not see a decrease in memory bugs, but there is a bump around 2021 and 2022. So does LevelDB, except for a spike in 2014 when the bug-tracking system was just set up. For SQLite and MariaDB, memory bug reports seem to have increased over the years; the trend is especially obvious in MariaDB. But this does not necessarily imply low software quality. Other factors, like the popularity of the database and growing concerns of memory safety, may affect the number of bug reports. DuckDB has only one year of record; no trend can be analyzed for it.

In terms of specific memory bug types, over-/underflow, and memory leaks are dominating in most of the years and in most databases. In MariaDB, dangling pointer bugs have been reported more frequently in recent years.

MySQL



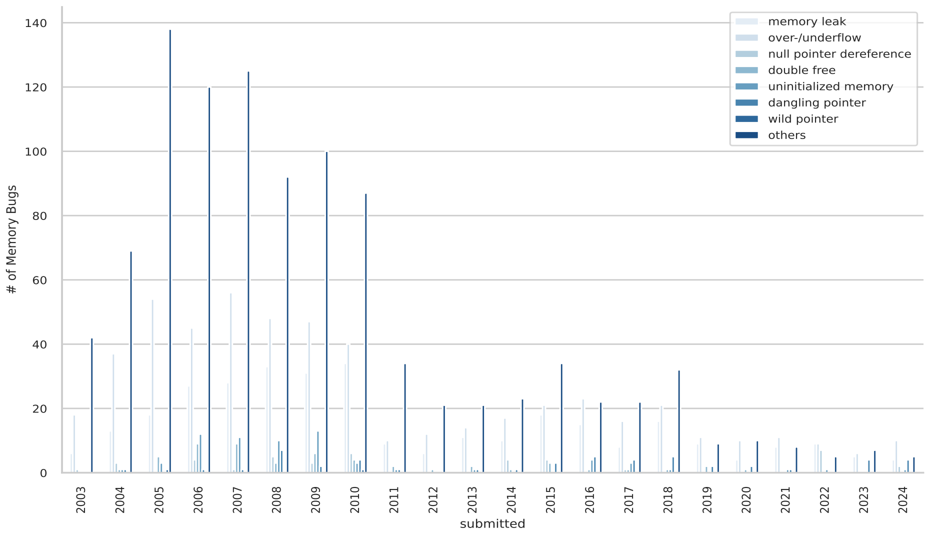
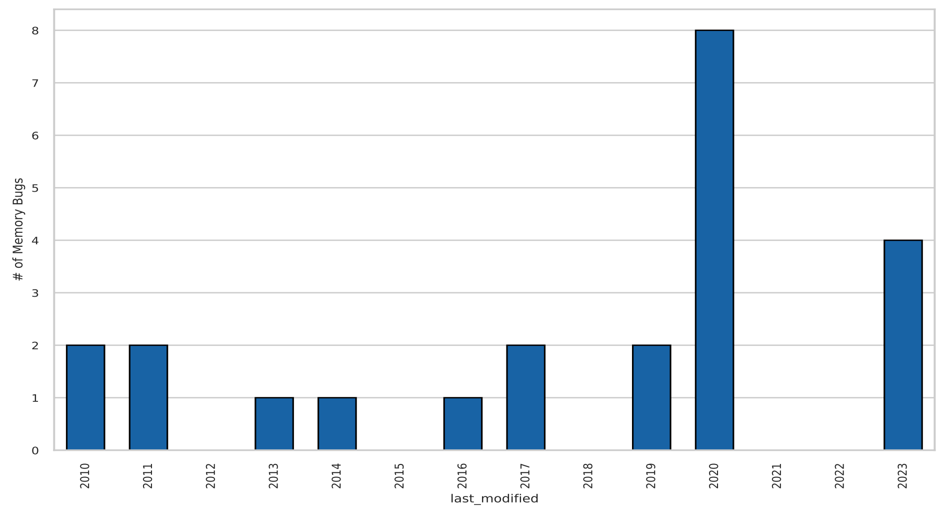


Figure 5.1.: Number of memory bugs from MySQL per year

SQLite



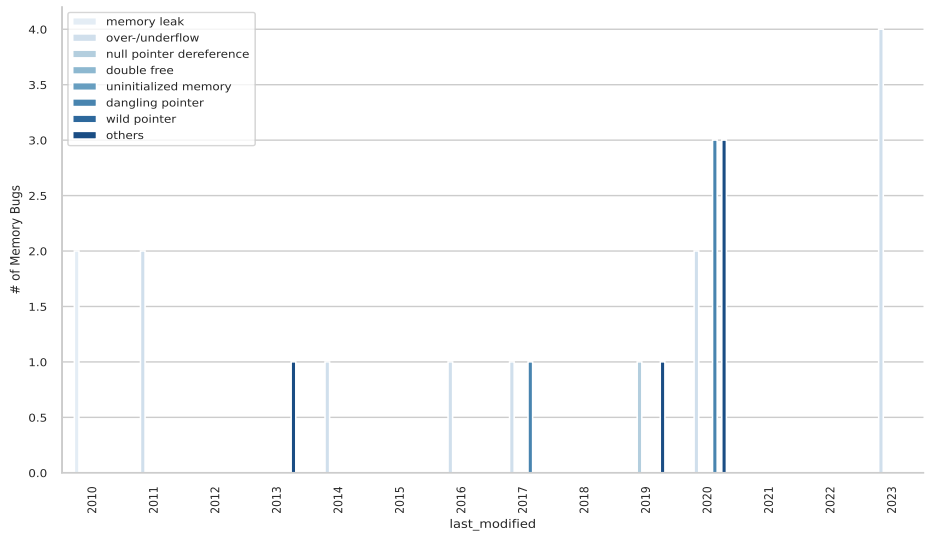
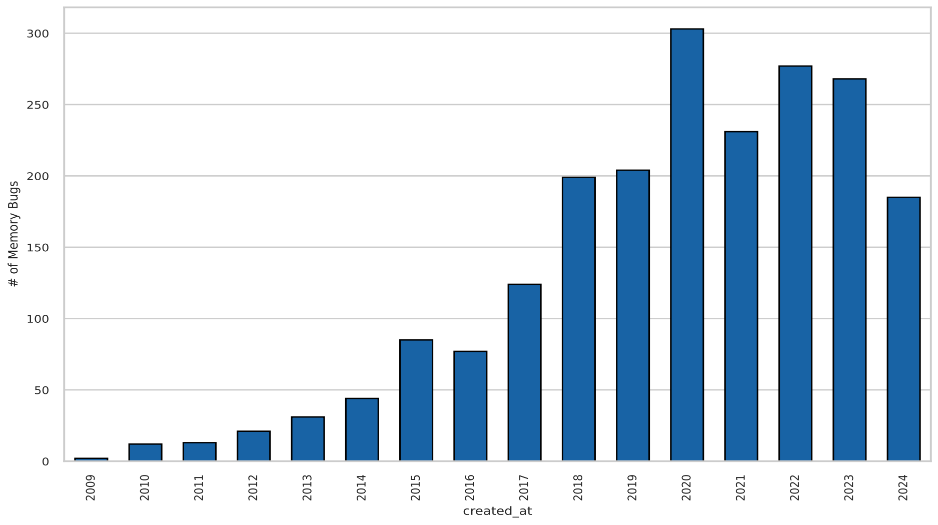


