Saarland University

Department of Computer Science

Impact of Knowledge-Sharing Platforms on Software Security and Academic Research

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von

Alfusainey Jallow

Saarbrücken, 2025

Tag des Kolloquiums:	24. Oktober 2025
Dekan:	Prof. Dr. Roland Speicher

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Dietrich Klakow
Berichterstattende:	Dr. Sven Bugiel
	Prof. Dr. Andreas Zeller
Akademischer Mitarbeiter:	Dr. Alexi Turcotte

Zusammenfassung

Stack Overflow ist eine beliebte Plattform bei Softwareentwicklern und -forschern. Entwickler nutzen sie häufig als Quelle für funktionale, kopierfertige Code-Snippets, während Forscher ihre Inhalte analysieren, um Trends, Verhaltensweisen und die Auswirkungen von Code-Wiederverwendung zu analysieren – insbesondere im Hinblick auf Sicherheit. Der Code auf Stack Overflow ist jedoch nicht statisch; die Community überarbeitet Code-Snippets und behebt dabei manchmal Fehler und Schwachstellen, ähnlich wie bei Code in versionskontrollierten Repositories. Diese fortlaufende Weiterentwicklung von Code-Snippets wirft wichtige Fragen zu ihren Auswirkungen auf die Softwaresicherheit und die Forschungsmethodik auf.

In dieser Dissertation untersuchen wir die Weiterentwicklung der Stack Overflow-Code-Snippets und ihre Auswirkungen auf die Softwaresicherheit und Forschungsergebnisse. Entwickler verwenden Snippets häufig wieder, ohne deren Updates zu verfolgen, was zu veraltetem und potenziell anfälligem Code in Open-Source-Projekten führt. Unsere Analyse von über 11.000 GitHub-Projekten fand Tausende solcher veralteten Snippets, darunter auch verpasste kritische Sicherheitsfixes.

Dies legt die Notwendigkeit nahe, Entwickler mit Tools zu unterstützen, die Stack Overflow kontinuierlich auf Sicherheitswarnungen oder Code-Fixes überwachen. Darüber hinaus beeinflusst die Weiterentwicklung der Stack Overflow-Code-Snippets und ihres Kontextes die Reproduzierbarkeit von Forschungsergebnissen. Sechs von uns replizierte Studien lieferten auf einer neueren Datensatzversion deutlich unterschiedliche Ergebnisse. Deshalb empfehlen wir Forschern, Stack Overflow-Daten als Zeitreihendaten zu betrachten, um einen besseren Kontext für die Interpretation der Ergebnisse zu schaffen.

Abstract

Stack Overflow is a widely used platform among software developers and researchers. Developers frequently rely on it as a source of functional, copy-ready code snippets, while researchers study its content to analyze trends, behaviors, and the implications of code reuse—particularly concerning security. However, code on Stack Overflow is not static; the community revises posted code snippets, sometimes addressing bugs and vulnerabilities, much like code in version-controlled repositories. This ongoing evolution raises important questions about its effect on software security and research methodology.

In this dissertation, we study the evolving nature of Stack Overflow code snippets and its impact on software security and research results. Developers often reuse snippets without tracking updates, leading to outdated and potentially vulnerable code in open-source projects. Our analysis of over 11,000 GitHub projects revealed thousands of such outdated snippets, including missed critical security fixes. This suggests the need to support developers with tools to help constantly monitor Stack Overflow for security warnings or code fixes. Additionally, the evolving nature of Stack Overflow code snippets and its surrounding context impacts the replicability of cross-sectional research findings, with six studies that we replicated yielding significantly different results on a newer dataset version. Accordingly, we recommend that researchers treat Stack Overflow data as a time series data source to provide better context for interpreting cross-sectional research findings.

Background of this Dissertation

This dissertation is grounded in the research and findings presented in the following two key papers, both of which I contributed as the primary author.

The idea and motivation for studying bugs in code snippets on Stack Overflow and their propagation to open-source projects on GitHub first emerged when the author of this dissertation attempted to participate in the *MSR 2019 Mining Challenge*. Building upon this initial concept, the author, in collaboration with his supervisor Dr. Sven Bugiel, further refined the focus to explore the evolution of code snippets on Stack Overflow and its potential impact on both software developers [P1] and security researchers [P2]. This refined focus ultimately led to the development of this dissertation.

The author is the primary contributor to [P1], being responsible for implementing and running the experiments as well as writing the majority of the paper. Michael Schilling actively participated in meetings, offering valuable insights and advice. He suggested expanding the project's scope to include more open-source projects and exploring cross-product clone detection to examine whether developers update code snippets copied from Stack Overflow. Additionally, Michael Schilling contributed to rewriting parts of the methodology section. Sven Bugiel contributed to the writing process and conducted some of the statistical analyses discussed in the results section. Both Sven Bugiel and Michael Schilling offered valuable insights during the implementation of the experiments and provided helpful suggestions for debugging the code when the author faced challenges. The author is the primary contributor to [P2], responsible for replicating all six research studies and writing the majority of the paper. Sven Bugiel contributed to both the writing and aspects of the statistical analysis.

[P1] Jallow, A., Schilling, M., Backes, M., and Bugiel, S. Measuring the effects of stack overflow code snippet evolution on open-source software security. In: *45th IEEE Symposium on Security and Privacy (SP'24)*. IEEE, 2024.

[P2] Jallow, A. and Bugiel, S. Stack overflow meets replication: security research amid evolving code snippets. In: *34th USENIX Security Symposium (USENIX Sec'25)*. USENIX Association, 2025.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my exceptional supervisor, Dr. Sven Bugiel, for his unwavering support, guidance, and encouragement throughout my studies at CISPA. His mentorship has been instrumental in shaping my academic journey and research vision. I am also sincerely thankful to my former supervisor, Prof. Michael Backes, who provided me with the funding and a calm, supportive environment to begin my research. His early guidance laid a strong foundation for my work, and I remain grateful for the opportunities he afforded me. I extend heartfelt thanks to Michael Schilling for our research collaboration on the IEEE Oakland paper. His insightful comments and valuable contributions greatly enhanced the quality and impact of the work.

I would also like to acknowledge Ahmad Salem for his friendship, stimulating conversations, and the many insightful discussions we have shared over the years. Sincere thanks to all my colleagues both in the Trusted Systems and Information Security research groups. I also wish to thank the staff of the IT Service Department at CISPA for their support, especially in setting up databases and resolving various server-related issues I encountered during my research.

I owe a special debt of gratitude to my wife and daughter. Their patience, love, and steadfast support carried me through the most challenging phases of this journey. My deepest appreciation also goes to my mother-in-law, Ines Mairhofer, whose incredible help in caring for our daughter made it possible for me to focus on my studies. As our daughter was born during the COVID pandemic and I transitioned to working from home, Ines's dedication and generosity were indispensable in balancing family and academic commitments. I am equally grateful to my father-in-law for his continuous support.

Contents

List of Figures

List of Tables

1 Introduction

The work of software developers is strenuous. To ease their job, developers seek assistance in different forms, such as better IDE support, reusable third-party code, accessible guidelines and best practices, or exchange with other developers. Studies have shown that, historically, developers frequently turned to web search for guidance when dealing with security-related APIs or libraries [2, 30, 31]. One internet information source in particular has become popular: Stack Overflow, the largest question and answer site for programmers to share and increase their knowledge. Stack Overflow is so popular among developers because most answers provide a short, concise explanation together with an example in the form of a code snippet about how a particular technology is supposed to work. For instance, a developer learning a new API will appreciate having an example code snippet that demonstrates how the API works in practice. Given the time constraints and economic pressure that developers face, having ready-to-use and good functional code snippets is a welcome time saver. As many recent studies show, this sentiment is common among developers and contributes heavily to the platform's unabated popularity [15, 75, 116, 29, 30, 31]

The high availability of functional code snippets that can be readily reused (copied) by developers has recently attracted the attention of security researchers interested in studying the impact that Stack Overflow has on software security [116, 31, 1, 119] and in designing better tooling support to help developers write more secure code [110, 21, 2, 36]. Prior *cross-sectional* research studies conducted developer studies by extracting code snippets from a specific version of the official Stack Overflow data dump [94]. The Stack Overflow data dump is published on a quarterly basis and includes all the content that has been created on the platform from its inception in 2008 up until the date the dump is released. Similarly, when developers copy a code snippet from Stack Overflow, they are incorporating a specific version, usually the most recent, creating a dependency in their project much like using a third-party library.

However, Stack Overflow is constantly evolving: new content is created all the time, while existing content is continuously edited and improved upon [15], sometimes addressing bugs and vulnerabilities. Conversely, this means that the content of one Stack Overflow dataset version may be widely different from another version. For developers, this means that when they copy a code snippet, they might miss important updates when the copied snippet is edited on Stack Overflow in the future. Effectively, this means that code on Stack Overflow needs to be treated as any other code in regular version control systems. Similarly, it is unclear how this evolution affects cross-sectional research findings based on particular dataset versions and what lessons can be learned for future studies using Stack Overflow data.

This observation sets the foundation for this dissertation, which offers new insights into the effects of Stack Overflow code snippet evolution and its surrounding context on software security [P1] and academic research [P2].

Effect on Developers (P1). We study the effects of Stack Overflow code snippet evolution on open-source software security by devising a method to 1) detect outdated code snippets versions from 1.5M Stack Overflow snippets in 11,479 popular GitHub projects and 2) detect security-relevant updates to those Stack Overflow code snippets not reflected in those GitHub projects. Through this work, we were able to discover that developers do not update dependent code snippets when those evolved on Stack Overflow. In particular, we found that 2,405 code snippet versions reused in 2,109 GitHub projects were outdated, with 43 projects missing fixes for bugs and vulnerabilities. These results suggest that developers need tools that continuously monitor Stack Overflow for security warnings and code fixes for reused code snippets and not only warn during copy-pasting.

Effect on Researchers (P2). We study the impact of Stack Overflow code evolution on the stability of prior research findings derived from Stack Overflow data and provide recommendations for future studies. We systematically reviewed papers published at security venues between 2005–2023 to identify key aspects of Stack Overflow that can affect study results, such as the language or context of code snippets. Our analysis reveals that certain aspects are non-stationary over time, which could lead to different conclusions if experiments are repeated at different times. We replicated six studies using a more recent dataset version to demonstrate this risk. The findings show that four papers produced significantly different results than the original findings, preventing the same conclusions from being drawn with a newer dataset version. Moving forward, we recommend treating Stack Overflow as a time series data source to provide more accurate context for research conclusions.

Outline

The remainder of this dissertation is structured as follows. In Chapter 2, we present the technical background on Stack Overflow, with a particular focus on post structure and version history. Chapter 3 investigates how the evolution of code snippets on Stack Overflow impacts software developers, while Chapter 4 explores its implications for research findings. Finally, Chapter 5 concludes the dissertation and outlines promising directions for future research.

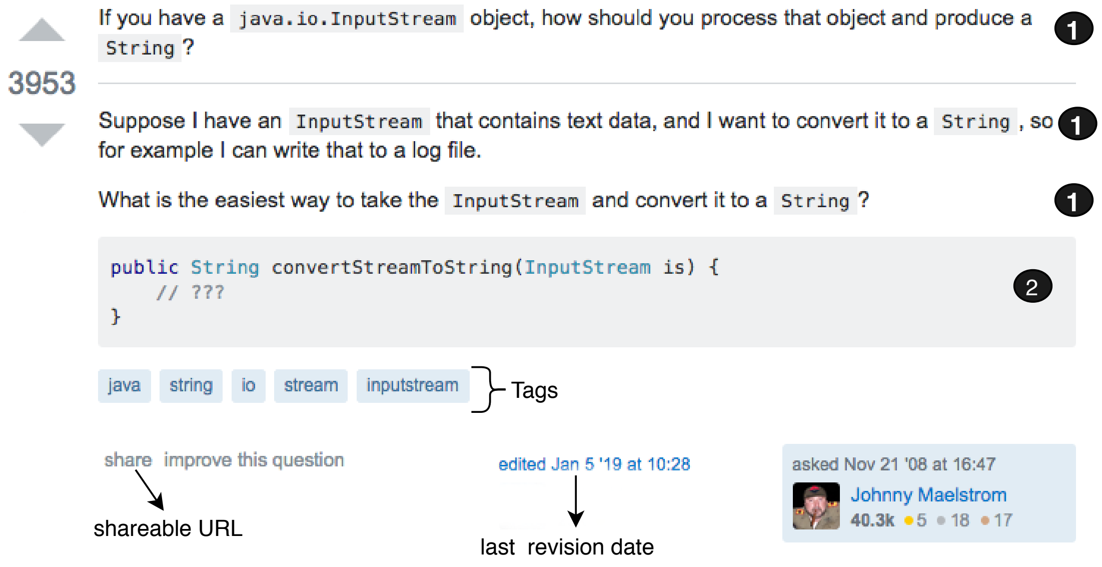
2
Technical Background

2.1 Stack Overflow Primer

Stack Overflow is a widely used platform for knowledge sharing, enabling developers to ask and answer programming-related questions. It is an essential resource for those seeking help with coding challenges and technical problems. The platform's question-and-answer structure, coupled with its large developer community, provides a vast repository of programming knowledge and insights. When developers encounter a programming issue, they can post a question on Stack Overflow, including enough details and, if possible, a minimal working example to help others provide accurate answers. Each question must have a title, a body, and up to five optional tags. When a developer enters a title, the platform suggests similar existing questions to prevent duplication. If the issue has already been addressed, the developer is directed to the relevant post. A new question is only allowed if no similar question exists. Once posted, other developers can respond to the question. As of now, Stack Overflow hosts over 26.7 million registered developers[102]. However, not all developers can answer every question, as their expertise vary.

Tags play a crucial role in connecting questions to developers who have the right expertise to answer them. Thus using tags ensures that the questions reaches those knowledgeable in the topic. Developers who subscribe to specific tags receive notifications whenever new questions with those tags are posted, allowing questioners to get answers from experienced developers within their area of expertise. Additionally, tags help users filter and locate questions of interest, and have been utilized in numerous previous research studies[49, 110, 13, 82, 32, 1, 72, 81] to identify the programming language of code snippets. Figure 2.1 illustrates a question posted by a user seeking to convert an *InputStream* into a Java String, accompanied by a skeleton method implementation. The question post includes three `text` blocks (❶) and one code block (❷). A text block refers to a segment of written content, typically composed of natural language text intended to provide contextual information around code snippets. On the other hand, a code block consists of one or more lines of code formatted using Markdown, a lightweight markup language for styling web content. Together, text blocks and code blocks provides a way for developers to effectively ask and answer questions. The post has five tags, including the java tag, which specifies the programming language of the code snippet. With a score of `3,953`, the question is widely regarded as useful by many developers in the community.

Each question or answer post on Stack Overflow is assigned a unique, shareable URL that serves as a direct reference to the post. Developers who reuse code snippets from Stack Overflow sometimes include this shareable URL in their codebase as a way to credit the original source. This practice aligns with Stack Overflow's licensing requirements, which mandate **attribution** when copying code snippets from the platform. A **discussion thread** comprises the original question along with all its associated answers.



3953

If you have a `java.io.InputStream` object, how should you process that object and produce a `String`?

Suppose I have an `InputStream` that contains text data, and I want to convert it to a `String`, so for example I can write that to a log file.

What is the easiest way to take the `InputStream` and convert it to a `String`?

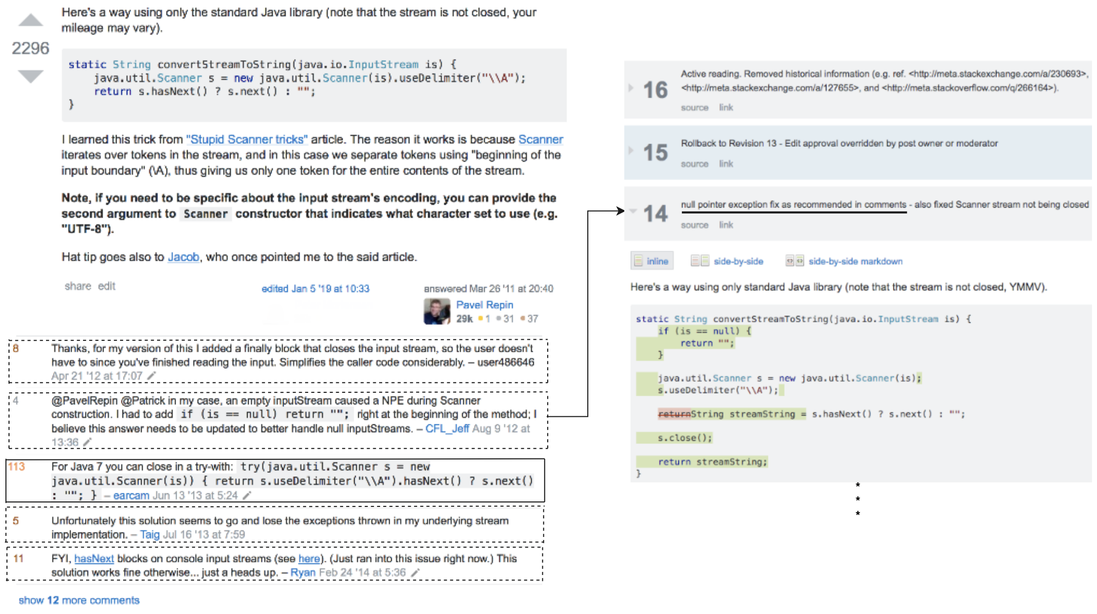
```
public String convertStreamToString(InputStream is) {
    // ???
}
```

Figure 2.1: Question post on Stack Overflow. The question has 3,953 votes (scores) and contains three text blocks (❶) and one code block (❷).