Figure 5.2.: Number of memory bugs of SQLite per year

MariaDB



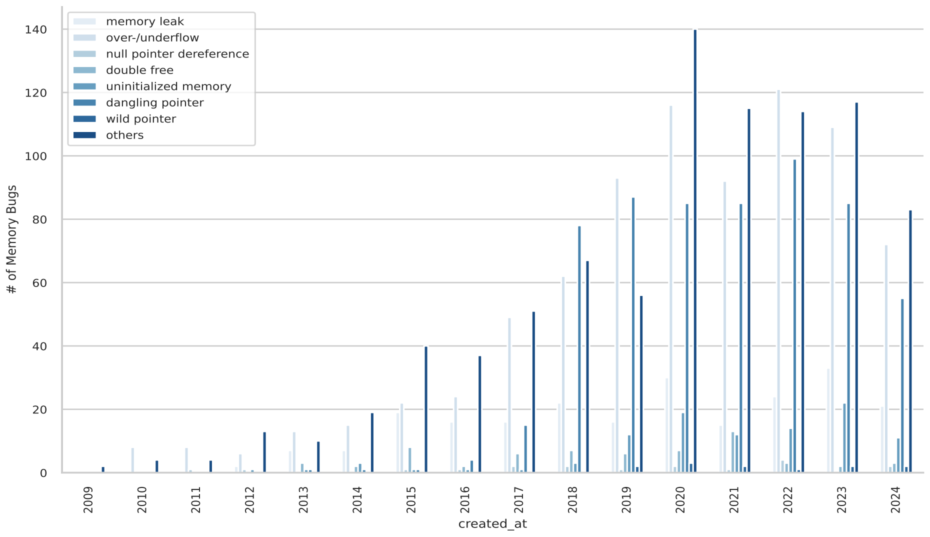
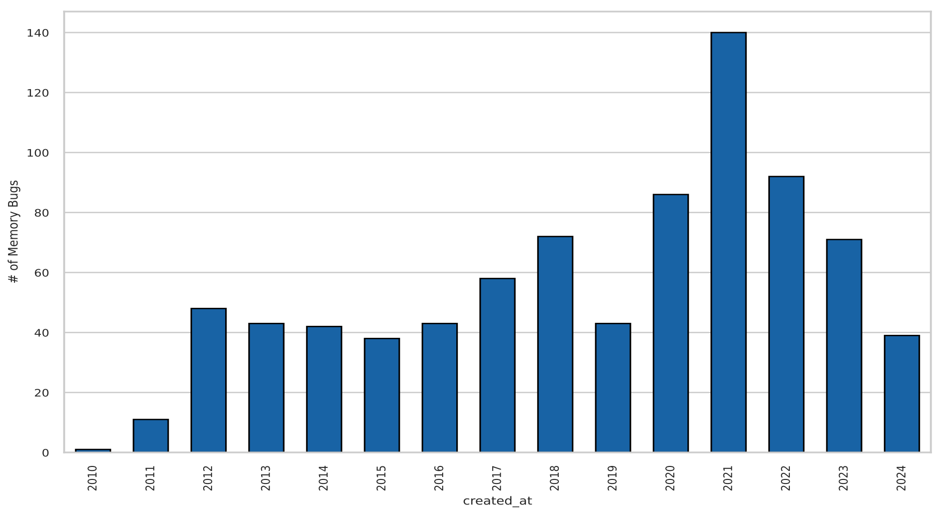


Figure 5.3.: Number of memory bugs from MariaDB per year

Redis



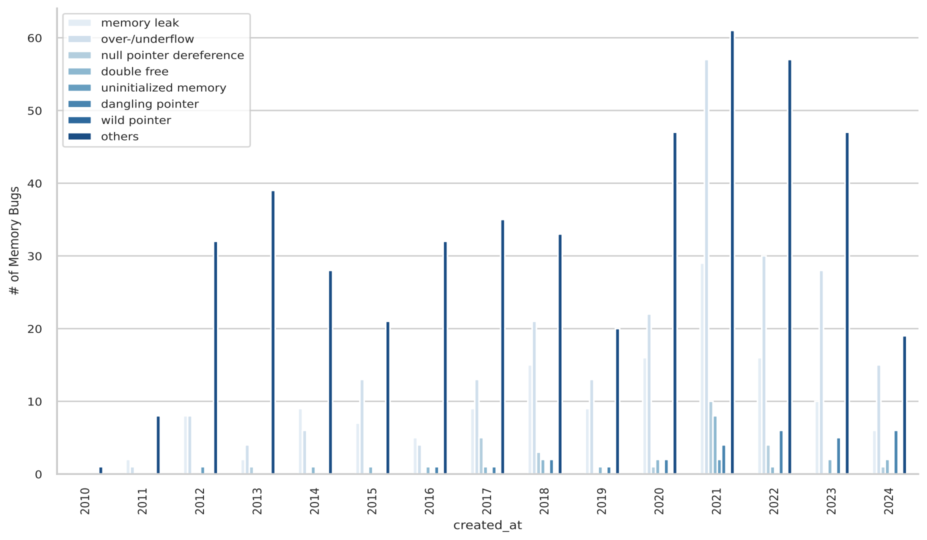
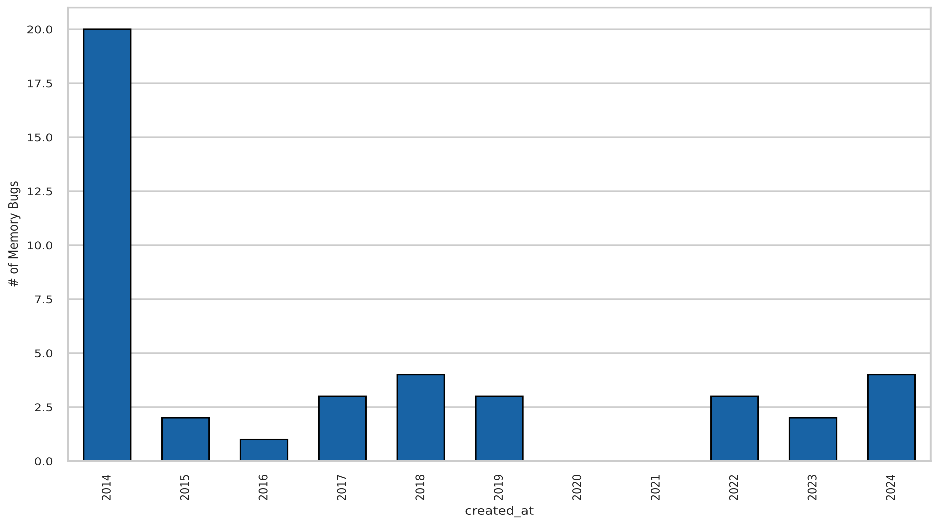


Figure 5.4.: Number of memory bugs from Redis per year

LevelDB



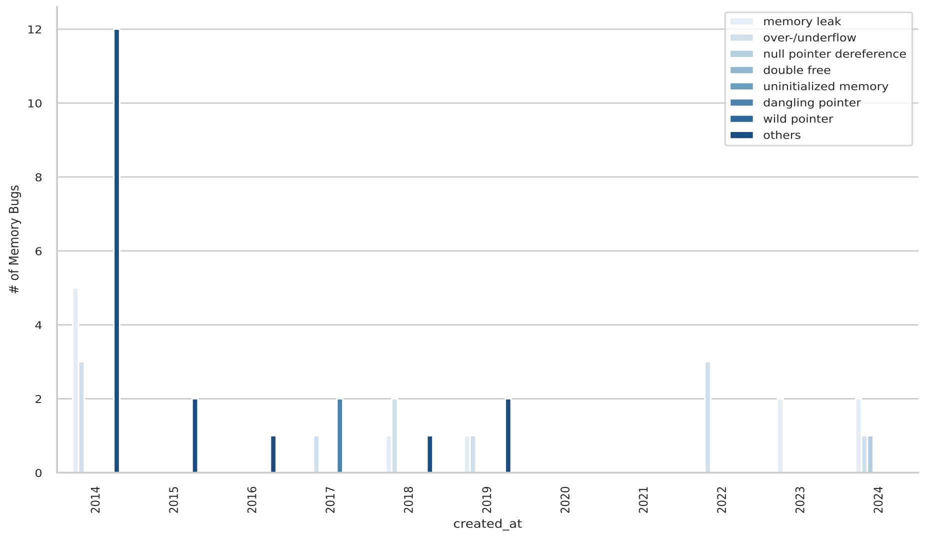
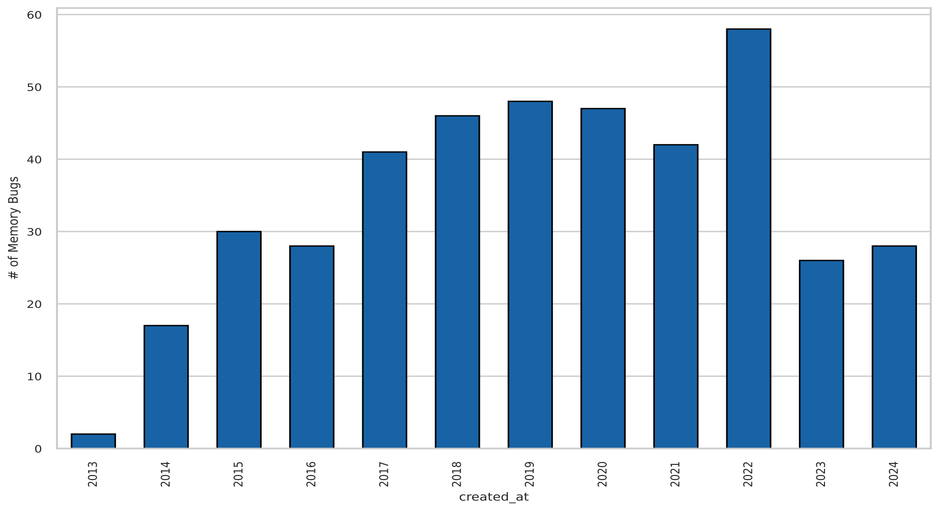


Figure 5.5.: Number of memory bugs from LevelDB per year

RocksDB



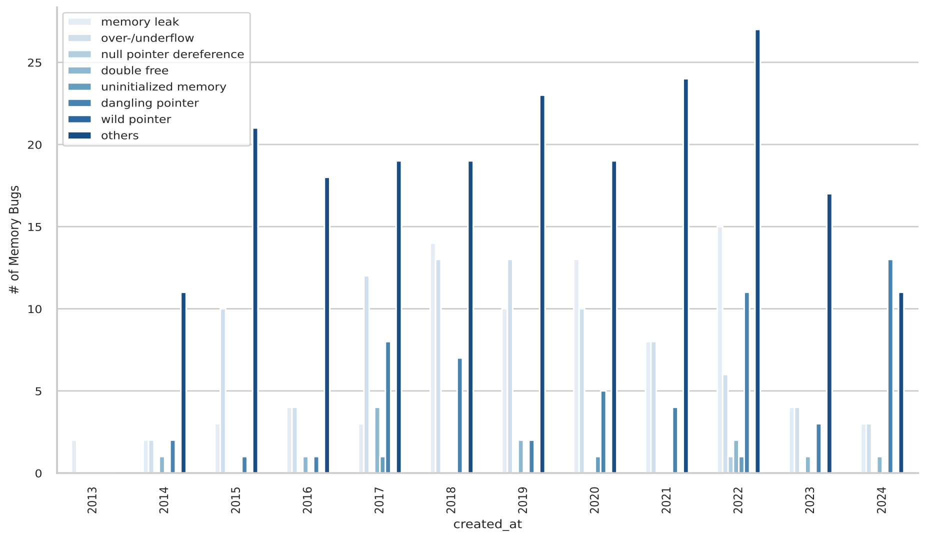


Figure 5.6.: Number of memory bugs from RocksDB per year

DuckDB

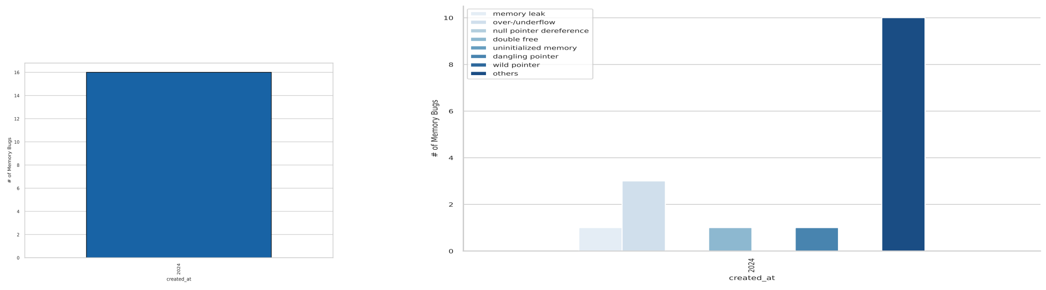
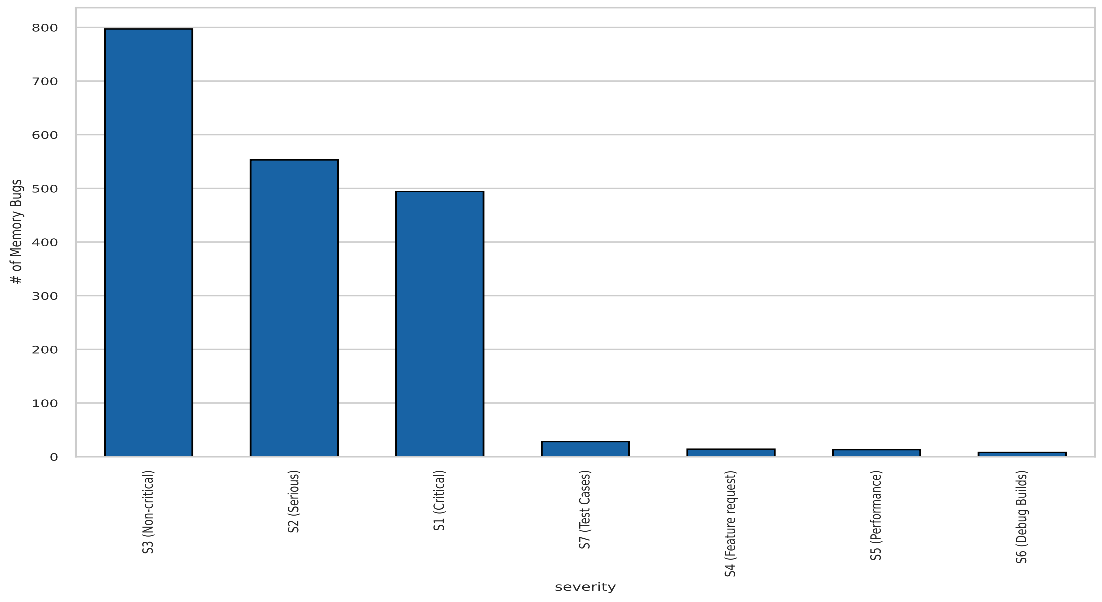


Figure 5.7.: Number of memory bugs from DuckDB per year

5.5. RQ4: Memory bug severity

Only MySQL and SQLite bug reports contain a severity field. MySQL has seven classes of severity, from S1 to S7: critical, serious, non-critical, feature request, performance, debug builds, and test cases; severity levels below S3(non-critical) are not bugs actually, whereas SQLite has only four classes of severity, critical, severe, important, and minor. MySQL memory bugs concentrate in S1-S3, which corresponds to the severity definition. More than half of the memory bugs are labeled as serious and critical in MySQL or more than important in SQLite. Over-/underflow and memory leaks still make up the most part of memory bugs across different security levels.

MySQL



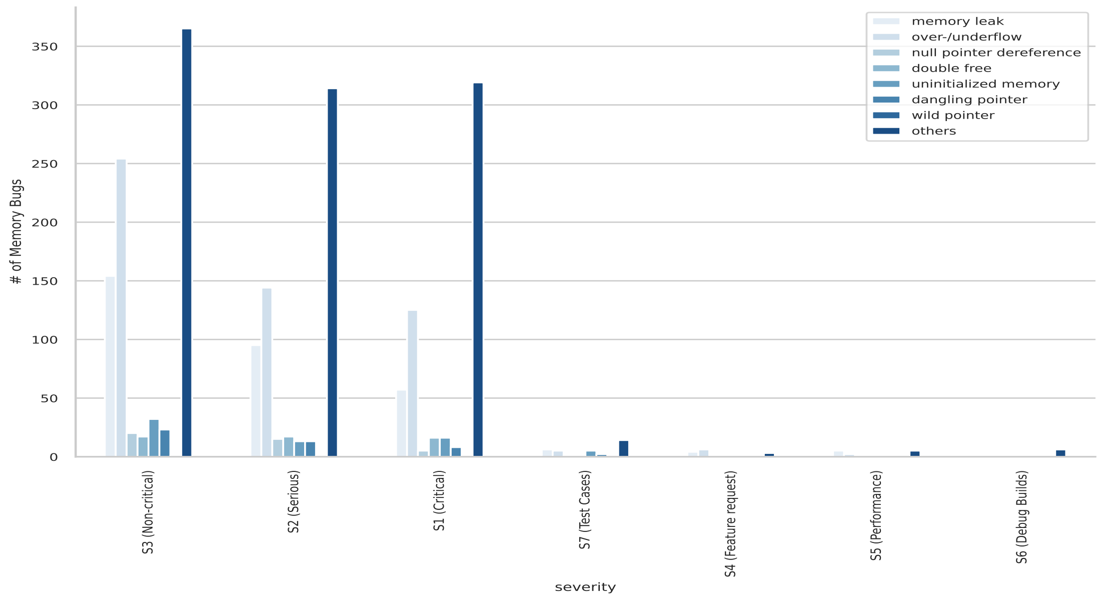
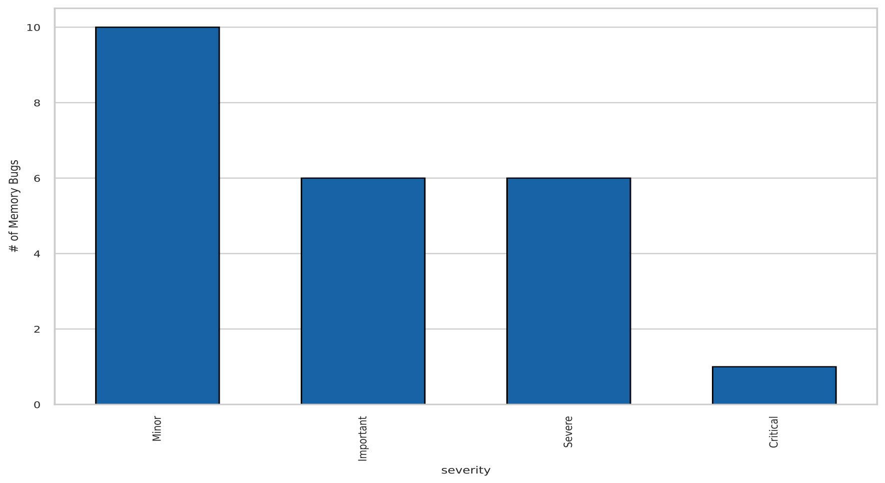