2.2 Version Control of Stack Overflow Artifacts

In general, posts on Stack Overflow are dynamic, not static, content. They are heavily shaped by the ongoing interaction and feedback from the community, such as through comments, upvotes, or downvotes. This community engagement can prompt the original post creator or other developers to edit, refine, or expand the content of the post to better address the needs or suggestions from the community. The feedback loop encourages continuous improvement and ensures that posts evolve over time to maintain relevance and accuracy. For each post created and for every subsequent edit made, Stack Overflow generates a post version object. This object records the specific changes made to the text and code within the post, capturing both the modifications and the associated metadata, including the date and time of the change and the identity of the author who made the edit. If the user chooses to include an optional commit message summarizing the changes, it is stored together with the post version. All post versions are stored in a post change history, which is a comprehensive record of every revision made to a post over time. This version history ensures that all changes are preserved, creating a detailed log of how posts have evolved, making it possible to trace the evolution of content. Despite keeping a record of all previous versions, only the most recent version of a post is displayed on the Stack Overflow website. Users typically interact with the current version of the post, while the post's change history remains behind the scenes, ensuring that earlier versions and edits are accessible if needed.

We briefly illustrate the versioning of Stack Overflow posts using a concrete example. Figure 2.2 depicts the revision history and comments made to an answer post that was created to the question in Figure 2.1. The revision history (on the right) shows all the revisions made to the answer. The comment section of the answer (left-bottom) shows



2296

Here's a way using only the standard Java library (note that the stream is not closed, your mileage may vary).

```java
static String convertStreamToString(java.io.InputStream is) {
    java.util.Scanner s = new java.util.Scanner(is).useDelimiter("\\A");
    return s.hasNext() ? s.next() : "";
}
```

I learned this trick from "Stupid Scanner tricks" article. The reason it works is because Scanner iterates over tokens in the stream, and in this case we separate tokens using "beginning of the input boundary" (\A), thus giving us only one token for the entire contents of the stream.

Note, if you need to be specific about the input stream's encoding, you can provide the second argument to `Scanner` constructor that indicates what character set to use (e.g. "UTF-8").

Hat tip goes also to Jacob, who once pointed me to the said article.

share edit   edited Jan 5 '19 at 10:33   answered Mar 26 '11 at 20:40
Pavel Repin
29k ●1 ●31 ●37

8 Thanks, for my version of this I added a finally block that closes the input stream, so the user doesn't have to since you've finished reading the input. Simplifies the caller code considerably. – user486646 Apr 21 '12 at 17:07

4 @PavelRepin @Patrick in my case, an empty inputStream caused a NPE during Scanner construction. I had to add `if (is == null) return "";` right at the beginning of the method; I believe this answer needs to be updated to better handle null inputStreams. – CFL_Jeff Aug 9 '12 at 13:36

113 For Java 7 you can close in a try-with: `try(java.util.Scanner s = new java.util.Scanner(is)) { return s.useDelimiter("\\A").hasNext() ? s.next() : ""; }` – earcam Jun 13 '13 at 5:24

5 Unfortunately this solution seems to go and lose the exceptions thrown in my underlying stream implementation. – Taig Jul 16 '13 at 7:59

11 FYI, hasNext blocks on console input streams (see here). (Just ran into this issue right now.) This solution works fine otherwise... just a heads up. – Ryan Feb 24 '14 at 5:36

show 12 more comments

16 Active reading. Removed historical information (e.g. ref. <http://meta.stackexchange.com/a/230693>, <http://meta.stackexchange.com/a/127655>, and <http://meta.stackoverflow.com/q/266164>).
source link

15 Rollback to Revision 13 - Edit approval overridden by post owner or moderator
source link

14 null pointer exception fix as recommended in comments - also fixed Scanner stream not being closed
source link

Here's a way using only standard Java library (note that the stream is not closed, YMMV).

```java
static String convertStreamToString(java.io.InputStream is) {
    if (is == null) {
        return "";
    }

    java.util.Scanner s = new java.util.Scanner(is);
    s.useDelimiter("\\A");

    ~~return~~String streamString = s.hasNext() ? s.next() : "";

    s.close();

    return streamString;
}
```

Figure 2.2: A Stack Overflow answer post with comments and revision history. Four of the comments (marked with dashed lines) point to flaws in the code snippet and one comment (solid line) suggests an alternative solution. The 14th revision of the post fixes a bug raised in the 2nd comment.

a number of comments provided by developers. In the second comment (marked with dashed line), a developer raises an issue that the code snippet in the answer is missing a null check, which can cause a `NullPointerException` if the input parameter is `null`. The third comment (shown in solid line) was provided by a different developer and offers an alternative solution to the same problem that works in a different Java runtime environment. A total of 113 developers found the comment that provided the alternative solution useful while 4 developers found the comment raising the null check issue useful. We thus regard the comment in dashed lines as a bug-report and the comment in solid lines as a feature request.[1] If we take a closer look at the 14th revision of the answer, we see that the revision was triggered by the bug-report comment. In this revision, a developer edited the post to include a null pointer check for the input parameter and recorded the changes together with a commit message summarizing the nature of the edit and indicating that the fix was due to an issue raised in the comments.

2.3 Tables and Schema Overview

Having reviewed an example of version control on Stack Overflow, let us now examine how the Stack Overflow database schema is structured to facilitate post versioning. Previous research on Stack Overflow has primarily focused on two types of data sets: the official Stack Exchange dataset [94] and the SOTorrent[54]. SOTorrent is derived

[1]There are three additional bug reports (shown in dashed lines) and an additional twelve comments which are hidden in the comments section

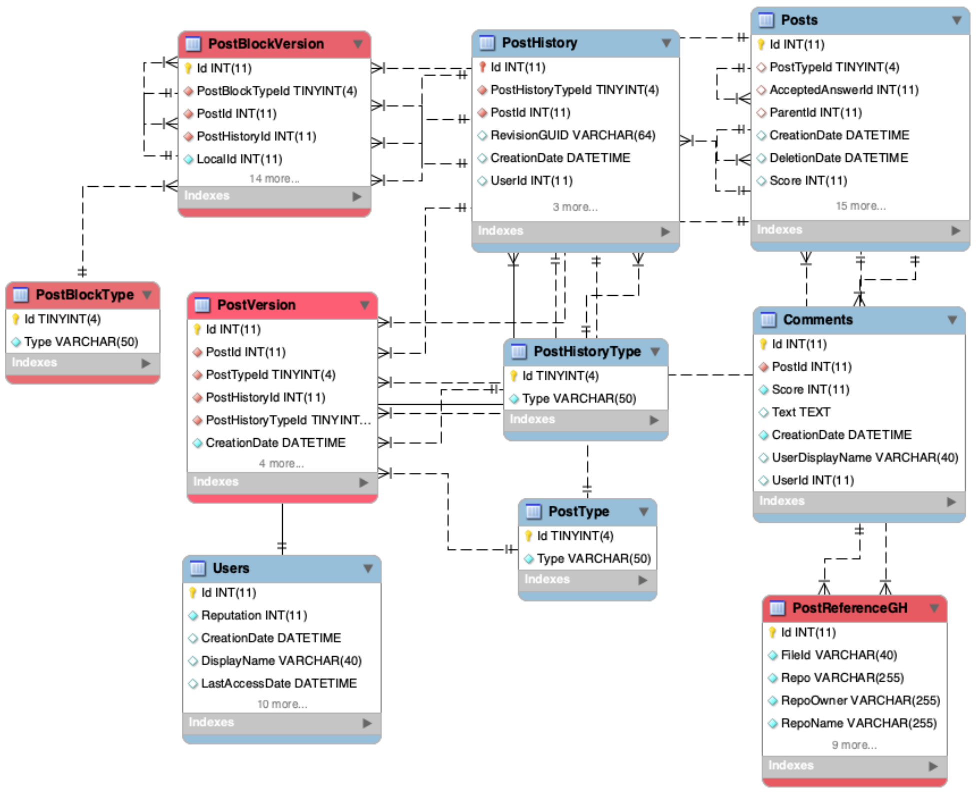


Figure 2.3: Entity-relationship diagram illustrating the relationships between key tables from the Stack Exchange dataset (shown in blue) and the SOTorrent dataset (shown in red).

from the official Stack Exchange dataset but includes additional tables that enable the tracking of the evolution of individual text and code snippets. This section will outline the main tables from both datasets that are utilized in this dissertation. Figure 2.3 illustrates the entity relationship diagram, depicting the structure and connections of the key database tables from the Stack Exchange dataset (shown in blue) and the SOTorrent dataset (shown in red).

2.3.1 Stack Exchange Dataset

Stack Exchange Inc. regularly releases a quarterly dataset containing all content created on Stack Overflow up to the time of the release. The latest available release of the dataset is publicly available on archive.org[2] and consist of a number of database tables.

The `Posts` table is central to the database, containing all question and answer posts. Each entry represents a post, with details such as the post's type (e.g., question, answer), the content (body and title), and associated metadata like creation dates, scores, and view counts. The table also includes references to related posts, such as the `ParentId`, which links answer posts to their corresponding question posts, and

[2]Note: Only the latest release is made available on archive.org, replacing the previous version of the dataset.

the `AcceptedAnswerId`, which points to the answer accepted by the post owner (i.e., questioner). The `Posts` table also tracks the number of answers, comments, and favorites associated with a post. This structure supports Stack Overflow's discussion format, allowing for the efficient organization and retrieval of post-related content. The `PostType` table categorizes posts by their type, such as Question or Answer. This classification allows the system to differentiate between various kinds of posts and apply appropriate rules and behaviors for each type. For example, answers may be linked to a specific question, while questions may have multiple answers associated with them. There are also other post types, such as `Wiki`, `PrivilegeWiki`, etc, and the platform currently supports **eight** different post types[105]. The `Comments` table stores all comments made on posts. Each comment is linked to a specific post through the PostId field, allowing users to provide feedback, clarifications, or suggestions. In addition to the content of the comment, the table captures metadata such as the comment's score, the user who posted it, and the creation date. This table is crucial for enabling interaction within the community, allowing users to communicate directly with the authors of posts. The comments also provide valuable context for code snippets and have been utilized in this dissertation to identify security-related issues and code fixes (see Section 3.2.5).

When posts are edited, these changes are recorded in the `PostHistory` table. This table logs each edit made to a post, categorizing them by the type of change (e.g., title edit, body edit, post deletion, etc). It records who made the change, when it occurred, and what the content of the post was at the time of the edit. This history allows Stack Overflow to track revisions over time at the post level and ensures that edits are transparent and well-documented. It also records an optional commit message summarizing the changes made. The `PostHistoryType` table provides additional context for the `PostHistory` table, categorizing the types of changes that can occur in a post. This helps Stack Overflow maintain a clear distinction between different types of post modifications, such as editing the body or title of a post versus deleting it or marking it as accepted. Stack Overflow currently supports `66` different post history types[104]. The `Users` table contains information about users on the platform, including their reputation, display name, account details, and activity history. Each user is assigned a unique ID, and the table captures details such as when the account was created, the user's profile information, and voting history. This table is essential for managing user profiles, tracking their contributions, and enabling features like upvoting and downvoting. Together, these tables create an interconnected structure that supports Stack Overflow's functionality. The Posts table serves as the core repository for content, while the `Comments`, `PostHistory`, `PostHistoryType`, and `PostType` tables support community interaction, and content evolution. The `Users` table ties all activities back to individual users.

2.3.2 SOTorrent Dataset

The Stack Exchange data set offers version control at the post level, meaning it does not track revisions at the level of individual text or code snippets contained in a post. Given that this dissertation examines how code snippets evolve, a more detailed, fine-

Table 2.1: Overview of some of the Post History Types in the `PostHistoryType` table. SOTorrent employs the first three history types to track modifications made to post content.

Id	PostHistoryType	Description
2	Initial Body	The initial raw body text of the post (in markdown format)
5	Edit Body	The modified post body text (in markdown format).
8	Rollback Body	The reverted raw body text (in markdown format).
1	Initial Title	The initial title of the post (applies to questions only).
3	Initial Tags	The initial list of tags of the post (applies to questions only).
4	Edit Title	The modified title of the question.
10	Post Closed	The post has been voted to be closed.
12	Post Deleted	The post has been voted to be removed.
16	Community Owned	The post is now community owned.
24	Suggested Edit Applied	The suggested edit has been applied.

grained version control is necessary to track the evolution of individual code blocks. The SOTorrent open dataset by Baltes et al. [15] extends the official Stack Exchange dataset by providing the level of granular versioning required to study the evolution of specific code snippets and their changes over time. The data set includes all the tables found in the official Stack Exchange dataset and includes additional tables that enable a more detailed analysis of code and text block evolution. The dataset introduced the concept of text blocks and code blocks to distinguish between human-written text and code snippets and to enable version control at the level of individual text and code elements. These are defined as `TextBlock` and `CodeBlock` types and stored in the `PostBlockType` table.

Recall that the `PostHistory` table in the Stack Exchange dataset tracks modifications to both the content of posts and their associated metadata. SOTorrent filters this data by focusing solely on edits that alter the actual content of posts. This filtering is done based on specific `PostHistoryTypeId` values—such as `2: Initial Body`, `5: ↷ Edit Body`, and `8: Rollback Body`—which signify modifications to the body content of posts (see the top three rows of Table 2.1). After identifying content edits, the SOTorrent dataset further breaks down each post's content into discrete text and code blocks and stores them in the `PostBlockVersion` table, with each block linked to its predecessor and successor, forming a continuous version history of individual pieces of content. This linear mapping allows for precise tracking of how individual text and code snippets evolve over time.

Furthermore, SOTorrent includes the `PostVersion` table, which contains only post versions from the `PostHistory` table where the content has been modified. This is done by extracting versions from the `PostHistory` table and filtering them based on their corresponding `PostHistoryTypeId` values to include only those with content changes. The table also establishes a predecessor/successor relationship, where each post version is linked to a predecessor and successor. The `PostReferenceGH` table stores references to Stack Overflow posts found within open-source GitHub projects. This table includes the URLs of the Stack Overflow posts, along with details about the source files and repositories that contain these references. Through this methodology, SOTorrent enables a more fine-grained analysis of how code and text snippets change over time, offering the version control needed for studying the impact of those changes on both developers and prior research. This dissertation only utilizes the aforementioned SOTorrent database tables; however, the dataset includes other tables that are not described here.

3
Effects of Evolution on Developers

As discussed in the previous chapter, content on Stack Overflow is evolves continuously as the developer community adds new snippets and updates existing ones [15], and in doing so, fixes bugs and vulnerabilities in code snippets. When developers reuse snippets from Stack Overflow, they create a dependency on that code. If the reused snippets are later updated on the platform with bug or security fixes, failure to incorporate those updates leaves the developers' code unnecessarily vulnerable or flawed. In essence, code on Stack Overflow should be treated like any other dependency, requiring version control and proper management. These observations set the foundation for this chapter as we explore the following research questions:

> **RQ1:** Do Stack Overflow code artifacts reused in developer code bases evolve on Stack Overflow?
>
> **RQ2:** Can we find evidence that developers monitor Stack Overflow for code updates?
>
> **RQ3:** Do developers miss security fixes on Stack Overflow for code also present in their code bases?

Past work by Manes and Baysal [71] has shown that code on GitHub, which attributes its origin to Stack Overflow, and the original code snippet on Stack Overflow evolve independently from each other. This suggests that developers do not track changes to reused Stack Overflow code. Hong et al. [49] investigated the change histories of Stack Overflow snippets to identify insecure code snippets. They found the first evidence that outdated, vulnerable snippets can be found in the current code bases of open-source C/C++ projects. Zhang et al. [118] detected vulnerable C/C++ snippets on Stack Overflow and found that snippet revisions are associated with reducing the number of code weaknesses.

To answer our research questions, we take inspiration from prior works, but we devise a methodology that addresses the shortcomings of those works. First, attributing Stack Overflow snippets when copying them to developer code bases is the exception and not the norm [13]. Thus, limiting the dataset to snippets of attributed posts provides only an incomplete, biased picture. Second, automatically classifying code snippets as vulnerable is an unsolved problem when not scoped to a narrow problem domain (e.g., crypto APIs [31]) and even state-of-the-art work [49] relies on heuristics and idiosyncrasies of a single programming language. Thus, the strategy to identify vulnerable snippets on a large scale and re-identify them in developer code bases is inherently limited by the classification tool (e.g., supported languages or accuracy). In our approach, we first used clone detection to identify all the versions of 1.5M code snippets from Stack Overflow (in total, 3.5M code snippet versions) in the code bases of 11,479 popular open-source projects on GitHub without limiting our dataset to only attributed snippets. We focused on code snippets in the most popular programming languages Python, Java, JavaScript, and C. From that, by comparing the change history of Stack Overflow snippets with the latest version of code on GitHub, we retrieve the set of code snippets that evolved on Stack Overflow since appearing in developer code bases, and that is hence outdated in the GitHub projects ($\rightarrow$ RQ1). Additionally, applying clone detection to the entire change history of the GitHub code bases and that of Stack Overflow posts allows us to detect whether developers updated their code to a newer version of the Stack Overflow snippet ($\rightarrow$ RQ2). Lastly, we used a combined method of natural language processing of commit messages and comments, analyzing code properties, and manual confirmation to determine which evolved code snippets on Stack Overflow contain security-relevant edits that have not been transferred to the code bases on GitHub ($\rightarrow$ RQ3).

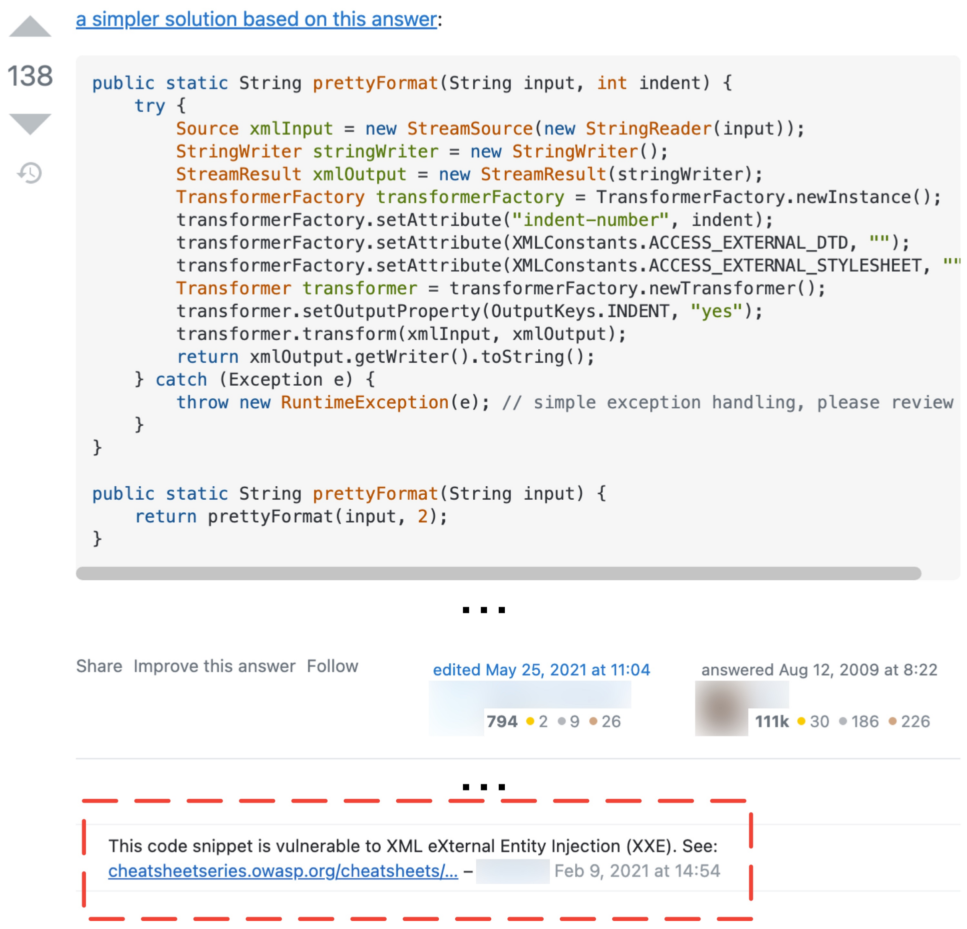
We found that 2,405 code snippet versions reused in 22,735 distinct GitHub source files were outdated, affecting 2,109 GitHub projects. Among those outdated snippet versions, we manually verified 26 to have a security-relevant update on Stack Overflow that fixes a known vulnerability. The fixes to those vulnerabilities on Stack Overflow were not reflected in 43 highly popular, non-forked open-source projects to whose maintainers we disclosed our findings. Further analyzing the 43 affected projects reveals that it took, on average, 1,060$\pm$506 days (max. 3,303 days) from when an insecure Stack Overflow code snippet is committed to a GitHub project until the time a Stack Overflow comment raising a security warning is made. However, as soon as a security warning is raised, it takes, on average, 296$\pm$200 days (max. 1,516) for a security issue in a code snippet to be fixed. For a time difference this long, it would be non-trivial for developers to manually track Stack Overflow for updates or be aware of (or react to) security discussions around a piece of code they reused such a long time ago.

This chapter is structured as follows: In Section 3.1, we present a real-world example from our findings to illustrate why missing security fixes on Stack Overflow can harm production software. In Section 3.2, we explain our methodology for answering the research questions, including how we gathered data from Stack Overflow and GitHub (Section 3.2.1), how we detected code reuse from Stack Overflow (Section 3.2.2), and why a filtering pipeline was necessary to manage the volume of code clones (Section 3.2.3). We also discuss how we identified outdated code reuse (Section 3.2.4), classified security fixes (Section 3.2.5), and verified them manually (Section 3.2.6). Finally, the chapter presents our findings (Section 3.3), reviews limitations (Section 3.5), related work (Section 3.6), and concludes with future directions in Section 3.7.

3.1 Motivating Example

Posts on Stack Overflow are not static content but strongly influenced by the community's reaction (e.g., through comments or up/down votes), which in turn can motivate the creator of a post, or the developer community, to edit and expand the content of posts. For the creation of every post as well as for every post edit, Stack Overflow creates a post version object to record all text and code changes made, together with the date the change was made and the author that made the changes. An optional commit message by the user summarizing the changes they made is stored alongside the post version. The post version object is then stored in a post change history, a collection of all the post versions. Stack Overflow provides version control for posts by recording all the edits made to posts using a post change history. Even though Stack Overflow has a comprehensive version control for posts, only the latest version of posts is displayed on the Stack Overflow website.

We briefly illustrate the versioning of Stack Overflow posts using a concrete example



a simpler solution based on this answer:

```
public static String prettyFormat(String input, int indent) {
    try {
        Source xmlInput = new StreamSource(new StringReader(input));
        StringWriter stringWriter = new StringWriter();
        StreamResult xmlOutput = new StreamResult(stringWriter);
        TransformerFactory transformerFactory = TransformerFactory.newInstance();
        transformerFactory.setAttribute("indent-number", indent);
        transformerFactory.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
        transformerFactory.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, ""
        Transformer transformer = transformerFactory.newTransformer();
        transformer.setOutputProperty(OutputKeys.INDENT, "yes");
        transformer.transform(xmlInput, xmlOutput);
        return xmlOutput.getWriter().toString();
    } catch (Exception e) {
        throw new RuntimeException(e); // simple exception handling, please review
    }
}

public static String prettyFormat(String input) {
    return prettyFormat(input, 2);
}
```

...

Share Improve this answer Follow

edited May 25, 2021 at 11:04
[redacted]
794 ● 2 ● 9 ● 26

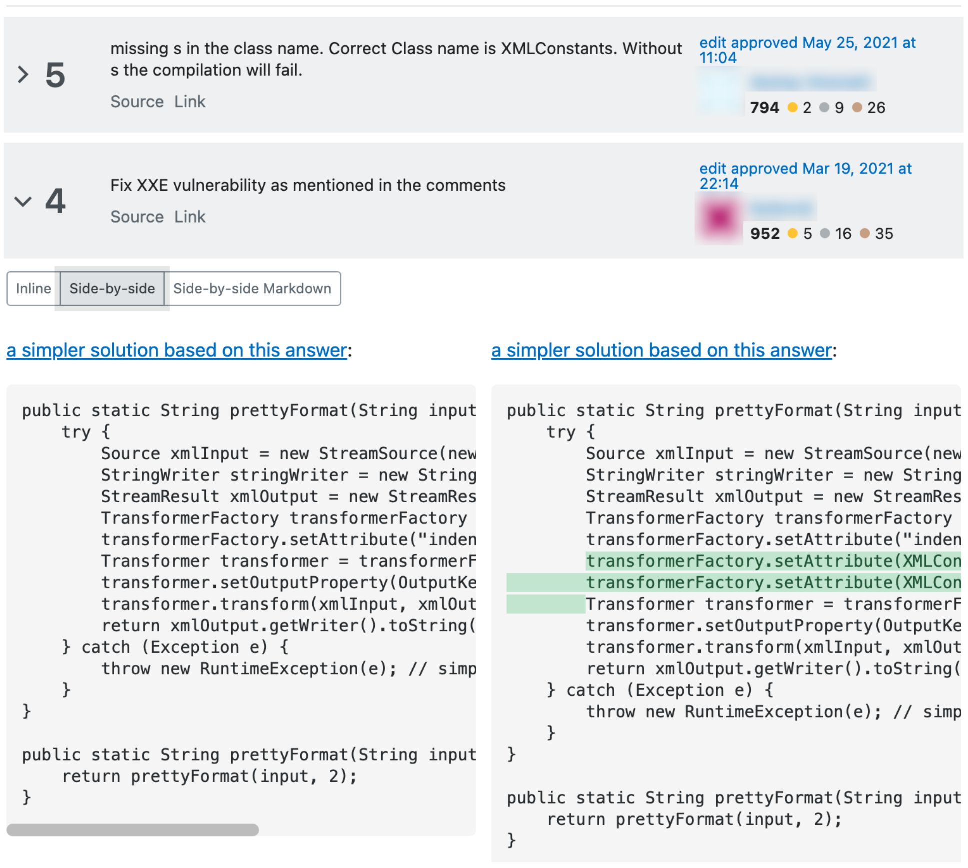
answered Aug 12, 2009 at 8:22
[redacted]
111k ● 30 ● 186 ● 226

...

This code snippet is vulnerable to XML eXternal Entity Injection (XXE). See: cheatsheetseries.owasp.org/cheatsheets/... – [redacted] Feb 9, 2021 at 14:54

Figure 3.1: A Stack Overflow answer post with comments and change history. The original post was created on Aug 12, 2009, and the code was edited on Feb 23, 2011. Almost ten years later, a comment (dashed line) points out an *XXE* vulnerability, which led to a code revision (see Figure 3.2).

from our results. Figure 3.1 depicts an example answer post (id `1264912`) [98] that was created on Aug 12, 2009. In the last comment of this answer post (marked with a dashed line), posted almost ten years after `version #3` of the post, a developer pointed out that the code snippet contains an `XML eXternal Entities (XXE)` injection vulnerability (CWE-611) and provided a link to the OWASP website detailing the nature of the vulnerability. This vulnerability, if not fixed, may allow an attacker to disclose confidential data or server-side request forgery. For this reason, the vulnerability is listed in the OWASP top 10 web application security risks. Figure 3.2 shows the last versions of the answer post. In `version #4` of the answer, the vulnerability was fixed thanks to this comment, as also noted in the commit message of that version. In this code version, the changes include the fix recommended in the OWASP website by protecting the `TransformerFactory` Java object. In our results, we found the vulnerable `version #3` of that code snippet in the current code base of the *Apache Chemistry* project and the popular *Apache Lucene-Solr* project. The vulnerable snippet was committed to the


› 5 missing s in the class name. Correct Class name is XMLConstants. Without s the compilation will fail.
Source Link
edit approved May 25, 2021 at 11:04
[redacted]
794 ● 2 ● 9 ● 26

⌄ 4 Fix XXE vulnerability as mentioned in the comments
Source Link
edit approved Mar 19, 2021 at 22:14
[redacted]
952 ● 5 ● 16 ● 35

Inline | Side-by-side | Side-by-side Markdown

a simpler solution based on this answer:

```
public static String prettyFormat(String input
    try {
        Source xmlInput = new StreamSource(new
        StringWriter stringWriter = new String
        StreamResult xmlOutput = new StreamRes
        TransformerFactory transformerFactory
        transformerFactory.setAttribute("inden
        Transformer transformer = transformerF
        transformer.setOutputProperty(OutputKe
        transformer.transform(xmlInput, xmlOut
        return xmlOutput.getWriter().toString(
    } catch (Exception e) {
        throw new RuntimeException(e); // simp
    }
}

public static String prettyFormat(String input
    return prettyFormat(input, 2);
}
```

a simpler solution based on this answer:

```
public static String prettyFormat(String input
    try {
        Source xmlInput = new StreamSource(new
        StringWriter stringWriter = new String
        StreamResult xmlOutput = new StreamRes
        TransformerFactory transformerFactory
        transformerFactory.setAttribute("inden
        transformerFactory.setAttribute(XMLCon
        transformerFactory.setAttribute(XMLCon
        Transformer transformer = transformerF
        transformer.setOutputProperty(OutputKe
        transformer.transform(xmlInput, xmlOut
        return xmlOutput.getWriter().toString(
    } catch (Exception e) {
        throw new RuntimeException(e); // simp
    }
}

public static String prettyFormat(String input
    return prettyFormat(input, 2);
}
```

Figure 3.2: *Fourth version* of the answer post in Figure 3.1. Based on the pointed-out XXE vulnerability, the 4th version of the post (on Mar 19, 2021) includes the corresponding fix, as also noted in the commit message of the version.

Apache Lucene-Solr project on Jan 25, 2012, almost one year after `version #3` was created and nine years before the vulnerability was fixed in `version #4`.

The evolution of artifacts on Stack Overflow is similar to how software evolves. A program evolves when bugs or vulnerabilities are fixed, new features are added, or code is updated due to changes in requirements. A version control system tracks changes to source code and other artifacts, while a bug tracking system is used to triage and solve bugs or issues. In this work, we transfer the concepts of bug tracking and version control from software projects to Stack Overflow. In the example above, the post change history provides version control for the code snippets, and the comments provide a form of loosely-coupled issue tracking.[1]

[1]It is worth pointing out that the commenting feature is not a bug-tracking system per se. Only comments about a buggy or insecure code snippet are considered a bug report.

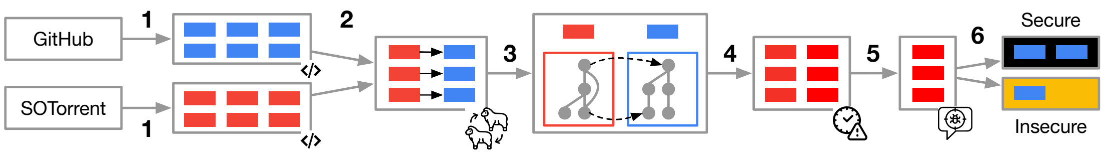


Figure 3.3: Our research methodology: (1) extraction of Stack Overflow code snippets and developer code from GitHub, (2) clone detection between Stack Overflow change histories and GitHub, (3) timeline analysis and filtering to identify best candidates for code clone pairs, (4) identifying outdated (RQ1) and updated (RQ2) snippets by extracting code evolution on Stack Overflow and comparing change histories between platforms, (5) filtering reused outdated Stack Overflow posts without comments and revisions without commit messages, and (6) classifying security-relevance of Stack Overflow code edits based on comments, commit messages, code changes, and manual confirmation for RQ3, detecting fixes to reused vulnerable code snippets which did not propagate to GitHub.

3.2 Research Methodology

Our data-driven approach to answering our questions is depicted in Figure 3.3. In general: We used *clone detection* to find code in open-source projects that are highly similar to code snippets on Stack Overflow. For each occurrence of reused code, we compared the change histories of this code between the open-source projects and Stack Overflow. From this comparison, we identified instances of outdated Stack Overflow code snippets within open-source projects. In a final step, we analyzed the change history of the reused, outdated code snippets for security issues and corresponding fixes by mining bug-fix commit messages and using information from the discussions on Stack Overflow around corresponding posts, and then mapped those results to the open-source projects containing affected code snippets.

Overall, we carried out this analysis process for four different programming languages: *Java* and *C*, since they are the focus of the majority of prior security-related studies on Stack Overflow [31, 36, 49, 116, 13, 1, 49, 110], and *Python* and *JavaScript* as they are currently the most popular languages on Stack Overflow [100]. We now describe the used data sets and the individual steps from Figure 3.3.

Table 3.1: Distributions of the code snippet data set from Stack Overflow showing each programming language's initial and final sample size. The final sample is selected by removing all single-version snippets and snippets containing at least one version with less than 10 LoC in their change history.

	Initial		**Final** (> 1 Version ∩ >= 10 LoC)						
					Versions				
Language	**No.**	**ϕ Score**	**No.**	**ϕ Score**	**No.**	**Min**	**Max**	**ϕ No.**	**Median**
Java	2,429,964	1.9	**524,099** (21.6%)	2.7	1,209,700	2	30	2.3	2
C	404,051	1.6	**112,138** (27.8%)	1.9	274,508	2	36	2.4	2
Python	1,580,623	2.0	**336,328** (21.3%)	2.9	807,345	2	26	2.4	2
JavaScript	2,704,287	1.8	**530,205** (19.6%)	2.9	1,227,326	2	743	2.3	2
Overall	7,118,925	1.9	**1,502,770** (21.1%)	2.8	3,518,879			2.3	

3.2.1 Study data sets

We used two main data sources for our study—SOTorrent [15] and GitHub—which are also commonly used in related works.