Figure 5.8.: Number of memory bugs from MySQL per severity

SQLite



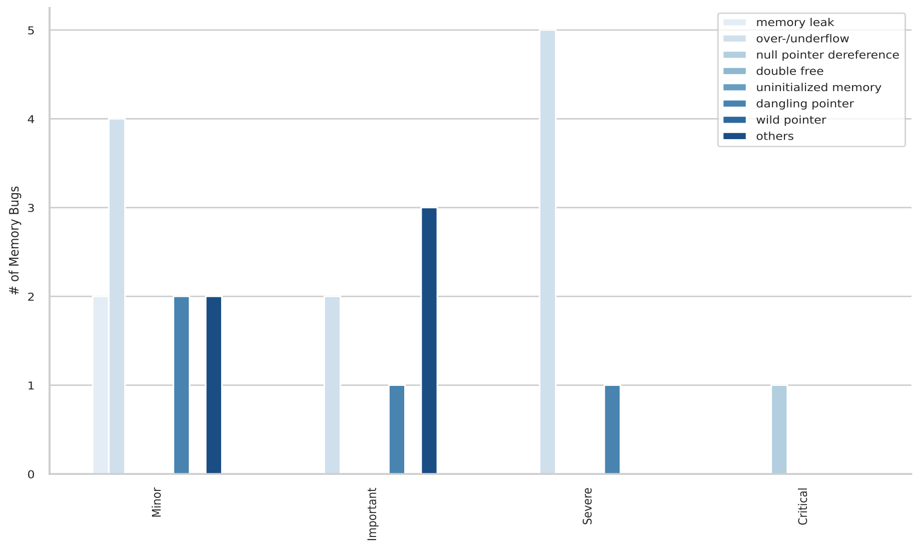


Figure 5.9.: Number of memory bugs from SQLite per severity

5.6. RQ5: Memory bug priority

We analyze the urgency of memory bugs in databases by inspecting the distribution of memory bugs based on priority. SQLite defines 4 levels of priority: immediate, high, medium, and low. MariaDB defines blocker, critical, major, minor, trivial 5 levels of priority. Figures 5.10 and 5.11 shows that most of the memory bugs from SQLite and MariaDB are of high priority. And Over-/underflow and dangling pointer are the top 2 memory bugs in high priority.

SQLite

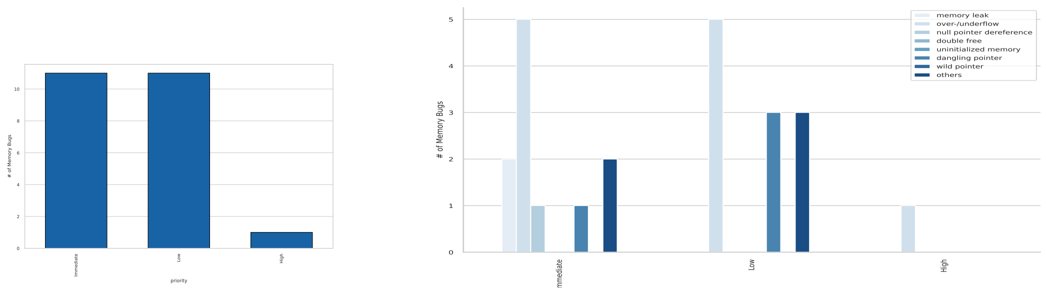


Figure 5.10.: Number of memory bugs from SQLite per priority

MariaDB

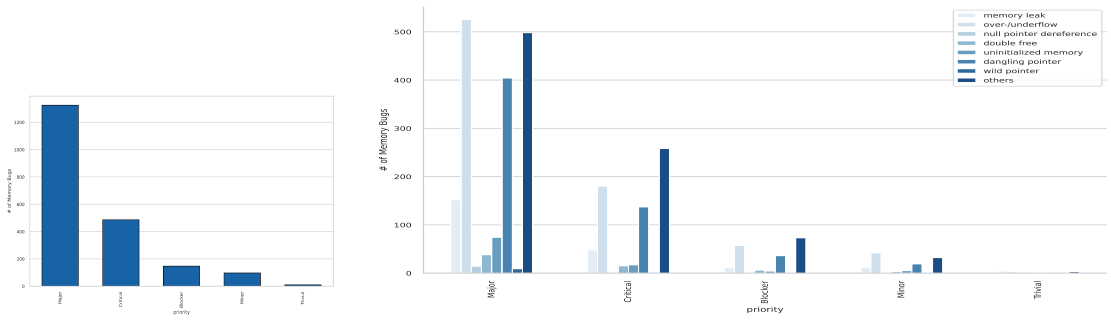


Figure 5.11.: Number of memory bugs from MariaDB per priority

5.7. RQ6: Memory bug locations

After analysis of the component/subsystem/project field of bug reports from MySQL /SQLite/MariaDB, we find that in SQLite many reported bugs reside in non-core components like shell, extensions, and utilities, and most locations are unknown from the report. But in MySQL and MariaDB, the majority of memory bugs are found in database server.

SQLite

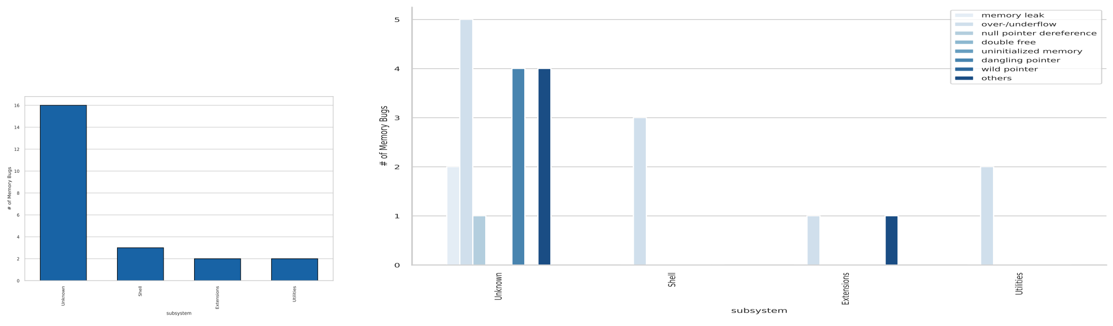
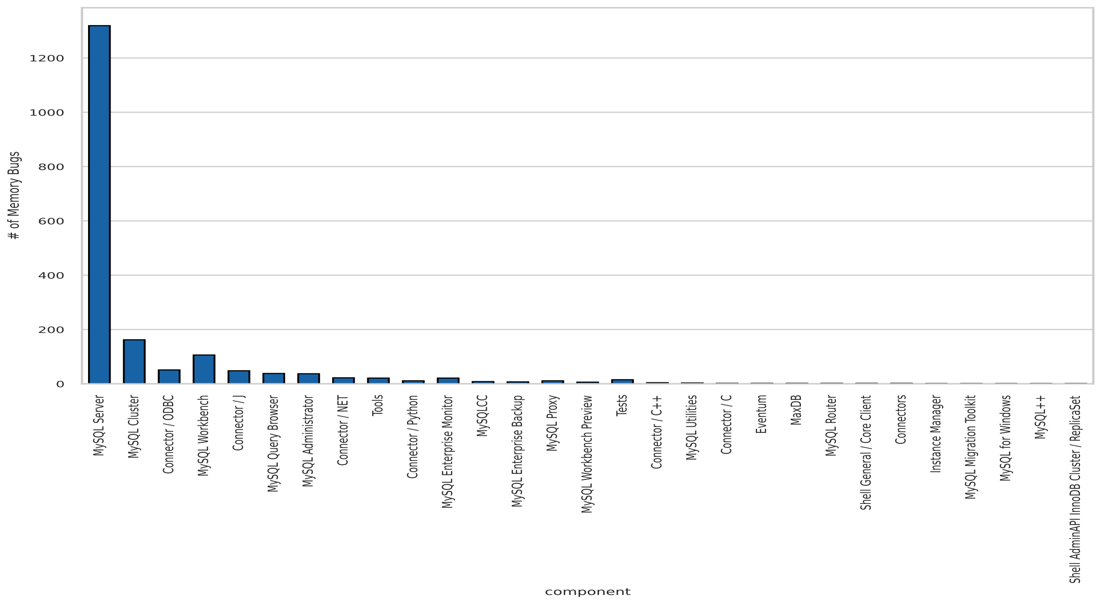