Stack Overflow code snippets: Our data set of Stack Overflow code snippets is based on the *December 31, 2020* release of the SOTorrent data set [15]. This data set stores the entire change history of all text and code on Stack Overflow, which provides the necessary version control that we need to track and analyze changes to individual code snippets. We will refer to a *snippet change history* when we mean the entirety of all versions of a code snippet.

Since SOTorrent does not contain information about the used programming language, we used Guesslang [42] to determine the programming language for the most recent versions of all code snippets with at least 5 LoC. Out of $31,287,646$ code snippets in the data set, Guesslang identified the language of $31,202,349$ code snippets. No assignment was possible for the remaining $85,297$ code snippets, mainly because the code snippets do not contain actual code. Our data set contains code snippets in 30 different programming languages.

A close examination of this data set showed that in line with prior work [115] many short code snippets consisted of highly trivial code, such as boilerplate code. Given that the first step in our study is to match Stack Overflow code snippets with code from open-source projects using clone detection, this would lead to a very high number of positive matches from whose analysis no meaningful information could be drawn. To reflect this fact, we limited our data set to only those snippets where each version in the change history consists of at least 10 LoC. Further, since our study focuses on code evolution, we excluded all code snippets that have never undergone any changes since their creation. We considered all code snippets that meet these criteria for clone detection and disregarded any other factors, such as post popularity, post views, or the reputation of users.

Table 3.1 provides an overview of the corresponding sample sizes for the four considered programming languages. We considered 1.5M code snippets, where most are written in Java (524k; 34.9% of final sample size) or JavaScript (530k; 35.3%). Those snippets have 3.5M versions, whereas the average snippet in our final data set has 2.3 versions. The average post in our initial data set had a score of 1.9, and the average post in our final sample had a score of 2.8, indicating that posts with revisions have higher scores.

Open source projects: We used GitHub for our data set of open-source projects, as it is one of the most popular code hosting platforms in open-source communities [41]. Code on GitHub is version controlled using the Git version control system, allowing us to measure historical code overlap between Stack Overflow and GitHub. For our sample, we focused on popular and well-maintained projects, as we consider the overall impact of missed security-critical updates to be the highest and because we assume that the code base is steadily maintained over a longer period.

Using the GitHub Search API, we retrieved a list of projects sorted by popularity (number of stars) for each of the four programming languages considered in this study. We then manually inspected each project in this initial candidate list and excluded: a) *forks* to avoid including essentially identical code in the analysis; b) *archived* (retired) projects as it is not reasonable to expect the maintainers to provide a bug fix in case they are notified; and c) projects that are known to be learning materials[2] because the focus of our study lay on real software projects. Our final GitHub sample consisted of 11,479 individual projects (Java: 2,290, C: 2,241, Python: 3,107, JavaScript: 3,841) containing 4,098,397 source files.

3.2.2 Clone detection

To identify shared code between our GitHub and Stack Overflow samples, we used clone detection to search for complete or partial matches between all of the source files in the current versions of the open-source projects and the complete change histories of code snippets from Stack Overflow. As a first step, we only considered the most recent version of the open-source projects since we were interested in projects that currently have a dependency on Stack Overflow code snippets. The result is a list of all the source files code with Stack Overflow in common. In a second step, we then rerun the clone detection for the files in this list, but this time for all previous versions stored in the Git version history of a project—this is a *cartesian-product clone detection* between the change histories of GitHub projects and their Stack Overflow dependencies.[3]

To account for the differences in programming languages, we used two different clone detection tools to identify common code: *NiCad* [87] for Java and C, and *PMD Copy-Paste Detector* [79] for Python and JavaScript. Both tools have been fine-tuned with parameters specific to Stack Overflow code snippets to achieve the best possible results. For PMD CPD, we adopted the fine-tuned parameters from Baltes et al. [13], which were already optimized for usage with Stack Overflow code. For NiCad, we fine-tuned the parameters ourselves and provide a detail explanation of the fine tunning process in Appendix A.1. It is important to note that PMD CPD can only recognize exact copies of Python and JavaScript code and only returns exactly matching sections in source files. NiCad also recognizes slightly modified copies and returns a similarity score between GitHub file sections and code snippets when this score exceeds 83% (the cutoff is based on our calibration process).

We identified 108,134 file sections in 3,439 GitHub projects where code common with *multiple* snippet versions from Stack Overflow were found. A section of a GitHub file is defined by the triple 1) start and 2) end line in 3) a specific commit. For most file sections, we found several Stack Overflow code snippet versions with (nearly) identical code, which aligns with findings of prior work [14] on code duplication on Stack Overflow. On average, for every single Python source file section, we found the same code in 4.81 code snippets (max. 68 snippets), for Java in 3.19 snippets (max. 67), for C in 1.33 snippets (max. 26), and for JavaScript in 2.97 snippets (max. 46). We found 8,538 distinct reused code snippets with together 16,479 versions (from 8,436 distinct posts), where the average file section had 3.22 snippets (and 6.55 versions) as matching clones.

[2]Examples of such projects: `https://github.com/topics/awesome`

[3]A cartesian-product clone detection between all versions of all Stack Overflow code snippets and the entire change history of all source files in our GitHub projects sample was not feasible. Already our step-wise approach took around 17 weeks using four servers (each with 64-core Intel Xeon E7-8867 and 1.5 TB Ram).

Table 3.2: The number of *distinct* posts, code snippets, and snippet versions for each filter step, the average no. of posts, snippets, and snippets version for each distinct detected GitHub file section, as well as the *total* count of posts, snippets, and versions removed by each filter.

	Input to filter (Distinct)				Average per GitHub file section			Removed by filter (Total)		
Filter	**File sections**	**Posts**	**Snippets**	**Versions**	**ϕ Posts**	**ϕ Snippets**	**ϕ Versions**	**Posts**	**Snippets**	**Versions**
F1: Attribution	108,134	8,436	8,538	16,479	3.2122	3.2152	6.5477	1,020	1,026	2,069
F2: Commit Date	102,236	8,118	8,217	15,827	3.1625	3.1656	6.4022	3,535	3,576	7,076
F3: Version Similarity	84,542	6,407	6,475	12,294	2.8375	2.8406	5.2595	0	0	5,673
F4: Snippet Similarity	84,542	6,407	6,475	7,086	2.8375	2.8406	2.8406	665	671	683
F5: Post Type	84,542	6,054	6,117	6,712	2.8176	2.8206	2.8206	1,151	1,155	1,190
F6: Post Popularity	84,542	5,237	5,296	5,850	1.6471	1.6500	1.6500	1,295	1,301	1,349
F7: Latest Post	84,542	4,336	4,389	4,880	1.0379	1.0405	1.0405	162	162	163
Final	84,542	4,241	4,272	4,755	1.0000	1.0000	1.0000	–	–	–

3.2.3 Identifying best candidates

Not all of the ≈16k snippet versions from the 8.5k snippets are relevant to our research questions. Since we are interested in measuring the impact of reused *outdated* snippets from Stack Overflow on the security of open-source GitHub projects, we can filter snippets that do not fit this setting. To this end, we implemented a *filter pipeline* that gradually reduced the number of candidate snippet versions per GitHub file section in our clone detection results. Table 3.2 details the filtering process for the clones to our initial 108,134 distinct file sections. The input to each filter step is reported as a count of *distinct* posts, snippets, and snippet versions. The average posts/snippets/versions per file section provide an overview of the *1:n* mapping between a single file section and detected clones on Stack Overflow. In cases where this relation becomes *1:0*, the file section is removed. In the following, we describe the individual filter steps.

First, we apply filters that are motivated by prior studies on the behavior of developers using Stack Overflow:

F1. Attributions Filter: Baltes et al. [13] have shown that some developers copy code from Stack Overflow and attribute the Stack Overflow source, a behavior in line with the Stack Overflow license. Accordingly, this filter relies on attribution as the strongest indication of code copy from specific Stack Overflow posts. If a section of a file or its immediate surrounding context includes attribution to a post on Stack Overflow, this filter excludes all snippets that do not originate from the corresponding post. In cases where a specific answer or question is not referenced (i.e., a Stack Overflow thread is attributed), all but the snippets belonging to the question and all answers in the thread are excluded.

F2. Commit Date Filter: Prior works [31, 29] have shown that developers seemingly copy code from Stack Overflow to their projects, and we apply this same assumption for the reduction of our data set. A logical prerequisite for a causal link between a Stack Overflow post and a GitHub source file is that the post existed before the code was committed on GitHub. Accordingly, this filter compares the change histories of the code snippets on Stack Overflow and the file sections on GitHub. It excludes all snippets created after the commit that introduces a snippet into a source file.

The number of Stack Overflow clone candidates per GitHub file section after **F2** still poses a severe challenge for the next steps in our analyses to determine the security impact of reused code. Ideally, we could classify all remaining ≈12k reused Stack Overflow snippets in our data set as security-relevant or not. However, during the project, it became evident that this type of analysis can only be partially automated with current tools and still requires much manual verification work. Although there are already promising approaches [49, 31, 36] for automating this task, these approaches have proven to be too limited in scope (e.g., works only with one programming language or a narrow aspect of security) or unsuitable for our current situation (for details, see Section 3.2.5). For this reason, we added additional filtering steps to reduce the number of matches between GitHub file sections and Stack Overflow snippet versions to a feasible amount for manual analysis.

The additional filters follow the basic idea that if a developer has reused code from Stack Overflow, it is rather unlikely that they have taken code from several different Stack Overflow snippets or different versions of them. Hence, this approach requires making several assumptions about how developers use Stack Overflow and comes with the limitation that the mapping may be subject to error. Despite these limitations, we believe this approach is currently the only practical solution allowing us to perform subsequent analyses of security relevance. When creating the additional steps of the filter pipeline, we oriented ourselves to first go through steps that leave the least room for interpretation.

In the next steps, we prioritized clones with higher similarity over those with lower similarity. The intuition is that the less similar the Stack Overflow snippets and GitHub code are, the less likely they are directly related. As PMD CPD only returns exactly matching sections and snippets, these filters are ineffective for Python and JavaScript.

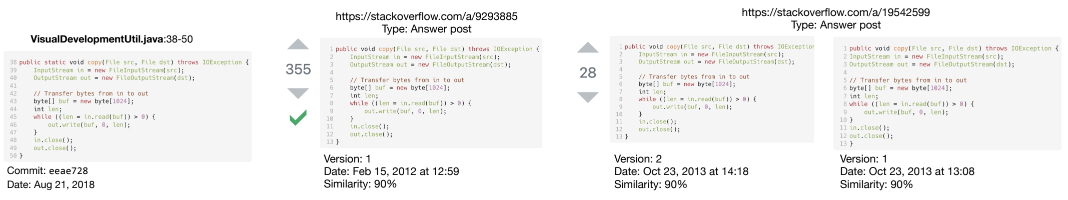
F3. Version Similarity Filter: If multiple versions of the same snippet are found in a file section, this filter excludes all versions except for the one(s) with the highest similarity. If multiple versions have an equally high similarity score, the filter selects the latest version, ensuring the most conservative result in our later analysis.

F4. Snippet Similarity Filter: This filter compares the similarity scores of the snippets for each file section, excluding all except the one(s) with the highest scores.

Lastly, if the above filters did not yield the best candidate version, we added filters considering developers' most likely behavior while using Stack Overflow.

F5. Post Type Filter: Following Baltes et al. [13], developers are more likely to reuse code from answer posts than question posts. Accordingly, if a file section contains clones of snippets from answers, this filter excludes all snippet clones from question posts from further consideration.

F6. Post Popularity Filter: This filter results directly from the nature of the Stack Overflow website. By default, Stack Overflow users are shown answers within posts sorted according to their score, which is a measurement of usefulness by other members of the Stack Overflow community. In line with other review systems (e.g., app stores and online markets), we assume developers will use this community-driven feedback when selecting information for their current programming task. Accordingly, we filter code snippets based on the popularity of their respective posts and select the code snippets belonging to the most popular post. However, a post might be popular for various reasons that cannot be easily determined automatically. To better consider the



VisualDevelopmentUtil.java:38-50

```
38 public static void copy(File src, File dst) throws IOException {
39     InputStream in = new FileInputStream(src);
40     OutputStream out = new FileOutputStream(dst);
41 
42     // Transfer bytes from in to out
43     byte[] buf = new byte[1024];
44     int len;
45     while ((len = in.read(buf)) > 0) {
46         out.write(buf, 0, len);
47     }
48     in.close();
49     out.close();
50 }
```

Commit: eeae728
Date: Aug 21, 2018

https://stackoverflow.com/a/9293885
Type: Answer post

```
public void copy(File src, File dst) throws IOException {
    InputStream in = new FileInputStream(src);
    OutputStream out = new FileOutputStream(dst);

    // Transfer bytes from in to out
    byte[] buf = new byte[1024];
    int len;
    while ((len = in.read(buf)) > 0) {
        out.write(buf, 0, len);
    }
    in.close();
    out.close();
}
```

Version: 1
Date: Feb 15, 2012 at 12:59
Similarity: 90%

https://stackoverflow.com/a/19542599
Type: Answer post

```
public void copy(File src, File dst) throws IOExceptior
    InputStream in = new FileInputStream(src);
    OutputStream out = new FileOutputStream(dst);

    // Transfer bytes from in to out
    byte[] buf = new byte[1024];
    int len;
    while ((len = in.read(buf)) > 0) {
        out.write(buf, 0, len);
    }
    in.close();
    out.close();
}
```

Version: 2
Date: Oct 23, 2013 at 14:18
Similarity: 90%

```
public void copy(File src, File dst) throws IOException {
InputStream in = new FileInputStream(src);
OutputStream out = new FileOutputStream(dst);

// Transfer bytes from in to out
byte[] buf = new byte[1024];
int len;
while ((len = in.read(buf)) > 0) {
    out.write(buf, 0, len);
}
in.close();
out.close();
}
```

Version: 1
Date: Oct 23, 2013 at 13:08
Similarity: 90%

(a) File section on GitHub **(b)** Stack Overflow post 9293885 **(c)** Stack Overflow post 19542599

Figure 3.4: Example for the execution of the filter pipeline. Filter **Version Similarity (F3)** removes version 1 of post 19542599. Filter **Post Popularity (F6)** selects revision 1 of post 9293885 as the final code snippet.

relevance of a post for a detected clone, this filter first computes the lines of code (LoC) of all the code snippets reported as clones and the file section containing the reported clones. If code snippets have the same LoC as the GitHub file section, the filter selects only the posts of those code snippets for comparing popularity. This means we assume at this point that developers rather copy entire code snippets than cherry-pick parts of them. If no such snippet exists, the filter selects the most popular one(s) from all posts.

F7. Latest Post Filter: This is the final filter in the pipeline, and at this point, all the remaining clones belong to the same post type (question or answer) and are equally popular. If the remaining clones belong to question posts, this filter selects the snippet belonging to the newest post. Otherwise, if the clones belong to answer posts and there is a single accepted answer, the code snippet in the accepted answer is selected. If there is more than one accepted answer, the code snippet belonging to the newest accepted answer is selected.

Example: We now illustrate our filter pipeline, using Figure 3.4, to go through the procedure for a concrete example. Figure 3.4a shows a code section from the Apache NetBeans project on GitHub—lines 38–50 of `VisualDevelopmentUtil.java` source file. Figure 3.4b and Figure 3.4c show one, respectively, two matching Stack Overflow code snippet versions from two different answer posts. For the snippet in Figure 3.4c, the clone detector found two different versions (version 1 and 2) that match the file section in Figure 3.4a. Our filter pipeline sequentially determines the best candidate version among those three candidates as follows: The *attributions filter (F1)* will not discard any clones because the GitHub source file contains no attributions to Stack Overflow. Commit `eeae728` introduced the reused code on Aug 21, 2018, after any of the three snippets were posted (2012 and 2013, respectively). Thus, the *commit date filter (F2)* does not remove any candidates either. Next, the two versions of the snippet in Figure 3.4c have an equal similarity of 90% with the file section. Hence, the *version similarity filter (F3)* will select version 2, since it is the newest one, and discard version 1, reducing the overall set of candidates to two versions from two code snippets. Since those two versions have the same similarity of 90%, the *snippet similarity filter (F4)* cannot reduce the candidate set further. Similarly, both code snippets are in answer posts—post *9293885* [97] and *19542599* [99]—consequently, the *post type filter (F5)* will not remove any snippet. Next, the *post popularity filter (F6)* filter first determines that both snippets have the same LoC (12) as the file section and, thus, selects the more popular snippet, which is the snippet in Figure 3.4b with a score of 355 in contrast to the other snippet with a score of 28. Since this filter already reduced the candidate set to one, the *latest post filter (F7)* had no effect. As a result, we consider only the post and code snippet in Figure 3.4b in the further analysis steps.

Final data set: As depicted in Table 3.2, after the filtering pipeline, we ended up with 4,755 distinct snippet versions from 4,272 snippets (*JavaScript*: 1,979; *Java*: 909; *Python*: 1,112; *C*: 272) from 4,241 posts that were reused in 84,542 distinct file sections in 2,824 GitHub projects. Thus, we identified reused Stack Overflow code snippets in almost 25% of all 11,479 investigated GitHub projects. Based on this final data set, we answer our research questions.

3.2.3.1 Filter pipeline evaluation

To understand whether the filter pipeline approximates developer behavior, we evaluated it using a ground truth dataset comprising code snippets copied from Stack Overflow to GitHub. Since it is not trivial to determine the exact source a piece of code is copied from (see Section 3.5.2), we relied on *attribution* since it is the strongest indicator of code reuse from Stack Overflow [71, 13, 110]. We collected all source files from our data set of detected clones containing an attribution to answer posts since developers copy code from attributed answers [13]. Overall, 292 source files attribute 37 distinct answer posts. Those source files contain 315 file sections for which we detected code clones in 219 distinct posts (*JavaScript*: 97; *Java*: 79; *Python*: 41; *C*: 2). Thus, 182 posts were also reported as clones but were not attributed. This enables us to determine whether, for each of the 315 file sections, the filter pipeline *without* the *Attributions Filter (F1)* filter will select clones from the 37 attributed posts or the 182 unattributed posts.

The filter pipeline with only filters **F2** through **F7** selected the code snippets from the attributed posts in 304 (96.5%) cases. Only in 11 cases did the filters not select code snippet clones from the attributed posts. In 9 of these 11 cases, source files contained attributions to several Stack Overflow posts. This may indicate that the filter pipeline makes the wrong judgment in cases where a source file attributes multiple Stack Overflow posts in direct proximity to reused code snippets. These results show that the filter pipeline reflects the behavior of developers reusing code from Stack Overflow.

3.2.4 Determining Outdatedness

Using our filtered data set, we answer **RQ1** and **RQ2**. To determine whether a clone of a snippet version is outdated or not, we check whether the *reused version* on GitHub is the latest version of the snippet on Stack Overflow. If the reused version is not the latest, the snippet evolved on Stack Overflow, rendering the reused GitHub version outdated. Otherwise, the copied version is up-to-date. Answering **RQ2** requires tracing the evolution of copied Stack Overflow snippet versions in GitHub source files. To determine whether an update to a snippet on Stack Overflow also transferred to the reused code snippet on GitHub, we look backward at the change histories of GitHub files and Stack Overflow code snippets. We leverage the result of our cartesian-product clone detection to detect whether an older version of the snippet in a GitHub file matches an older version of the Stack Overflow snippet, indicating that the GitHub project

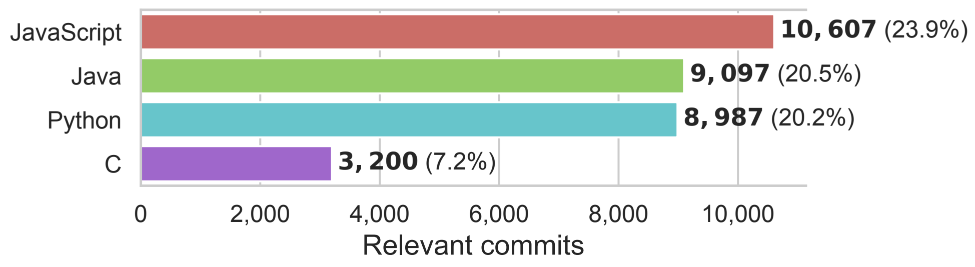


Figure 3.5: Distribution of *potential* security-relevant commit messages for different languages based on keyword-search.

developer became aware of the change on Stack Overflow (e.g., by monitoring the post, potentially proposing the edit on Stack Overflow or other channels like independent code origins).

3.2.5 Finding Security Fixes

To answer **RQ3** as to whether developers miss security-relevant updates on Stack Overflow for reused outdated code, we needed to find a way to filter out the large fraction of inconsequential code changes on Stack Overflow [89] and to determine whether an update to a code snippet version fixes a security issue. However, determining whether a code edit fixes a security issue is a non-trivial task, especially for incomplete/not self-contained code snippets as found on Stack Overflow. To the best of our knowledge, there are no generic, automated tools capable of determining whether or not a code update fixes a security issue. Prior studies on Stack Overflow with the need to classify the security of code snippets are either too narrowly focused [31, 116, 1]—for example, only misuse of Java crypto APIs—or work only with specific programming languages [110, 36, 1, 31].

The closest to a generic solution for classifying the change histories of Stack Overflow snippets is Dicos [49] (we briefly discussed this in Section 3.6). Using keyword search in comments as well as changes in code snippets, Dicos detects security-relevant changes to snippets. Unfortunately, Dicos' implementation has limitations that prohibit an application in our study. First, it only considers the changes between the first and the last version but not between intermediate versions, i.e., pinpointing the exact Stack Overflow code snippet versions that are vulnerable and bug-fixed. This prohibits the detection of outdated-but-secure snippets in GitHub files (RQ3) and prevents the detection of bug fixing in GitHub projects (RQ2). Second, Dicos' classification requires certain code edits—changes to control flow and security-relevant APIs—which are tailored to the idiosyncracies of C/C++ and further narrow the scope of detection.

Thus, without an automated, suitable classifier for code snippets, we used keyword search in comments (inspired by Dicos) and mining commit messages to find bug-fix commits. The software engineering community has extensively studied the mining of commit messages to find bug fixes. We leveraged the approaches by Pan et al. [58] and by Osman et al. [43] to find potential bug fixes in Stack Overflow code edits. Figure 3.5 provides a quick overview of *potentially* security-relevant commit messages in *all* commit messages in SOTorrent based on the keywords from those prior works.

Since comments and commit messages are natural language texts, a possible approach could be to build an NLP classifier capable of determining whether or not a given sentence raises a bug report or indicates a security fix. However, the only data set of labeled code snippets is by Fischer et al. [31] and is limited to crypto API misuse in Java code. A first experiment based on this data set showed that the resulting detection of *generic* security-relevant text with a classifier trained on this data set performed poorly. Thus, we decided to apply approaches that use keyword search and manually verify security-relevant comments, commit messages, and code edits—which forms a data set for future research on the security of Stack Overflow snippets.

3.2.6 Manual Verification of Security-relevant Edits

The final step of our methodology is verifying whether comments, commit messages, and code snippet edits are security-relevant. Two researchers went through the list of all potential security-relevant changes and manually classified them according to the following three categories:

Security-relevant: A fix in a snippet is security-relevant if the issue fixed has a known weakness to which a Common Weakness Enumeration (CWE) [22] identifier is assigned. The assignment to CWEs is based on a combination of the surrounding context (comments and commit messages) and the *diff* in the post version that fixed the issue.

Bug Fixes: Fixes in this category are general software bugs with no CWE identifiers assigned and, hence, no direct security implication. *Undefined Behaviour* is one of the software bugs in this category. These are specific errors that produce unexpected results in an application (e.g., missing null checks or incorrect arithmetic calculations).

Improvements: Fixes in this category concern edits to reused code snippets that are not in the above categories. Examples include performance improvements, adding new language features, or removing deprecated APIs.

Based on this classification, we answer our **RQ3** whether an open-source contributor missed a security-relevant change on Stack Overflow since the time a reused code snippet was introduced in the open-source project.

3.3 Results and Findings

We now summarize our analysis results to determine whether developers monitor Stack Overflow for updates to code they have reused into their projects ($\rightarrow$ RQ1). Next, we look at whether developers monitor Stack Overflow for updates to code snippets reused in their projects ($\rightarrow$ RQ2). This helps us understand the extent to which developers miss code updates on Stack Overflow ($\rightarrow$RQ1). Afterwards, we examine if developers miss updates that address security vulnerabilities in the code snippets they have used from Stack Overflow ($\rightarrow$ RQ3).

Table 3.3: Summary of *distinct* projects and file sections containing outdated code snippets reused from Stack Overflow, grouped by programming language.

	Outdated Versions	**Affected File Sections**	**Affected Projects**
C	155 (53%)	894	**195** (70%)
Java	417 (44%)	1,380	**426** (62%)
Python	628 (51%)	6,938	**549** (74%)
JavaScript	1,205 (53%)	29,411	**939** (84%)
Total	**2,405** (51%)	**38,623** (46%)	**2,109** (75%)

Table 3.4: Top-10 most reused outdated snippets

Snippet ID	Reuse count	Snippet ID	Reuse count
`184588891`	9,592	`121742444`	1,044
`49722853`	5,662	`67677724`	788
`243061964`	1,372	`23640102`	717
`200993642`	1,271	`216546726`	651
`229059551`	1,227	`188694515`	640

3.3.1 Projects missing updates on Stack Overflow (**RQ1**)

Table 3.3 summarizes the file sections containing outdated versions of Stack Overflow code snippets. Overall, we found that 2,405 snippet *versions* (51% out of 4,755 reused snippets) from 2,305 snippets were outdated, and we found them in 38,623 (46% out of 84,542) file sections on GitHub. Those outdated snippets come from 2,290 distinct Stack Overflow posts and are reused in 22,735 distinct source files from 2,109 distinct GitHub projects. In other words, almost every fifth (2,109 out of 11,479) project in our data set contains *outdated* code from Stack Overflow. Considering only the 2,824 projects with reused code, almost 75% of them contained at least one outdated snippet.

The average affected project had 11.01±5.83 source files with outdated snippets, where JavaScript projects have the highest average number of affected source files (19.13±12.96), and Java projects the lowest number on average (2.69±0.65). Comparing the number of distinct outdated snippet versions (2,405) and affected files (22,735) shows that snippets commonly appear in multiple places in affected projects. The average outdated code snippet *version* in our data set has been reused in 12.14±7.64 *distinct* source files. Snippet `184588891` is the most reused snippet in our data set and was reused in 7,088 distinct source files (and 9,592 times in total). Comparing the number of distinct posts where those outdated snippets have been posted with the number of distinct snippets shows that 15 snippets appeared in the same post as another snippet and that 14 posts contained multiple outdated reused snippets. Figure 3.6 depicts how often each of the 2,302 distinct snippets were reused in an outdated version in GitHub projects. The average outdated snippet in our data set is reused 16.87±9.83 times. Table 3.4 lists the top-10 most reused outdated snippets in our data set.

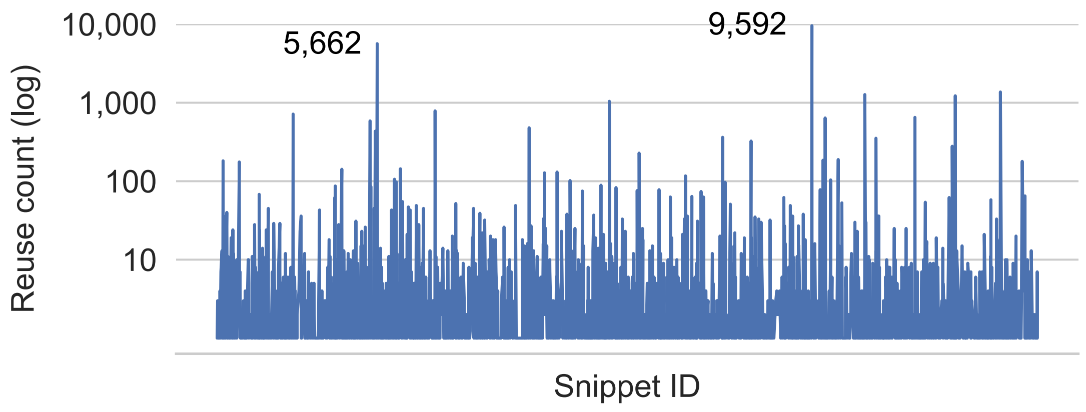


Figure 3.6: Frequency of reused individual snippets.

3.3.2 Detecting code snippet updates (**RQ2**)

After examining the cartesian-product clone detection between change histories of GitHub files and Stack Overflow snippets that were reused in those files, we did not find any evidence for updates in any of the projects, i.e., the reused snippet versions were constant in the source file since the origin commit in which the snippet was introduced.

We found every second reused snippet to be outdated, with no differences between snippets from different languages. This affected every fifth popular open-source project at least once. Further, we did not find any indication that GitHub developers transferred updates to reused Stack Overflow code from Stack Overflow to their code bases. Thus, code snippets seem to be at most up-to-date until a code snippet evolved and never again afterward. This indicates that developers do not track snippets copied from Stack Overflow for changes or are unaware that the code they reused is being discussed and updated/fixed on Stack Overflow.

3.3.3 Security updates on Stack Overflow (**RQ3**)

Using our keyword search approach, we found that 505 posts (22% out of the 2,290 posts linked to the outdated GitHub projects) contain potential security-relevant discussions or code fixes. Together, these posts comprise 278 answers and 227 questions. By *manually validating* each of the 227 question posts, we discovered that only a single question post contains a fix to an insecure code. This fix did not propagate to the AppScale GTS project, to which the insecure version of the snippet was copied. For the 278 answers, we found 25 containing genuine security/bug fixes. The remaining question and answer posts were either not security-relevant or developers copied the patched snippet version.

We found 26 posts containing fixes to 15 distinct security issues missing in 43 (2% out of 2,109) distinct GitHub projects. Table 3.5 lists the types of issues that we found, grouped according to their vulnerability category and referenced by their CWE identifiers. Since a project might be affected by multiple vulnerability types and posts might contain multiple vulnerabilities, we also report the count of distinct posts and

Table 3.5: Distinct security/bug issues found in Stack Overflow snippets, whose fixes on Stack Overflow are not reflected in 43 distinct GitHub projects that contained vulnerable versions of code snippets from Stack Overflow

CWE		Lang.	#Posts	#Count	#Projects
Security-relevant Weaknesses					
120	Buffer Overflow	C	1	1	1
172	Encoding Error	Python	1	1	1
754	Improper Check for Unusual or Exceptional Conditions	Python	1	1	1
		C	1	2	2
		JavaScript	1	2	1
20	Improper Input validation	C	2	2	2
404	Improper release of resource	Java	3	6	6
704	Incorrect Type Conversion or Cast	C	2	5	5
835	Infinite Loop	C	1	2	2
1339	Insufficient Precision	JavaScript	2	4	3
772	Missing release of resource	Java	1	1	1
		Python	1	1	1
690	Unchecked Return Value to NULL Pointer Dereference	JavaScript	1	2	2
475	Undefined Behavior for Input to API	Java	1	2	2
194	Unexpected Sign Extension	C	1	1	1
611	XML eXternal Entity (XXE) Injection	Java	1	2	2
General Bugfixes					
	Undefined behaviour	Python	2	2	2
		Java	1	2	2
		JavaScript	1	1	1
	Others	Java	2	2	2
		Python	1	9	8
	Total (Distinct)		**28 (26)**	**51**	**48 (43)**

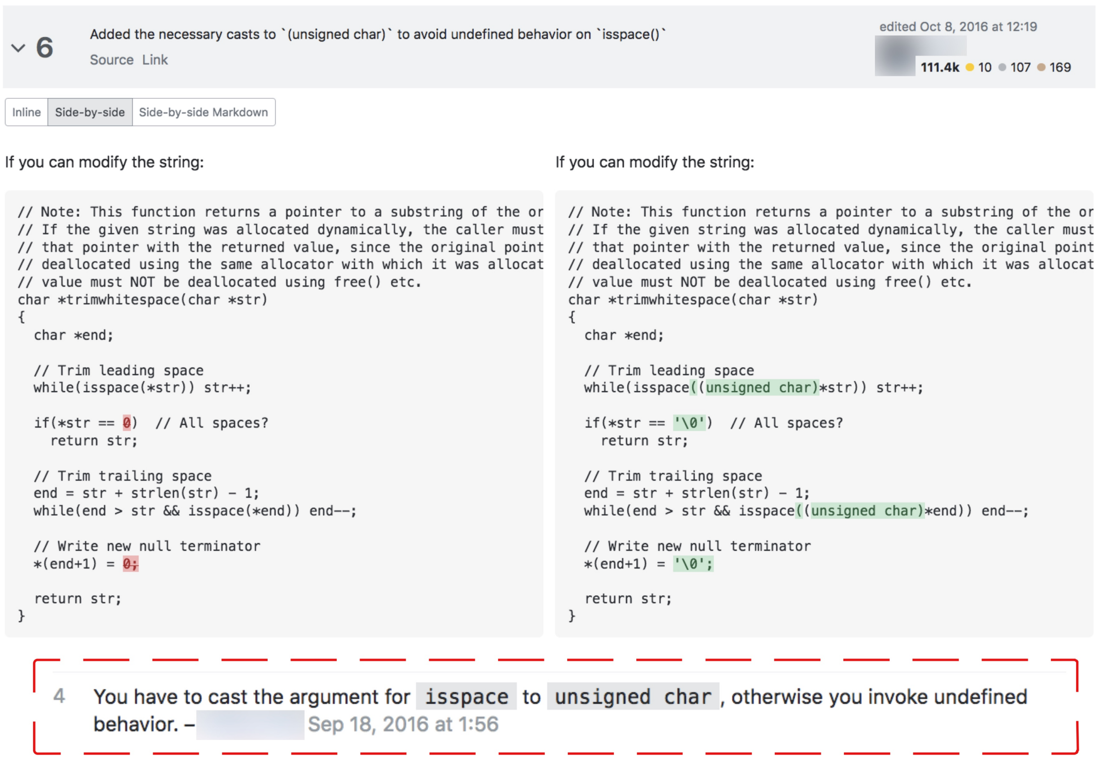
projects. General bug fixes (no CWE assigned) affected 15 distinct projects, while 38 distinct projects copied snippets with weaknesses that have security ramifications. We found multiple vulnerability types in three projects, ngrok-libev (3), hashkill (3), and Mirai-Source-Code (2), and we found one answer post (id `779960`) that contained three vulnerabilities (CWE-754/-835/-20).