Figure 5.12.: Number of memory bugs from SQLite in subsystems

MySQL



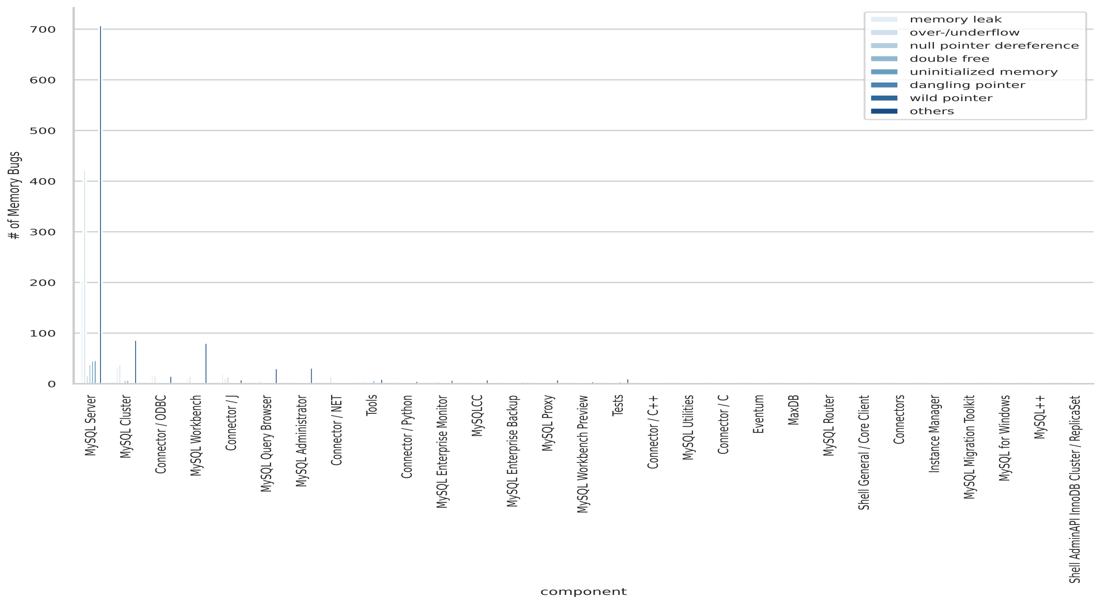
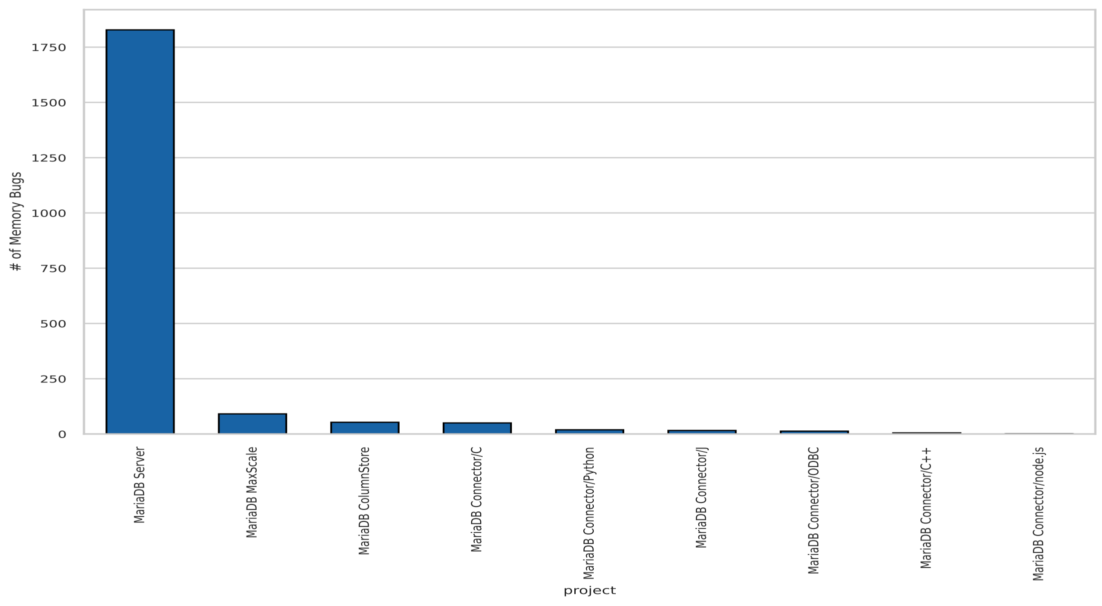


Figure 5.13.: Number of memory bugs from MySQL in components

MariaDB



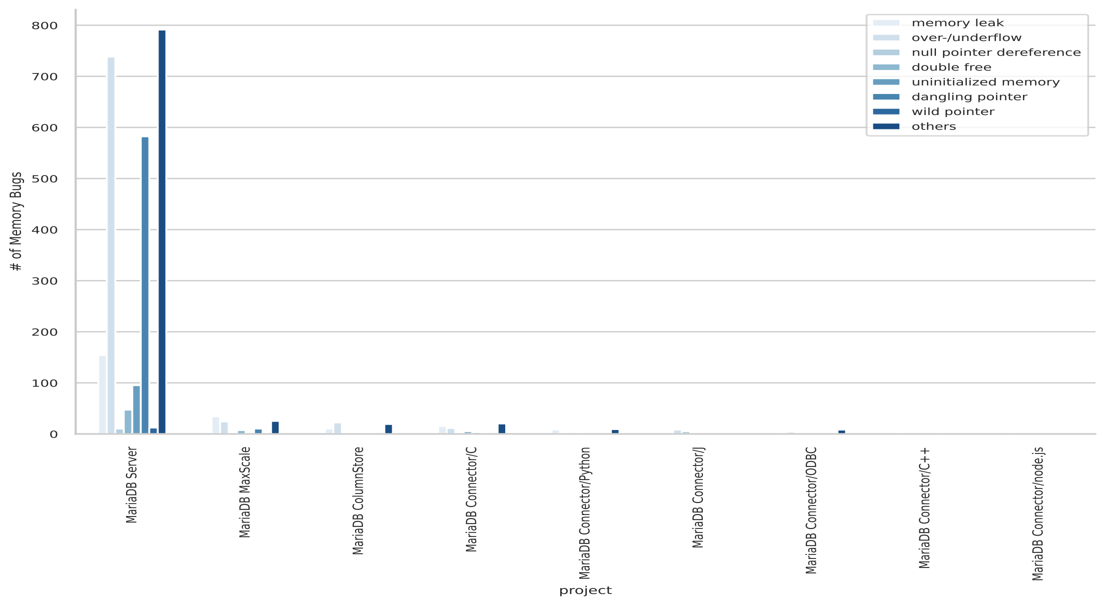


Figure 5.14.: Number of memory bugs from MariaDB in projects

5.8. RQ7: Memory-safety problem caused security exploitation

We hardly find memory bugs tracked as CVEs for MySQL, LevelDB, DuckDB, and RocksDB. For SQLite, MariaDB, and Redis, there are more CVEs with memory issues than in the collection of bug reports on bug tracking systems in terms of proportion (18.2%, 12.3%, and 20.4%, respectively). Only MariaDB has 68% (17 out of 25) memory-safety CVEs reported to the bug tracking system; in other databases, no memory-safety CVEs are found in their bug tracking systems.

Among the 32 serious vulnerabilities published in Redis GitHub repository as of the year of 2024, over 70% are memory safety problems.

SQLite lists 37 recent SQLite CVEs (from the year of 2019) on its documentation website, among which 13 are not a bug of SQLite but misinformation or external bugs. In the remaining 24 CVEs, only 5 are not memory safety problems and memory bugs accouts for nearly 80% in this list. See Table 5.6 for details.

CVE records

Table 5.3.: Number of CVE records per type per database

Database	ML	ODF	NPD	DF	UM	DP	Others	All Memory Bugs	Total
MySQL	0	0	0	0	0	1	0	1	1022
SQLite	0	24	1	0	2	1	16	44	241
MariaDB	0	9	0	0	0	0	16	25	203
Redis	1	38	0	0	0	1	1	41	201
LevelDB	0	0	0	0	0	0	0	0	1
DuckDB	0	0	0	0	0	0	0	0	3
RocksDB	0	0	0	0	0	1	0	1	4

Table 5.4.: Number of CVE records reported as a bug per type per database

Database	ML	ODF	NPD	DF	UM	DP	Others	All
MariaDB	0	2	0	0	0	0	15	17

Redis security vulnerabilities

Table 5.5.: Number of security vulnerabilities per type from Redis

Bug type	ML	ODF	NPD	DF	UM	DP	Others	All Memory Bugs	Total
Count	0	22	1	0	0	0	0	23	32