The most severe vulnerabilities—according to OWASP—that were fixed on Stack Overflow but missing in open source projects are *XML eXternal Entities injection* (CWE-611), *Buffer Overflow* (CWE-120), and *Improper Input Validation* (CWE-20) vulnerabilities. The most reused, outdated, insecure code snippet is from answer post `122721` [96], which suffers from an Incorrect Type Conversion (704) issue (see Figure 3.7). The code snippet containing the issue was posted in version #1 of the answer on Sep 23, 2008, and was reused in 4 open-source projects. Eight years later, a developer pointed out in the comments on Sep 18, 2016, that the code snippet will invoke undefined behavior if the necessary cast is not added to the `isspace` library function. The issue was subsequently fixed 20 days later. However, this fix did not propagate to the four open-source projects that reused the insecure version. We provide further examples in Appendix A.2.

Table 3.6 shows the 43 affected GitHub projects, which were "watched" on average 157.4 times (max. 1,494), received an average of 4,089.63 stars (max. 70,614), and were forked on average 1,085.4 times (max. 16,121). Considering this high number of forks, the vulnerabilities and bugs we found might have propagated on GitHub, and the 43

Table 3.6: GitHub projects missing bug/security fixes to code on Stack Overflow. Two projects (*) were retired months after we curated our GitHub projects sample; nonetheless, the maintainers were still notified.

Project	**Weakness**	**Language**	**Watch**	**Fork**	**Star**	**Contributors**
Odoo	Undefined Behaviour	JavaScript	1,494	16,121	24,881	1, 385
Wikimedia Commons App	CWE-404	Java	60	968	731	270
The Fuck	Others	Python	848	3,189	70,614	176
Apache NetBeans	CWE-404	Java	157	689	1,836	175
Amazon S3cmd	Others	Python	103	850	3,962	175
Open Event Frontend	CWE-1339	Python	22	1,694	2,188	151
CARTO	CWE-690	JavaScript	207	672	2,582	134
logback	CWE-404	Java	167	1,115	2,404	104
Cider	CWE-754 (2)	JavaScript	24	174	348	65
Apache Nutch	Undefined Behaviour	Java	240	1,199	2,356	46
JPEXS Flash Decompiler	Others	Java	187	534	3,128	33
Python Audio Library	CWE-754	Python	207	1,083	4,742	22
WordOps	CWE-772	Python	58	167	866	21
Eclipse N4JS	CWE-772	Java	11	25	26	18
Weevely	Undefined Behaviour	Python	132	548	2,596	18
Gmvauls	CWE-172	Python	80	272	3,429	16
RomRaider	CWE-172	Java	80	272	3,429	15
Kubebox	CWE-690	JavaScript	47	139	1,912	13
*AndroidTreeView	CWE-475	JavaScript	84	614	2,884	5
Chrome-Extensions	CWE-1339	JavaScript	80	483	894	5
Apache Lucene-Solr	CWE-611	Java	315	2,723	4,357	234
Android MoneyManagerEx	CWE-404	Java	49	167	327	17
Apache Fineract Android	CWE-404	Java	20	147	31	13
Faster-RCNN in Tensorflow	Others	Python	88	1,149	2,461	7
STATSD-C	CWE-120	C	6	13	75	6
Faster RCNN with PyTorch	Others	Python	52	464	1,609	3
Gnucash for Android	CWE-475	Java	101	525	1,143	46
AppScale GTS	Undefined Behaviour	Python	159	293	2,419	39
Eclipse Ceylon	Undefined Behaviour	Java	41	64	385	34
HowManyPeopleAreAround	Others	Python	165	383	6,687	12
WebRTC-Experiment	CWE-1339(2)	JavaScript	669	3,852	10,604	11
*Apache Chemistry	CWE-611	Java	11	60	46	6
Chinese OCR	Others	Python	93	1,077	2524	1
shadowsocks-libev	Others	C	14	900	87	81
OpenPGP for Android	Others	Java	31	79	237	59
Mirai BotNet	CWE-20, CWE-704	C	545	3,362	7,323	5
FastER RCNN	Others	Python	52	431	890	1
ngrok-libev	CWE-754, CWE-835, CWE-704	C	9	25	34	1
Layout Cast	Others	Python	71	185	1,711	5
Hashkill Password Recovery	CWE-20, CWE-754, CWE-835	C	23	55	183	3
Breeze HTTP Server	CWE-194	C	9	19	57	1
SassPHP	CWE-704	C	2	67	40	1
Tigger	CWE-704	C	2	6	40	6
Average			**157.4**	**1,085.4**	**4089.6**	**79.7**
Max			**1,494**	**16,121**	**70,614**	**1,385**



6 Added the necessary casts to `(unsigned char)` to avoid undefined behavior on `isspace()`

Source Link

edited Oct 8, 2016 at 12:19

[redacted] 111.4k ● 10 ● 107 ● 169

Inline | Side-by-side | Side-by-side Markdown

If you can modify the string:

```
// Note: This function returns a pointer to a substring of the or
// If the given string was allocated dynamically, the caller must
// that pointer with the returned value, since the original point
// deallocated using the same allocator with which it was allocat
// value must NOT be deallocated using free() etc.
char *trimwhitespace(char *str)
{
  char *end;

  // Trim leading space
  while(isspace(*str)) str++;

  if(*str == 0)  // All spaces?
    return str;

  // Trim trailing space
  end = str + strlen(str) - 1;
  while(end > str && isspace(*end)) end--;

  // Write new null terminator
  *(end+1) = 0;

  return str;
}
```

If you can modify the string:

```
// Note: This function returns a pointer to a substring of the or
// If the given string was allocated dynamically, the caller must
// that pointer with the returned value, since the original point
// deallocated using the same allocator with which it was allocat
// value must NOT be deallocated using free() etc.
char *trimwhitespace(char *str)
{
  char *end;

  // Trim leading space
  while(isspace((unsigned char)*str)) str++;

  if(*str == '\0')  // All spaces?
    return str;

  // Trim trailing space
  end = str + strlen(str) - 1;
  while(end > str && isspace((unsigned char)*end)) end--;

  // Write new null terminator
  *(end+1) = '\0';

  return str;
}
```

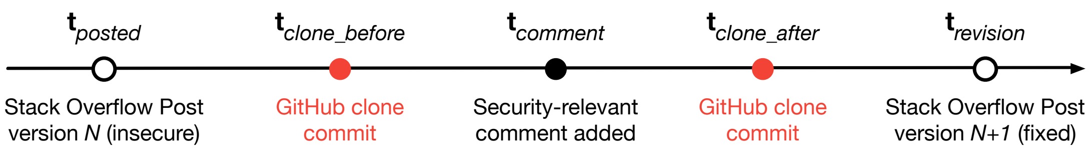
4 You have to cast the argument for `isspace` to `unsigned char`, otherwise you invoke undefined behavior. – [redacted] Sep 18, 2016 at 1:56

Figure 3.7: Answer `122721` containing a call to the *isspace* library function without the necessary cast to *unsigned char* (left-hand-side), which was pointed out in the comments (bottom) and subsequently fixed 20 days later (right-hand-side), as also noted in the commit message of the revision.

projects affected are likely a lower bound. Besides missing security fixes, we found another 20 posts containing fixes to improvements that were not reflected in 26 GitHub projects (see Appendix A.3).

> Though the number of posts we verified as true positives to contain a security weakness is low, we re-identified those snippets in a noticeable number of highly popular open-source projects with thousands of forks and stars. Given that we only studied non-forked projects, this popularity can amplify the impact of weaknesses in origin repositories.

Incident timelines: Based on the projects with missing security fixes, we attempt to understand potential developer behavior better. We base this analysis on the order



t_{posted}
t_{clone_before}
$t_{comment}$
t_{clone_after}
$t_{revision}$
Stack Overflow Post version *N* (insecure)
GitHub clone commit
Security-relevant comment added
GitHub clone commit
Stack Overflow Post version *N+1* (fixed)

Figure 3.8: Timeline of events between two versions of a post.

Table 3.7: Descriptive statistics for each event type measured in days. Margin of error for 95% confidence.

Events (in days)		Min	Median	Max	Average
All events ($N = 29$)					
t_{posted}	$\rightarrow t_{comment}$	270	1,394	3,639	1,394±338
$t_{comment}$	$\rightarrow t_{revision}$	0	20	1,516	296±200
Commit before comment ($N = 17$)					
t_{posted}	$\rightarrow t_{clone\_before}$	89	336	2,289	556±251
t_{clone_before}	$\rightarrow t_{comment}$	20	815	3,303	1,060±506
Commit after comment ($N = 12$)					
$t_{comment}$	$\rightarrow t_{clone\_after}$	9	224	1,221	495±236

of events when reusing code from Stack Overflow in GitHub projects and the intervals between those events. Figure 3.8 depicts the possible timeline of events when a developer commits a clone of an insecure code snippet to a GitHub project after it was posted on Stack Overflow (t_{posted}). The commit can happen either before a security-relevant comment is made in the Stack Overflow post (t_{clone_before}) or afterward (t_{clone_after}). As a consequence of this comment ($t_{comment}$), the code snippet is revised to fix the issue ($t_{revision}$). We constructed this timeline for all open-source projects missing security fixes. If multiple comments raise the same issue, we select the earliest comment to understand whether (or not) an indication of a security issue with the code snippet was present when the snippet was reused.

For 12 of the 26 posts with fixes, the security issue was only raised in the commit message of the revision. For the remaining 14 posts, the issues were raised in the comments, and we consider only those 14 posts at this point. Of those 14 posts, 15 distinct snippet versions were reused in 25 source files of 24 GitHub projects. We analyzed the timeline of 29 incidents where vulnerable code with a CWE was reused and a security-relevant comment was posted.

Table 3.7 summarizes our analysis results. The reused code was committed in 17 of the 25 cases before the first security-relevant comment appeared. On average, developers reused a code snippet ≈1.5 years (556 days) after it was posted on Stack Overflow, and a security-relevant comment appeared on average ≈2.9 years (1,060 days) after the commit. There could be various reasons for this long interval before a comment, e.g., initial unawareness about vulnerabilities or delayed comments by security-savvy users. Consequently, developers must track Stack Overflow threads with reused code for a long time to avoid missing relevant comments.

For the remaining 12 cases, a security-relevant comment was available when the code snippet was committed to GitHub, i.e., the developer could have been informed about the issue when adding vulnerable code to the project. On average, the comment was more than a year (495 days) present on Stack Overflow at the time of commit to GitHub.

Overall, we found that it takes, on average, almost four years (1,394 days) for a security-relevant comment to appear and, on average, nearly ten months (296 days) for a revision that fixes the security issue. However, we found three posts whose fixes occurred on the same day an issue was raised, and these posts affected three projects, with one of the posts (ID `779960`) containing three issues that were commented on and fixed on the same day.

Discussions on Stack Overflow contribute to the evolution of code on the platform. When those discussions have a security nature, they trigger fixes to bugs and vulnerabilities in code snippets. These findings show that prior works' treatment of Stack Overflow code as *static* limits the ability to build software tools to aid developers in reusing code from Stack Overflow more securely. We believe that if open-source contributors are equipped with a tool that consistently monitors Stack Overflow for security warnings and code fixes to reused code, those discussions and fixes can be brought timely to their notice. This will allow for fixes by the community to propagate faster to developer code bases.

3.4 Responsible Disclosure to Maintainers

We have disclosed the bugs and vulnerabilities we found to the maintainers of the affected projects. We used the dedicated security mailing lists to disclose the security issues found in Apache and Eclipse projects. For projects that do not define a specific policy, we directly emailed the main project contributor (i.e., the one with the most commits). We opened a public GitHub issue ticket for projects with only bugs (i.e., no security ramifications). We sent out 51 notifications, one for each bug/security issue we found, to the maintainers of the 43 affected projects. We received 40 responses (from 36 projects), with three indicating the project is no longer maintained and will not be updated. The remaining 33 projects responded and provided a fix while the maintainers of seven projects did not respond.

3.5 Discussion and Limitations

Similar to the security implications posed by outdated library versions on the security of production systems [10, 25, 63], outdated code snippets reused from Stack Overflow can have security implications for open-source projects. In our results, half of the code snippets reused in open-source projects were outdated (RQ1). We did not find evidence that developers updated the code snippets reused from Stack Overflow after the snippets evolved to newer versions (RQ2). While prior work has shown that developers reused vulnerable code snippets from Stack Overflow [31, 36, 116, 1, 49], our results additionally show that such vulnerabilities on Stack Overflow are being discussed and pointed out by the Stack Overflow community, which may consequently lead to a fix on Stack Overflow. However, those fixes did not propagate to the analyzed open-source projects on GitHub (RQ3). Looking at developers that attributed Stack Overflow as the source for copied code snippets, our results show that they are not immune to this problem: 13 (30%) of the 43 affected GitHub projects contain attributions to 6 (23% out of 26) Stack Overflow posts. However, despite attribution, developers were unaware that the reused snippets received security discussions and code fixes.

3.5.1 Security Classification

A big limitation in general for studying Stack Overflow and our work is the lack of tools/methods for detecting vulnerable code snippets *in general*. Dicos [49] is a good step in the right direction. However, as we discovered, it is still too narrow in its language support and scope of vulnerabilities to be suitable as an automated classifier for our initially large data set. Consequently, in our methodology, we reduced the problem space by focusing on *outdated* and *filtered* code snippets, whose total number is feasible for manual verification. Thus, our classification trades large scaling for higher reliability and versatility. Due to this necessary trade-off, our methodology might miss security-relevant discussions in other threads on Stack Overflow that were filtered out. For example, in our study, the filter pipeline discarded 569 *potentially* security-relevant posts reused in 9,971 file sections (11.79% of 84,542). For 4,335 of those file sections, the pipeline selected another security-relevant post as the best candidate. For the remaining 5,636 file sections, the filter pipeline selected a *non*-security-relevant post and filtered out 327 distinct potentially security-relevant posts. We manually confirmed 202 of the 327 posts as not security-relevant, i.e., the pipeline did not miss a security issue discussed in those Stack Overflow posts. The filter pipeline selected 9 (of the remaining 125) true positive security-relevant posts as the best candidate for at least one other file section but always favored a non-security-relevant post over 116 of the security-relevant posts reused in 1,112 (1.3%) file sections, for which we hence *may* have missed a security issue. We conducted a Chi-Square (χ^2) test to understand whether the filter pipeline favors non-security-relevant posts over potentially security-relevant posts. Table 3.8 shows the observed and expected values based on the number of posts that are input to filter F3 and the output of filter F7. We excluded filters F1 (Attribution) and F2 (Commit Date) to focus on potential security-relevant posts discarded by the heuristics-based part of the filter pipeline. The results indicates that the filter pipeline tends to favor non-security-relevant posts over security-relevant posts ((χ^2, df=1, N=6407) = 11.0674, $p < 0.001$), which might lead to under-reporting the number of projects with missing security fixes. However, the corresponding effect size (Cramer's V = 0.042) can be considered negligible and should have very little impact on our results. Thus, a refined approach without a filter pipeline might find more GitHub projects with outdated, insecure code snippets from Stack Overflow than the 43 projects we discovered. Still, our results demonstrate that the problem of *missing security fixes* exists and requires further attention.

The security filter identified 505 posts with security-relevant keywords (see Section 3.3.3), of which 238 were manually confirmed true positives and 267 confirmed false positives. This means the security filter has a ≈50% chance of correctly classifying a post as security-relevant. In addition, we manually verified the remaining 1,785 outdated posts not matching the keywords of the security filter and found four with actual security-relevant edits (i.e., 0.22% false negatives). This means the security filter misses very few but over-reports security-relevant posts, necessitating additional manual verification work of true positives. Moreover, like prior work [49], our context-based approach to determine security and bug fixes in Stack Overflow change histories is limited because commit messages on Stack Overflow are optional, and we relied on the quality of discussions by Stack Overflow users. Consequently, when there are no discussions

	Not Flagged	**Flagged**	**Total**
Not Filtered Out	3,267 (3213.02)	974 (1027.98)	4,241
Filtered Out	1,587 (1640.98)	579 (525.02)	2,166
Total	**4,854**	**1,553**	**6,407**

Table 3.8: Cross table between posts **filtered** vs. **flagged** by our security filter, with the observed values and the expected values.

about insecurities or commit messages mentioning a fix, our approach considers those cases secure by default.

Lastly, to ensure a high external validity of our results and minimal false positives, we used a conservative approach to select the most probable code snippet clone in our filter pipeline, fine-tuned and selected the minimum similarity threshold for NiCad, and relied on the fine-tuned parameters from Baltes et al. [13] to detect exact clones with PMD CPD. As a result, differences in the false negative rates of both clone detectors might lead to underestimating the number of outdated file sections for some of the programming languages more than for others.

Despite the trade-off in data set size and the limitations in clone detection and snippet classification, we discovered several popular GitHub projects in different programming languages affected by various security issues in outdated reused code. Still, those results should be seen as a lower bound to the problem of insecure, reused, and outdated Stack Overflow snippets on GitHub.

3.5.2 Origins of Common Code

In this work, we studied how code reused between Stack Overflow and GitHub evolves. To create a feasible data set of snippets for our study, the methodology in this paper assumes code provenance from Stack Overflow posts to GitHub projects, i.e., that GitHub developers copy code from Stack Overflow. Hence, when a code fragment appears on Stack Overflow and GitHub, we focused on code snippets that appear *first* on Stack Overflow before their GitHub counterparts. However, a code snippet on both GitHub and Stack Overflow could have been independently copied from a third-party source. For example, prior work [76, 15, 71, 81] has shown that some code snippets on Stack Overflow are copied over from tutorials, from GitHub projects, or other software collections (e.g., Qualitas). The 43 affected GitHub projects in our results did not contain an attribution to external sources other than Stack Overflow.

Considering the rich discussions on Stack Overflow, which we also leveraged for our security classification of outdated snippets, we think it very promising for future work to bridge the gap between these two knowledge-sharing ecosystems [71] and support developers with insights from Stack Overflow whenever they use code that is present on Stack Overflow *irrespective of whether the code was copied from Stack Overflow* or not. This could further enhance tool support for developers, as discussed earlier. For our current security-oriented study, this was impossible considering the sheer volume of this code overlap (see Section 3.2.2), and we focused on outdated snippets that might have been copied from Stack Overflow to GitHub.

3.6 Related Works

Stack Overflow has been found to be a valuable source for studying developers and different recent works on usable security involve Stack Overflow. We provide a brief overview of related works that studied code on Stack Overflow or assessed the problem of outdated third-party libraries, which carries a conceptual resemblance to the problem of using outdated code from Stack Overflow.

3.6.1 Stack Overflow

3.6.1.1 Analyzing Vulnerabilities on *SO*

Nadi et al. [74] reported that many developers consult *SO* for help when implementing solutions requiring the usage of cryptographic APIs, because developers struggle to understand how the APIs are designed to work. The study examined 100 *SO* posts with snippets utilizing crypto APIs and surveyed 48 developers to understand the obstacles they face when using crypto APIs. Similarly, Rahman [83] showed the challenges that developers face while implementing security features. Like Nadi et al., they concluded that developers that copy-paste code mostly reuse insecure usages of crypto APIs, because they do not understand how the APIs should be used in a way that is safe.

Fahl et al. [30, 29] did a study on developers that consulted *SO* for help and found a couple of high profile applications that are vulnerable to Man-In-The-Middle attacks because developers copy *SO* code examples that specifically disable TLS functionality. They also found high profile Android apps that do not properly store user credentials, credit card numbers, and other sensitive user private data.

Acar et al. [116] studied which resources developers consult when they are faced with a security-related problem. To this end, they conducted a lab study with 54 developers and provided them with different information sources to consult while implementing a security feature. They found that developers that consulted Stack Overflow presented a more functionally correct but insecure solution than those that consulted other resources. Their conclusion is that developers that use books and official documentations are more likely to write secure code while those that consulted Stack Overflow are more prone to writing insecure code.

3.6.1.2 Detecting Vulnerable *SO* snippets

Different works investigated the extent to which insecure code examples find their way into production code. Zhang et al. [119] did a study of security API misuse patterns and found that a number of *SO* code examples containing improper API usages were reused in open source projects. Verdi et al. [110] manually classified 69 known CVE related vulnerabilities on *SO* and found usages of vulnerable snippets in over $2,850$ projects on GitHub. Abdalkareem et al. [1] and Fischer et al. [31] independently investigated the amount of insecure code samples on *SO* that resurfaced in high profile Android apps published on Google Play. Abdalkareem et al. studied the reuse of *SO* code snippets in

Android apps and found that the potential of introducing bugs in apps is higher when code is reused from *SO*. Fischer et al. used a learning-based approach to classify code snippets into secure or insecure depending on the usage of crypto APIs in Java. They discovered that about 30% of the insecure code samples have been reused in security critical apps published on Google Play.

3.6.1.3 Tooling Support

Chen et al. [21] have shown that the popularity of *SO* posts cannot be used as a way to judge the insecurity of code snippets in a post, since both secure and insecure post were both rated very highly. Therefore, a way is needed to better support developers in making secure programming choices. Fischer et al. [36] added a warning to posts containing insecure code snippets. When developers consult a post that contains insecure code, a warning together with a detailed description of the security risk is presented. Developers have a choice to either copy the insecure code or consult a list of recommended posts that perform a similar functionality but contain secure code. Verdi et al. [110] implemented a similar approach by developing a browser extension capable of detecting known vulnerabilities in C++ code snippets.The recent *Dicos* paper by Hong et al. [49] is closest to our work. Dicos detects security fixes in the change histories of Stack Overflow posts by searching for security-relevant keywords in natural language texts and scanning for security-relevant changes to a snippet's control flow or security-sensitive APIs. Applying Dicos to 668k posts tagged with C, C++, or Android, the authors found 12k insecure posts. Using clone detection on 2k popular C/C++ open-source projects, the authors could further detect insecure snippets in 151 projects. While Dicos' results relate to our RQ3, i.e., if outdated insecure snippets are found in GitHub projects, Dicos' implementation is limited (e.g., it only compares the initial and the last version in the change history of code snippets) and not directly applicable for our methodology (e.g., insecure snippets must contain certain types of edits idiosyncratic to selected programming languages). In Section 3.2.5, we discussed the applicability of Dicos for our methodology and pointed out the differences with our work.

While all the works discussed above made crucial contributions in pointing out the problem of vulnerable code snippets on Stack Overflow, developed tools to help developers safely reuse code snippets on Stack Overflow, and studied the propagation of code snippets from Stack Overflow to the code bases of software developers, none of them considered the code snippets on Stack Overflow as "evolving code" or studied the impact of that evolution on software security.

3.6.1.4 Analyzing the Revison History of Stack Overflow

Zhang et al. [118] scanned 650k C/C++ snippets from Stack Overflow for 89 CWEs (Common Weakness Enumeration) and discovered 13k vulnerable snippets. The authors also investigated the code evolution history of C/C++ snippets and found that, in general, code revisions are associated with reducing the number of code weaknesses.

Stack Overflow and GitHub have also been the subjects of measurement and developer behavior studies, e.g., [115, 57, 40, 26, 3, 93, 14, 76, 13]. Closest to our approach, Manes and Baysal [71] compared the change histories of 23k GitHub projects and 4.6k Stack Overflow code snippets that were explicitly *attributed*. They found that reused snippets develop independently and that Stack Overflow snippets primarily evolve in their text blocks. The authors did not use code clone detection to determine if attribution was accompanied by copied code and relied entirely on time series analysis like impact latencies. They left the application of code clone detection open for future work. Further, Baltes et al. [15] created the *SOTorrent* dataset, which we also use in our work (see Chapter 3). Based on this dataset, the authors showed that code on Stack Overflow is maintained and does evolve.

3.6.2 Third-party Libraries

Outdated third-party libraries have been shown to have security implications for apps that rely on them [10, 63, 25]. Derr et al. [10] built a technique that detects third-party libraries used in Android apps in order to understand the security implications of apps using outdated library versions. They found that vulnerabilities stayed longer in apps because app developers were slow in updating outdated library versions. In particular, a vulnerability due to crypto API misuse in external libraries affected 296 top apps deployed to 3.7bn devices. Lauinger et al. [63] investigated the usage of JavaScript libraries in web apps and found that web developers do not react on time to update library versions to the most recent bug fix release and, as a result, 37% of 133K web apps included at least one library with a known vulnerability. They found transitive libraries to pose more security risks since those libraries are more likely to contain vulnerabilities that attackers can exploit. In a follow-up work, Derr et al. [25] analysed about 1 million apps and found that 85.6% of the outdated libraries used in the apps could have simply been avoided had the developers simply change the library version bundled in their Android apps, without even modifying any app code. The authors also interviewed developers and discovered that many developers refrained from updates due to fear of incompatibilities, lack of necessity, or being unaware of updates.

By copying code from *SO*, developers create a dependency similar to including a software library. Thus, there exist strong conceptual similarities, however, no work has studied if the same problems as with outdated libraries also exist for outdated code snippets. In contrast to library detection, we face different challenges: First, we need to re-identify very small code pieces with just a handful lines of code in contrast to complete software libraries. Second, in contrast to libraries, there are no systems like Common Vulnerabilities and Exposures (CVE) to clearly report vulnerable code snippet versions, but instead we have to rely on the snippets and their surrounding *SO* threads (e.g., comments) to classify snippets, similar to Dicos [49].

3.7 Conclusion

We measured the impact of Stack Overflow code artifact evolution on software projects that rely on specific versions of code snippets reused from Stack Overflow. The presence of duplicate code on Stack Overflow made it non-trivial to pinpoint an exact snippet version reused in a section of a source file in an open-source project. This prompted us to build a filter pipeline to reduce the number of code snippet versions reported as clones in a single source file section. By analyzing the timeline of reused code snippets, we found that 51% of code snippets reused from Stack Overflow were outdated, which affected 2,109 open-source projects. At the same time, we found no evidence that developers update reused versions of Stack Overflow code snippets, regardless of whether the snippets underwent bug or security fixes on Stack Overflow. Finally, by examining comments and commit messages of Stack Overflow posts, we found fixes to 15 security issues that are missing in 43 GitHub projects, with an additional 20 general code improvements missing in 26 GitHub projects. Considering the popularity of the 43 projects missing security fixes, we believe this number to be the lower bound of the affected code bases. Our findings call for action to provide developers with tools to track Stack Overflow long-term for security-focused discussions and code fixes to reused code.

4
Effects of Evolution on Researchers

In the previous chapter, we discussed and presented our findings with regards to how the evolution of code snippets on Stack Overflow affects developers who reuse them but did not monitor Stack Overflow for updates. We found that 2,405 code snippet versions reused in 22,735 distinct GitHub source files were outdated, affecting 2,109 GitHub projects. Among those outdated snippet versions, we found 26 to have a security-relevant update on Stack Overflow that fixes a known vulnerability. The fixes to those vulnerabilities on Stack Overflow were not reflected in 43 highly popular, non-forked open-source projects to whose maintainers we disclosed our findings.

In this chapter, we focus on researchers and conduct a meta-research study of prior research works that investigated the security properties of code snippets on Stack Overflow. In recent years, several security-focused studies [116, 31, 36, 49, 118, 32, 90, 110, 82, 71, 72, 81, 1, 21] have examined Stack Overflow to analyze the security of shared code on the platform, develop tools for secure code reuse, or use it as a proxy for studying developer behavior. This research is fostered by the quarterly releases of a dataset containing all content created on the Stack Exchange Inc. platform since its launch in 2008. Like developers, security researchers also consider only the current version of the Stack Overflow data set when studying its content. Transferring the lessons learned about Stack Overflow code evolution from the previous Chapter 3 to researchers raises questions about how this evolution affects research findings based on particular dataset versions, the long-term validity of these findings, and what lessons can be learned for future studies using Stack Overflow data. This chapter discusses how we addressed the following meta-research questions:

MQ1: *Which aspects of Stack Overflow affect the validity of prior research?*

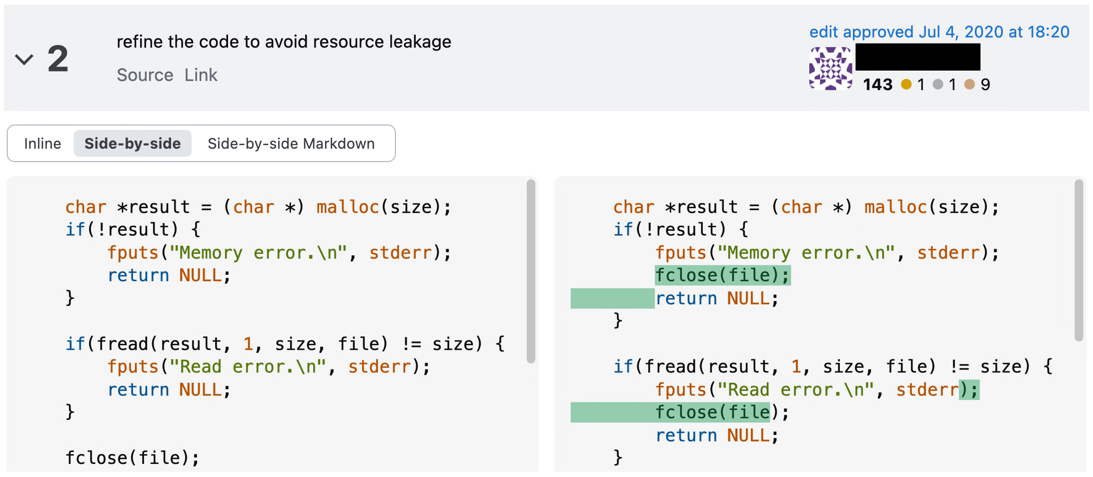
MQ2: *How much do Stack Overflow code snippets and surrounding context evolve?*

MQ3: *How is the validity of prior research impacted by evolution on Stack Overflow?*

The remainder of this chapter is organized as follows: In Section 4.1, we present a real-world example from Zhang et al. [118] to illustrate how the evolution of code snippets can influence research findings. Section 4.2 outlines our systematization of prior studies that investigated the security properties of Stack Overflow code snippets, while Section 4.3 provides a full-text review of the selected studies. Section 4.4 presents a time series analysis to explore how code snippets and discussions related to security and privacy evolve on Stack Overflow. In Section 4.5, we describe six replicated studies using a more recent dataset to assess whether code evolution affects research findings. Finally, Section 4.6 discusses the limitations of our work, and Section 4.8 concludes the chapter.

4.1 Motivating Example

Figure 4.1 depicts an example code snippet [103] from the results by Zhang et al. [118], who used the December 2018 version of the SOTorrent dataset to study whether code revisions improve security. They identified two CWEs in this code: two instances of



```
char *result = (char *) malloc(size);
if(!result) {
    fputs("Memory error.\n", stderr);
    return NULL;
}

if(fread(result, 1, size, file) != size) {
    fputs("Read error.\n", stderr);
    return NULL;
}

fclose(file);
```

```
char *result = (char *) malloc(size);
if(!result) {
    fputs("Memory error.\n", stderr);
    fclose(file);
    return NULL;
}

if(fread(result, 1, size, file) != size) {
    fputs("Read error.\n", stderr);
    fclose(file);
    return NULL;
}
```

Figure 4.1: First version of the answer (left-hand side) labeled by the authors as insecure and unmodified with three CWE instances. The 2nd version of the answer on the right-hand side shows both instances of CWE-775 fixed on July 4th, changing the snippet's status to *improved*.

CWE-775 and one instance of CWE-401, all introduced in the first version in January 2013. By December 2018, this snippet had never been revised, leaving these security issues unresolved. The authors, therefore, labeled the snippet as insecure and concluded that the snippet *had never been revised* to address the three CWE instances. By July 2020, this conclusion was obsolete, as the snippet had been revised to fix both instances of CWE-775. This revision, posted after the authors sampled their data, reduced the CWE instances from three to one, changing the snippet's status to *improved*.

The example shows that a code snippet can undergo edits throughout its lifespan, causing its security status to change over time. Consequently, code snippets might be insecure at a given time but secure at a future time or vice-versa. Therefore, it seems prima facie intuitive for researchers studying Stack Overflow snippets to conduct measurements at multiple points in time using different dataset versions to account for these fluctuations. Integrating these evolving trends and changes in snippets may enhance the robustness of research studies.

4.2 Systematization of Relevant Works

To understand how prior work may be affected by Stack Overflow evolution (**MQ1**), we systematize studies that investigated the security properties of Stack Overflow code snippets. First, we systematically surveyed the literature for relevant works (Section 4.2.1). Following this, we developed a taxonomy of the methodologies used by these studies to analyze Stack Overflow datasets (Section 4.2.2). Two researchers conducted this systematization and decided on the criteria for the taxonomy.

4.2.1 Literature Search

We conducted a systematic literature review following the guidelines of Kitchenham and Charters [59]. Below, we outline the criteria we apply to assess whether a study meets the necessary requirements for inclusion. These criteria are designed to ensure that only relevant studies are considered.

Inclusion and Exclusion Criteria. A study is included if it meets all our inclusion criteria (*IC*) and is excluded if it meets any one of the exclusion criteria (*EC*) outline below:

IC1. The study must focus on the Stack Overflow website.

IC2. The study must examine code snippets on Stack Overflow.

IC3. The study must analyze the security of code snippets or identify and address bugs or faults in code snippets.

IC4. The study must be published between 01/2005 and 12/2023.

IC5. The study is a journal article or in conference proceedings that are DBLP-indexed.

EC1. The study focuses only on code reuse from Stack Overflow without studying the security of or identifying bugs or faults in snippets.

EC2. It is published outside the date range in *EC3*.

EC4. is a systematic review or meta-analysis, as only primary studies are considered.

EC.5 It is a Non-peer-reviewed study (e.g., technical reports).