Recent SQLite CVEs

Table 5.6.: Number of security vulnerabilities per type from SQLite

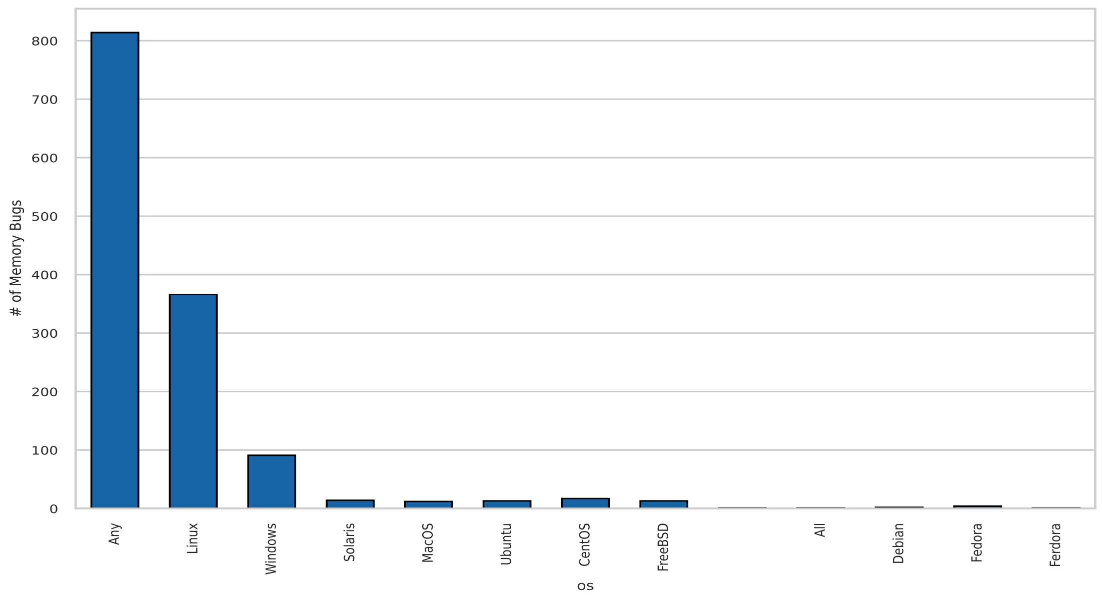
Bug type	ML	ODF	NPD	DF	UM	DP	Others	All Memory Bugs	Total
Count	0	3	6	0	5	5	0	19	24

5.9. Additional analysis

MySQL bug reports contain information on the operating system and CPU architecture. So, we also analyze the distribution of memory bugs from MySQL based on these two fields. The OS and CPU architecture data in MySQL bug reports are pretty. After getting aggregation results from Elasticsearch, we do some rough processing by removing parentheses and strings inside and keeping only the main part.

MySQL-OS

Memory bugs from MySQL have been reported with various kinds of operating systems. Most of them are Unix/Linux systems; these systems are also reported with many memory bugs, only second to the "Any" category. This may be associated with the fact that Unix/Linus systems are the most popular operating system for servers where MySQL database is usually deployed.



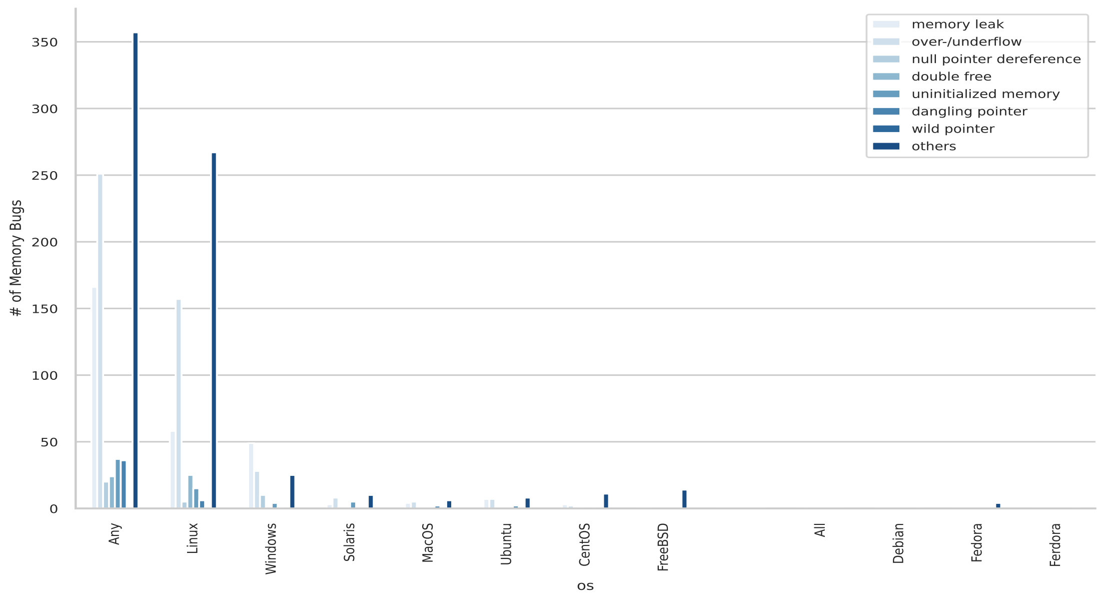


Figure 5.15.: Number of memory bugs from MySQL in operating systems

MySQL-CPU architecture

The CPU architectures reported with memory bugs in MySQL, after processing, has only 3 unique values: any, x86, and ARM. Most memory bugs fall under the "Any" category. The lack of data on x86 and ARM makes it hard to reason about the relationship between memory bugs and CPU architecture.

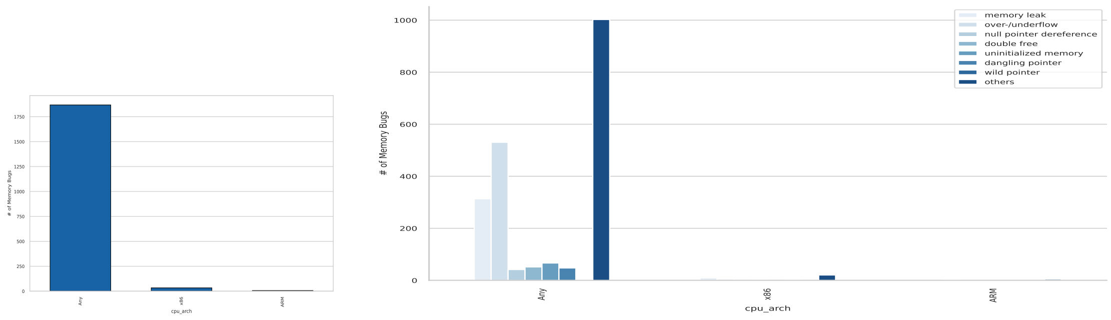


Figure 5.16.: Number of memory bugs from MySQL per CPU architecture

6. Related work

6.1. Empirical study

Researchers from various fields apply empirical studies to investigate the phenomena through data collection from real-world contexts. In the field of bug fixes, Zhong et al.[41] address critical gaps in understanding automatic program repair by analyzing over 9,000 bug fixes from six Java projects (e.g., Apache Cassandra, Lucene). Their work proves the limitations of automatic program repair quantitatively and provides actionable insights for improvement. They develop an automated tool to identify and classify bug fixes based on the issue number and keyword two criteria. Although we and Zhong et al. study different problems, our analysis adopts the same technique and uses keywords to identify bugs.

6.2. Empirical study of bug

Many efforts have been made to study bug characteristics of large software systems. Li et al. [17] investigate the evolving nature of software bugs in modern open-source software by analyzing fixed runtime bugs from Mozilla and Apache HTTP Server to determine whether previous findings on software errors are still relevant given the progress made in bug detection tools, increased use of multi-threading, security concerns, and the open-source development paradigm. The study manually examined 362 bugs and used machine learning techniques to automatically classify 29,000 bug reports from Bugzilla. The analysis shows that memory bugs have decreased, and semantic bugs are dominant. In our study, we use a categorization of memory bugs similar to theirs and perform study on memory bugs from database systems instead of general bugs from general software. While their study shows that memory bugs are decreasing, what we find from our analysis is that memory bugs in database systems are not diminishing

Vahabzadeh et al.[37] study the prevalence, impact, and main cause of bugs in test code across 211 Apache Software Foundation projects by mining and manually analyzing 5,556 test-related bug reports. Thung et al.[35] study bugs in machine learning systems and answer questions about bug causes, frequencies, and potential impacts. Their work is similar to ours in terms of analysis method, but we study memory bugs instead of general bugs and focus on database systems instead of other software and systems.

Zhang et al.[40] study bugs and fixes by analyzing 1100 bug-fix commits across six major Rust projects (including a compiler, database, and operating system) and the Rust standard library, which expands the understanding of software quality in Rust beyond memory safety concerns. Their analysis finds that memory safety bugs are relatively rare but are concentrated in low-level Rust projects such as databases and operating systems. Compared to our focused study of memory bugs in databases, their analysis spreads across various bugs and projects, cares more about bug symptoms and their relationship with unsafe code, and only gets a high-level understanding of memory bugs in databases.

6.3. Empirical study of memory safety

Xu et al.[38] investigate whether Rust fulfills its memory safety promises by analyzing all Rust-related CVEs with memory safety issues. They manually examine 186 Rust memory-safety bugs, categorizing them and evaluating their severity, causes, and potential mitigations.Their work confirms that Rust Safe code does prevent memory-safety bugs and shows that use-after-free is the most common issue, followed by buffer overflows and double-free errors. We and Xu et al. study topics that are very close and use similar methods. Both studies CVEs with memory safety issues, but our study covers a broader range, including the complete bug databases and, therefore, many more bugs and leads to more representative results. They only do a manual examination of the bugs. However, our study mainly uses automated tools.

7. Conclusion

This thesis presents a comprehensive empirical study of memory bugs in seven widely used databases—MySQL, SQLite, MariaDB, Redis, LevelDB, DuckDB, and RocksDB—spanning 150k+ bug reports and 20+ years of development history. By combining automated data collection, keyword-based classification, and multi-dimensional analysis, we uncover critical insights into the nature, trends, and impacts of memory safety bugs, bridging a significant gap in academic research and industry practices. Our analysis shows that Overflow/underflow errors, memory leaks, and dangling pointers constitute the majority of memory bugs and confirms the high proportion (70-80% memory-related security vulnerabilities in Redis and SQLite. Contrary to prior studies of general software, memory bugs in databases like MariaDB and Redis show no decline over time. Most of the memory bugs in MySQL, SQLite, and MariaDB are labeled as middle to high severity and priority. This work demonstrates that memory safety is not a solved problem in database systems. Despite advancements in bug detection, the prevalence of memory-related vulnerabilities underscores the need for paradigm shifts in database engineering—from adopting memory-safe languages to rethinking testing frameworks. The full implementation is available at GitHub repository

8. Future work

For future work, we plan to collect more memory safety keywords and find a better automatic way to identify irrelevant bug reports. We also plan to analyze more database systems like PostgreSQL and their memory bugs. We also plan to investigate the consequences of memory bugs, such as program crashes, data corruption, performance degradation, incorrect functionality, etc., to expand our understanding of memory bugs' impact on database systems. We would also like to investigate how factors like project popularity, size, and complexity of projects affect bug counts.

A. Tables of Numbers of Memory Bugs

This appendix provides tables containing the exact numbers of memory bugs that are shown in plots in Chapter 5.

A.1. MySQL

Table A.1.: Number of memory bugs from MySQL per year

Year	ML	ODF	NPD	DF	UM	DP	Others
2003	6	18	1	0	0	0	42
2004	13	37	3	1	1	1	69
2005	18	54	0	5	3	0	138
2006	27	45	4	9	12	1	120
2007	28	56	1	9	11	1	125
2008	33	48	5	3	10	7	92
2009	31	47	3	6	13	2	100
2010	34	40	6	4	4	4	87
2011	9	10	0	2	1	1	34
2012	6	12	0	1	0	0	21
2013	11	14	0	2	1	1	21
2014	10	17	4	1	0	1	23
2015	18	21	4	3	0	3	34
2016	15	23	0	1	4	5	22
2017	8	16	1	1	3	4	22
2018	16	21	0	1	1	5	32
2019	9	11	0	2	0	2	9
2020	4	10	0	1	0	2	10
2021	8	11	0	0	1	1	8
2022	9	9	7	0	1	0	5
2023	5	6	0	0	0	4	7
2024	4	10	2	0	1	4	5

Table A.2.: Number of memory bugs from MySQL per severity

Severity	ML	ODF	NPD	DF	UM	DP	Others
S3 (Non-critical)	154	254	20	17	32	23	365
S2 (Serious)	95	144	15	17	14	13	314
S1 (Critical)	57	125	5	16	16	8	319
S7 (Test Cases)	6	5	1	1	5	2	14
S4 (Feature request)	4	6	0	0	0	1	3
S5 (Performance)	5	2	0	0	0	1	5
S6 (Debug Builds)	1	0	0	1	0	1	6

Table A.3.: Number of memory bugs from MySQL in operating systems

OS	ML	ODF	NPD	DF	UM	DP	Others
Any	166	251	20	24	37	36	357
Linux	58	157	5	25	16	6	267
Windows	49	28	10	0	4	1	25
Solaris	3	8	0	0	5	0	10
MacOS	4	5	1	1	1	2	6
Ubuntu	7	7	0	0	0	2	8
CentOS	3	2	0	1	0	1	11
FreeBSD	1	1	0	0	0	0	14
	0	0	0	0	0	0	1
All	0	1	0	0	0	0	0
Debian	1	0	0	0	0	0	1
Fedora	0	0	0	0	0	0	4
Ferdora	0	0	0	0	0	0	1

Table A.4.: Number of memory bugs from MySQL per CPU architecture

CPU arch	ML	ODF	NPD	DF	UM	DP	Others
Any	313	530	41	51	67	47	1002
x86	9	4	0	1	0	2	20
ARM	0	2	0	0	0	0	4

Table A.5.: Number of memory bugs from MySQL in components

Component	ML	ODF	NPD	DF	UM	DP	Others
MySQL Server	204	421	16	38	46	46	707
MySQL Cluster	33	38	1	7	7	1	86
Connector / ODBC	17	16	1	1	3	2	15
MySQL Workbench	10	15	1	2	0	0	80
Connector / J	19	10	14	0	0	0	8
MySQL Query Browser	2	6	1	1	0	0	30
MySQL Administrator	3	3	1	0	0	0	31
Connector / NET	5	14	1	0	0	0	2
Tools	6	1	1	0	6	0	9
Connector / Python	5	1	0	0	0	0	5
MySQL Enterprise Monitor	6	4	4	0	0	0	7
MySQLCC	0	0	0	0	0	0	8
MySQL Enterprise Backup	0	3	0	1	0	0	3
MySQL Proxy	3	0	0	0	0	0	8
MySQL Workbench Preview	0	1	0	1	0	0	4
Tests	3	0	0	0	4	0	10
Connector / C++	1	0	0	1	0	0	2
MySQL Utilities	2	0	0	0	0	0	1
Connector / C	1	0	0	0	0	0	1
Eventum	0	0	0	0	0	0	2
MaxDB	0	1	0	0	0	0	1
MySQL Router	1	0	0	0	0	0	1
Shell General / Core Client	0	0	0	0	0	0	2
Connectors	0	0	0	0	1	0	1
Instance Manager	0	0	0	0	0	0	1
MySQL Migration Toolkit	0	1	0	0	0	0	0
MySQL for Windows	0	1	0	0	0	0	0
MySQL++	1	0	0	0	0	0	0
Shell AdminAPI InnoDB Cluster	0	0	0	0	0	0	1

A.2. SQLite

Table A.6.: Number of memory bugs from SQLite per year

Year	ML	ODF	NPD	DF	UM	DP	Others
2010	2	0	0	0	0	0	0
2011	0	2	0	0	0	0	0
2012	0	0	0	0	0	0	0
2013	0	0	0	0	0	0	1
2014	0	1	0	0	0	0	0
2015	0	0	0	0	0	0	0
2016	0	1	0	0	0	0	0
2017	0	1	0	0	0	1	0
2018	0	0	0	0	0	0	0
2019	0	0	1	0	0	0	1
2020	0	2	0	0	0	3	3
2021	0	0	0	0	0	0	0
2022	0	0	0	0	0	0	0
2023	0	4	0	0	0	0	0

Table A.7.: Number of memory bugs from SQLite by severity

Severity	ML	ODF	NPD	DF	UM	DP	Others
Minor	2	4	0	0	0	2	2
Important	0	2	0	0	0	1	3
Severe	0	5	0	0	0	1	0
Critical	0	0	1	0	0	0	0

Table A.8.: Number of memory bugs from SQLite by priority

Priority	ML	ODF	NPD	DF	UM	DP	Others
Immediate	2	5	1	0	0	1	2
Low	0	5	0	0	0	3	3
High	0	1	0	0	0	0	0

Table A.9.: Number of memory bugs from SQLite in subsystems

Subsystem	ML	ODF	NPD	DF	UM	DP	Others
Unknown	2	5	1	0	0	4	4
Shell	0	3	0	0	0	0	0
Extensions	0	1	0	0	0	0	1
Utilities	0	2	0	0	0	0	0

A.3. MariaDB

Table A.10.: Number of memory bugs from MariaDB per year

Year	ML	ODF	NPD	DF	UM	DP	Others
2009	0	0	0	0	0	0	2
2010	0	8	0	0	0	0	4
2011	0	8	1	0	0	0	4
2012	2	6	1	0	1	0	13
2013	7	13	0	3	1	1	10
2014	7	15	0	2	3	1	19
2015	19	22	1	8	1	1	40
2016	16	24	1	2	1	4	37
2017	16	49	2	6	1	15	51
2018	22	62	2	7	3	78	67
2019	16	93	1	6	14	87	56
2020	30	116	2	7	22	85	140
2021	15	92	1	13	14	85	115
2022	24	121	4	3	15	99	114
2023	33	109	0	2	24	85	117
2024	21	72	2	3	13	55	83

Table A.11.: Number of memory bugs from MariaDB by priority

Proirity	ML	ODF	NPD	DF	UM	DP	Others
Major	152	525	14	38	83	404	498
Critical	48	180	2	15	19	137	258
Blocker	12	57	1	6	4	36	73
Minor	12	42	1	3	6	19	32
Trivial	4	3	0	0	1	0	3

Table A.12.: Number of memory bugs from MariaDB in projects

Project	ML	ODF	NPD	DF	UM	DP	Others
MariaDB Server	154	738	10	47	107	582	791
MariaDB MaxScale	34	24	1	7	2	10	25
MariaDB ColumnStore	10	22	1	2	1	1	19
MariaDB Connector/C	15	11	1	5	3	2	20
MariaDB Connector/Python	8	1	0	1	0	0	9
MariaDB Connector/J	3	8	5	0	0	0	0
MariaDB Connector/ODBC	1	4	0	0	0	0	8
MariaDB Connector/C++	2	2	0	0	0	1	0
MariaDB Connector/node.js	1	0	0	0	0	0	0

A.4. Redis

Table A.13.: Number of memory bugs from Redis per year

Year	ML	ODF	NPD	DF	UM	DP	Others
2010	0	0	0	0	0	0	1
2011	2	1	0	0	0	0	8
2012	8	8	0	0	1	0	32
2013	2	4	1	0	0	0	39
2014	9	6	0	1	0	0	28
2015	7	13	0	1	0	0	21
2016	5	4	0	1	0	1	32
2017	9	13	5	1	0	1	35
2018	15	21	3	2	0	2	33
2019	9	13	0	1	0	1	20
2020	16	22	1	2	0	2	47
2021	29	57	10	8	2	4	61
2022	16	30	4	1	0	6	57
2023	10	28	0	2	0	5	47
2024	6	15	1	2	0	6	19

A.5. LevelDB

Table A.14.: Number of memory bugs from LevelDB per year

Year	ML	ODF	NPD	DF	UM	DP	Others
2014	5	3	0	0	0	0	12
2015	0	0	0	0	0	0	2
2016	0	0	0	0	0	0	1
2017	0	1	0	0	0	2	0
2018	1	2	0	0	0	0	1
2019	1	1	0	0	0	0	2
2020	0	0	0	0	0	0	0
2021	0	0	0	0	0	0	0
2022	0	3	0	0	0	0	0
2023	2	0	0	0	0	0	0
2024	2	1	1	0	0	0	0

A.6. DuckDB

Table A.15.: Number of memory bugs from DuckDB per year

Year	ML	ODF	NPD	DF	UM	DP	Others
2024	1	3	0	1	0	1	10

A.7. RocksDB

Table A.16.: Number of memory bugs from RocksDB per year

Year	ML	ODF	NPD	DF	UM	DP	Others
2013	2	0	0	0	0	0	0
2014	2	2	0	1	0	2	11
2015	3	10	0	0	0	1	21
2016	4	4	0	1	0	1	18
2017	3	12	0	4	1	8	19
2018	14	13	0	0	0	7	19
2019	10	13	0	2	0	2	23
2020	13	10	0	0	1	5	19
2021	8	8	0	0	0	4	24
2022	15	6	1	2	1	11	27
2023	4	4	0	1	0	3	17
2024	3	3	0	1	0	13	11

Abbreviations

CVE	Common Vulnerabilities and Exposures
CNA	CVE Numbering Authority
NVD	National Vulnerability Database
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
PAT	Personal Access Token
REST	Representational State Transfer
API	Application Programming Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
CSS	Cascading Style Sheets
OLAP	Online Analytical Processing
ML	Memory Leak
ODF	Over-/underflow
NPD	Null pointer dereference

DF	Double Free
UM	Uninitialized Memory
DP	Dangling pointer

List of Figures

List of Tables

Bibliography

[1] *A proactive approach to more secure code | MSRC Blog | Microsoft Security Response Center*. URL: `https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/` (visited on 02/07/2025).

[2] *About repository security advisories - GitHub Docs*. URL: `https://docs.github.com/en/code-security/security-advisories/working-with-repository-security-advisories/about-repository-security-advisories` (visited on 02/07/2025).

[3] Ankit. *ankitpokhrel/jira-cli*. original-date: 2020-11-22T08:48:39Z. Feb. 7, 2025.

[4] *Blog - Towards the next generation of XNU memory safety: kalloc_type - Apple Security Research*. URL: `https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/` (visited on 02/03/2025).

[5] *CVE Website*. URL: `https://www.cve.org/`.

[6] *CWE - Common Weakness Enumeration*. URL: `https://cwe.mitre.org/index.html` (visited on 02/07/2025).

[7] *DuckDB – An in-process SQL OLAP database management system*. URL: `https://duckdb.org/` (visited on 02/06/2025).

[8] *elastic/elasticsearch-py*. original-date: 2013-05-01T22:47:39Z. Feb. 4, 2025.

[9] FIRST. *Common Vulnerability Scoring System v4.0: Specification Document*. Tech. rep. Accessed: 2025-02-03. Forum of Incident Response and Security Teams (FIRST), 2023.

[10] *Fossil: A Coherent Software Configuration Management System*. URL: `https://fossil-scm.org/home/doc/trunk/www/index.wiki`.

[11] *GitHub - google/leveldb: LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values.* URL: `https://github.com/google/leveldb` (visited on 02/06/2025).

[12] *GitHub Issues · Project planning for developers · GitHub*. URL: `https://github.com/features/issues` (visited on 02/07/2025).

[13] *https://www.elastic.co/elasticsearch*. URL: `https://www.elastic.co/elasticsearch` (visited on 02/07/2025).

[14] *https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html.* URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html (visited on 02/07/2025).

[15] *Jira | Issue & Project Tracking Software | Atlassian.* URL: https://www.atlassian.com/software/jira (visited on 02/07/2025).

[16] O. Lauzanne. *pyquery: A jQuery-like library for Python.* https://pyquery.readthedocs.io/. 2022.

[17] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. "Have things changed now? An empirical study of bug characteristics in modern open source software." In: *Proceedings of the 1st workshop on Architectural and system support for improving software dependability.* 2006, pp. 25–33.

[18] *Llama.* URL: https://www.llama.com/ (visited on 02/06/2025).

[19] *MariaDB Foundation - MariaDB.org.* URL: https://mariadb.org/ (visited on 02/06/2025).

[20] *Memory safety.* URL: https://www.chromium.org/Home/chromium-security/memory-safety/ (visited on 02/07/2025).

[21] *MySQL.* URL: https://www.mysql.com/ (visited on 02/06/2025).

[22] *MySQL Bugs.* URL: https://bugs.mysql.com/ (visited on 02/07/2025).

[23] *NVD - Home.* URL: https://nvd.nist.gov/.

[24] *PyGithub/PyGithub.* original-date: 2012-02-25T12:53:47Z. Feb. 6, 2025.

[25] *Rate limits for the REST API - GitHub Docs.* URL: https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api?apiVersion=2022-11-28 (visited on 02/07/2025).

[26] A. Rebert and C. Kern. *Secure by Design: Google's Perspective on Memory Safety.* Tech. rep. Google Security Engineering, 2024.

[27] *Redis - The Real-time Data Platform.* URL: https://redis.io/ (visited on 02/06/2025).

[28] K. Reitz. *Requests: HTTP for Humans.* https://docs.python-requests.org/. 2011.

[29] L. Richardson. *Beautiful Soup: A library for parsing HTML and XML.* https://beautiful-soup-4.readthedocs.io/. 2004.

[30] *RocksDB | A persistent key-value store | RocksDB.* URL: https://rocksdb.org/ (visited on 02/06/2025).

[31] *Scrapy: A Fast and Powerful Scraping and Web Crawling Framework.* https://scrapy.org/. Accessed: [2025-02-07].

[32] Selenium Project. *Selenium WebDriver*. `https://www.selenium.dev/`. Accessed: [2025-02-07].

[33] *SQLite Home Page*. URL: `https://www.sqlite.org/` (visited on 02/06/2025).

[34] *SQLite: Ticket Main Menu*. URL: `https://sqlite.org/src/reportlist`.

[35] F. Thung, S. Wang, D. Lo, and L. Jiang. "An empirical study of bugs in machine learning systems." In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE. 2012, pp. 271–280.

[36] C. to urllib3. *urllib3: A HTTP client for Python*. `https://urllib3.readthedocs.io/`. 2025.

[37] A. Vahabzadeh, A. M. Fard, and A. Mesbah. "An empirical study of bugs in test code." In: *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. 2015, pp. 101–110.

[38] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu. "Memory-safety challenge considered solved? An in-depth study with all Rust CVEs." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.1 (2021), pp. 1–25.

[39] J. Zeng. *jayzeng/scrapy-elasticsearch*. original-date: 2014-08-13T20:08:01Z. Nov. 3, 2024.

[40] C. Zhang, Y. Feng, Y. Zhang, Y. Dai, and B. Xu. "Beyond Memory Safety: an Empirical Study on Bugs and Fixes of Rust Programs." In: *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2024, pp. 272–283.

[41] H. Zhong and Z. Su. "An empirical study on real bug fixes." In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 913–923.

[42] C. Zhu. *pyspider: A Powerful Spider (Web Crawler) System in Python*. `https://docs.pyspider.org/`. Accessed: [2025-02-07]. 2014.