Search Strategy. Figure 4.2 depicts the flow of our systematic literature review. We manually reviewed the titles and abstracts of studies listed on the dedicated Meta Exchange thread [95] listing academic papers that use Stack Exchange datasets. This yielded one relevant paper. Next, we retrieved all studies that cited the SOTorrent dataset [15], a widely used dataset for studying Stack Overflow. Of these 150 studies, 128 fit the inclusion criteria. Following the recommendations by Kitchenham and Charters [59], we automated part of our literature review process. We employed OpenAI's GPT4o model to scan abstracts and identify studies focusing on the security of Stack Overflow code snippets. A preliminary evaluation of GPT4o showed that it performs well in classifying studies based on our inclusion and exclusion criteria (details see Appendix B.1.1). We use GPT4o to efficiently screen non-relevant studies based on a better contextual understanding of security concepts than keyword-based filtering. Papers identified as relevant by GPT4o are manually verified through full-text analysis. With the help of GPT4o, we identified six additional relevant studies among the 128 candidates. Based on the reviewed studies, we created a list of search terms to identify

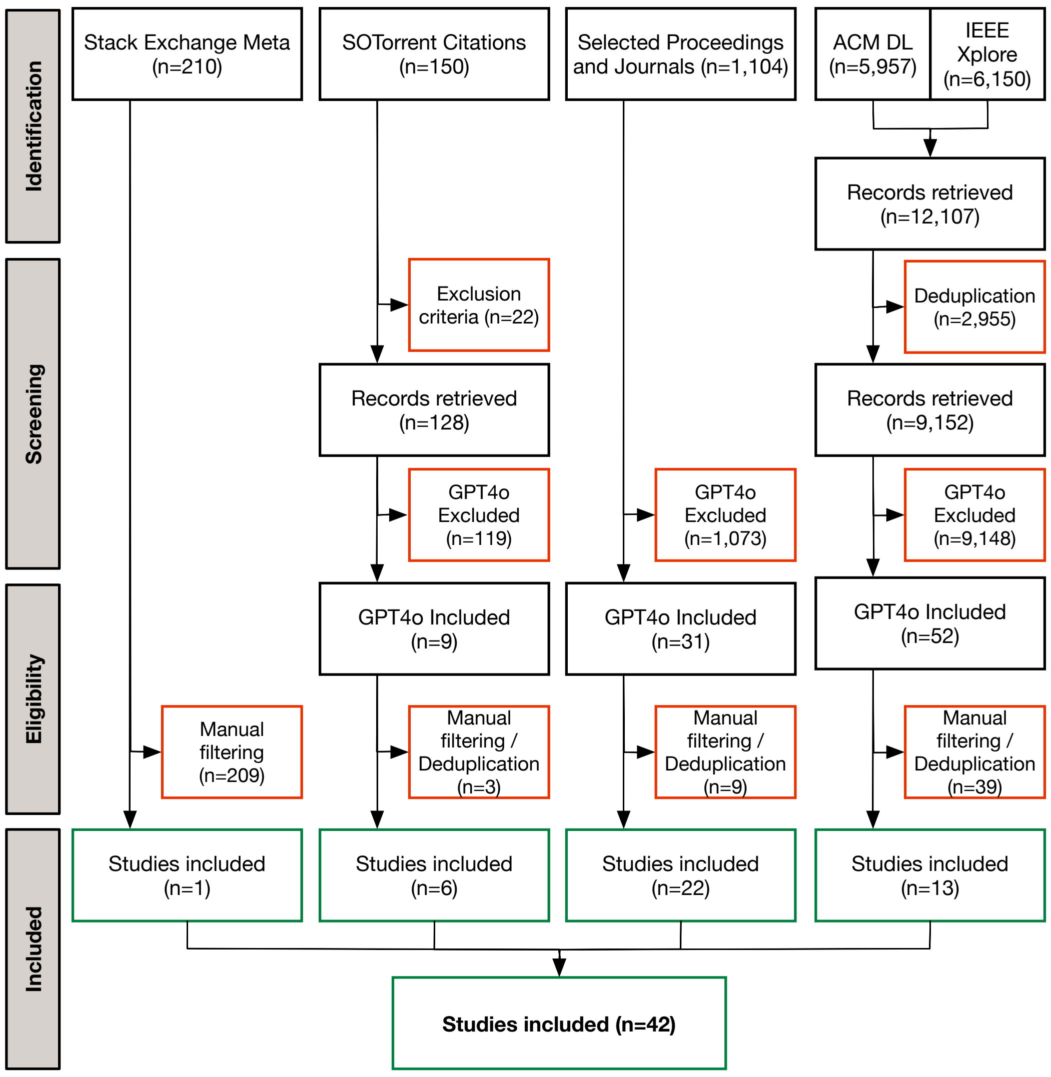


Figure 4.2: PRISMA diagram of our literature review

all research studies related to Stack Overflow, irrespective of whether they analyzed code snippets or investigated their security.

Listing 4.1 shows the search terms we used to identify research studies that investigated Stack Overflow. To identify studies containing the keywords in their titles or abstracts, we normalized all keywords, titles, and abstracts to lowercase before performing the search. For example, "Stack Overflow" or "StackOverflow" was normalized to "stack overflow" and "stackoverflow" to ensure consistent matching. We used these terms to search for studies in the proceedings of 29 conferences and the volumes of 3 journals. Table 4.1 lists the venues whose proceedings and volumes we search for candidate papers that study Stack Overflow using the search terms shown in Listing 4.1.

We identified 1,104 matching studies across all venues. Using GPT-4o, we analyzed their abstracts, resulting in 22 confirmed relevant studies. We conducted additional searches in IEEE Xplore and ACM Digital Library using the same search terms. After

Listing 4.1: Search terms for literature search

```
"stackoverflow" OR "stack overflow" OR "crowd knowledge" OR
"crowdsource knowledge" OR "crowd—source knowledge" OR
"Q&A websites" OR "Q&A sites" OR "social Q&A websites" OR
"online Q&A communities" OR "community question answering" OR
"knowledge sharing" OR "crowdsourcing" OR "knowledge—sharing" OR
"Q&A Forums" OR "Online Code Snippets"
```

Table 4.1: Academic Venues Surveyed During the Systematic Literature Review.

Venue	Cat.	Type
ACM Conference on Computer and Communications Security (CCS)	SP	C
IEEE Symposium on Security and Privacy (SP)	SP	C
USENIX Security Symposium	SP	C
Network and Distributed System Security Symposium (NDSS)	SP	C
Annual Computer Security Applications Conference (ACSAC)	SP	C
ACM ASIA Conference on Computer and Communications Security	SP	C
IEEE European Symposium on Security and Privacy	SP	C
Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)	HCI	C
ACM Conference on Human Factors in Computing Systems (CHI)	HCI	C
ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW)	HCI	C
ACM Symposium on User Interface Software and Technology (UIST)	SE	C
ACM SIGSOFT Conference on the Foundations of Software Engineering (FSE)	SE	C
International Conference on Software Engineering (ICSE)	SE	C
International Conference on Automated Software Engineering (ASE)	SE	C
IEEE International Conference on Software Maintenance (ICSM)	SE	C
International Symposium on Software Reliability Engineering (ISSRE)	SE	C
International Conference on Software Testing, Verification and Validation (ICST)	SE	C
International Conference on Program Comprehension (ICPC)	SE	C
International Conference on Mining Software Repositories (MSR)	SE	C
Empirical Software Engineering and Measurement (ESEM)	SE	C
International Conference on Software Analysis, Evolution, and Reengineering (SANER)	SE	C
European Software Engineering Conference (ESEC)	SE	C
IEEE International Conference on Software Maintenance and Evolution (ICSME)	SE	C
IEEE Conference on Reverse Engineering (WCRE)	SE	C
International Symposium on Software Testing and Analysis (ISSTA)	SE	C
Fundamental Approaches to Software Engineering (FASE)	SE	C
IEEE International Conference on Program Comprehension (ICPC)	SE	C
The Web Conference (WWW)	SE	J
IEEE Transactions on Software Engineering (TSE)	SE	J
Empirical Software Engineering (EMSE)	SE	J

C = Conference; J = Journal

deduplication, this resulted in 9,152 unique publications, which we retrieved from the two systems. Using GPT-4o, we confirmed an additional 13 studies that met our criteria, totaling 42 relevant studies.

4.2.2 Comparison Criteria

Our literature search yielded 42 relevant studies. We reviewed the full text of each study and its artifacts to develop criteria for comparison of different works. We found nine criteria, explained in the following, and Table 4.2 compares the 42 considered studies based on these criteria. Criteria **D1**–**D5** define dependencies on Stack Overflow data (used in Section 4.3), where criteria **R1**–**R4** are relevant for our replication studies (in Section 4.5).

D1. Programming Languages: Lists the programming language(s) of code snippets investigated in a study. If a study selects snippets independent of language, we assign ✪.

Table 4.2: Comparison of security-focused studies on Stack Overflow. We replicated the highlighted studies.

		Dataset Snapshot	D1. Prog. Languages	D2. Code Scanners	D3. Code Evolution	D4. Surrounding Context	D5. Sample Size Filter	R1. Artifact Availability	R2. Language Detection	R3. Code Reuse Detect.	R4. Human-Centered
Zhang et al.	[118]	12/2018	C	✔	✔	✘	✔	✘	✪, M	✘	✘
Hong et al.	[49]	12/2020	C	✪, N	✔	✔	✔	○	✔	✪, S	✘
Fischer et al.	[36]	03/2018	J	✪, M	✘	✘	✔	◐	N/A	✘	✘
Fischer et al.	[31]	03/2016	J	✪, M	✘	✘	✔	◐	N/A	✪, P	✘
Rahman et al.	[82]	12/2018	P	✪, SM	✘	✘	✔	◑	✔	✘	✘
Campos et al.	[32]	12/2018	JS	✔	✘	✘	✔	✪	✔	✔	✘
Verdi et al.	[110]	09/2018	C	✘	✘	✘	✔	◐	✔	✔	✘
Selvaraj et al.	[90]	01/2022	C	✔	✔	✘	✔	◐	✪, M	✘	✘
Acar et al.	[116]	10/2015	J	✘	✘	✘	✘	✘	N/A	✘	✔
Chen et al.	[21]	?/2018	J	✘	✘	✘	✔	✘	N/A	✘	✘
Meng et al.	[72]	08/2017	J	✘	✘	✔	✔	◐	✔	✘	✘
Ragkhitwets	[81]	01/2016	J	✘	✔	✘	✔	◐	✔	✪,SI,CC	✔
Bai et al.	[12]	N/A	J	✘	✘	✘	✔	✘	N/A	✪,M	✔
Bagherzadeh et al.	[11]	N/A	J,S	✘	✘	✔	✔	◐	✔	✘	✘
Chen et al.	[20]	?/2018	J	✪,M	✘	✘	✔	✘	N/A	✘	✘
Zhang et al.	[119]	10/2016	J	✪,P	✘	✘	✔	✘	✔	✔,CC	✘
Rahman et al.	[84]	08/2021	J	✘	✘	✘	✔	✘	✔	✔,CC	✔
Reinhardt et al.	[85]	N/A	J	✔	✘	✘	✔	✘	✔	✔,CC	✘
Licorish et al.	[64]	?/2016	J	✔	✘	✔	✘	✘	✔	✘	✘
Schmidt et al.	[88]	03/2022	JS,P	✘	✘	✔	✘	◑	✔	✘	✘
Yi Liu et al.	[66]	N/A	J	✪,M	✘	✔	✘	✘	✔	✘	✘
Ren et al.	[86]	03/2019	J	✪,M	✘	✔	✔	✘	✔	✘	✔
Licorish et al.	[65]	?/2016	J	✔	✘	✘	✔	✘	✔	✘	✘
Rangeet Pan	[77]	N/A	P	✘	✘	✔	✔	✘	✔	✘	✘
Ye et al.	[117]	N/A	J	✪,M	✘	✘	✔	✘	✔	✘	✘
Chen et al.	[19]	N/A	J	✪,M	✘	✔	✔	✘	✔	✔,CX	✔
Zhang et al.	[120]	N/A	P	✘	✘	✘	✔	◑	N/A	✘	✘
Alhanahnah et al.	[5]	N/A	J	✔	✘	✔	✔	◑	N/A	✘	✘
Imai et al.	[50]	N/A	J	✔	✘	✘	✘	✘	✪,R	✪,Se	✘
Fischer et al.	[35]	03/2018	J	✪,M	✘	✘	✔	✘	N/A	✘	✔
Almeida et al.	[6]	N/A	JS	✔	✘	✘	✔	✘	N/A	✘	✘
Islam et al.	[53]	N/A	P	✘	✘	✘	✔	◑	N/A	✘	✘
Mahajan et al.	[69]	03/2019	J	✪,A	✘	✘	✔	◑	✔	✘	✔
Mahajan et al.	[70]	03/2019	J	✔	✘	✘	✔	◑	N/A	✘	✔
Yadavally et al.	[114]	N/A	J,C	✘	✘	✘	✔	◑	✪,S	✘	✘
Firouzi et al.	[34]	09/2018	C#	✘	✘	✘	✔	✘	✔	✘	✘
Ghanbari et al.	[39]	09/2018	P	✪,F	✘	✘	✔	✘	✔	✘	✘
Gao et al.	[38]	N/A	J	✔	✘	✘	✔	✘	N/A	✘	✘
M. Chakraborty	[18]	01/2021	P	✘	✘	✘	✔	✘	N/A	✘	✘
Moghadam et al.	[73]	N/A	J	✘	✘	✘	✔	✘	✔	✘	✘
Madsen et al.	[68]	N/A	JS	✪,S	✘	✔	✔	✘	✔	✘	✘
Jhoo et al.	[56]	N/A	P	✪,S	✘	✘	✔	✘	✔	✘	✘

D1: J = Java, C = C/C++, JS = JavaScript, P = Python, S = Scala, P = PHP
D2: SM = String matching, M = Machine learning, N = NLP, R = Regular Expression, A = Abstract Program Graph, P = Pattern Recognition, F = Fault Localization, S = Static analysis
R2: M = Machine Learning, R = Regular Expression, S = Static analysis
R3: S = SourcererCC, SI = Simian, CC = CCFinder, CX = CCFinderX, P = PDG, Se = SeByte, M = MOSS

D2. Code Scanners: *Off-the-shelve* scanners (✔) or *custom* code classifiers (✪) can be used to find security weaknesses or general code quality issues in code snippets. For custom solutions, we list the techniques for finding security flaws, e.g., static analysis, machine learning, or graph query analysis. We assign ✘ for studies detecting security issues manually.

D3. Code Evolution: Indicates whether a study considered (✔) code evolution in their methodology or not (✘).

D4. Surrounding Context: Comments and post descriptions are natural language text that provides context around code snippets. We assign ✔ if a paper utilizes surrounding context to classify code snippets (e.g., using NLP); otherwise, ✘.

D5. Sample Size Filter: Studies may exclude code snippets based on specific criteria. For example, a study examining cryptographic API usage in Java might filter out snippets that do not use those APIs, or a study focusing on the effects of code revisions might exclude single-version snippets. We assign ✘ if a study includes all snippets without filtering; otherwise, we assign ✔.

R1. Artifact Availability: Code and data artifacts are important for revisiting and replicating prior findings; here, specifically, they are used to compare results on different versions of Stack Overflow datasets without undergoing the tedious (and error-prone) effort of re-implementing prior approaches. If paper artifacts (either code, data, or both) are unavailable, we assign ✘. We distinguish between artifacts that are *fully* or *partially* available. An artifact is *fully functional available* (✪) if both data and code are available and, most importantly, the code is free of bugs and can be used directly without changes to the code base. We assign ● if the code of a fully available artifact has bugs that require significant effort to fix. An artifact is *partially* available if only code (◑) or only data (◐) is available. If the code of a partially available artifact requires significant bug fixing, we assign ○.

R2. Language Detection: The Stack Exchange and SOTorrent datasets do not state the programming languages of code snippets, requiring researchers to determine these. Researchers can rely on other data (✔) like *tags* of posts, or they can employ code analysis tools (✪). When a tool is used, we indicate the underlying technique (e.g., machine learning or static analysis) used for language detection. If the authors did not mention their detection technique, we assign **N/A**.

R3. Code Reuse Detection: Several studies indicate that developers reuse code from Stack Overflow, sometimes attributing the copied code snippets with their URL [13]. This criterion differentiates between studies using attribution (✔) and studies using tools (✪), e.g., clone detection, to identify code reuse. If a tool is used, we give the name of the tool or technique. We assign ✘ if code reuse is not considered.

R4. Human-Centered Research: Differentiates human-centered studies of Stack Overflow data involving software developers as participants. If the study's methodology includes developer participation, we assign ✔; otherwise, we assign a ✘ for purely data-driven studies.

4.3 Relevance of Stack Overflow Evolution

Table 4.2 lists the relevant works from our literature review. In the following, we elaborate on the criteria for these papers (**D1**–**D5**) and how these can be affected by content evolution on Stack Overflow.

The work by Zhang et al. [118] investigated whether code revisions on Stack Overflow are associated with improving or worsening the security of code snippets (**D3**: ✔). The study focused on *C* and *C++* code snippets (**D1**: C/C++) with at least 5 LoC (**D5**: ✔). The *CppCheck* static analysis tool was used to detect security weaknesses (**D2**: ✔, **D4**: ✘).

The *DICOS* tool by Hong et al. [49] analyzes the change history of C/C++ code snippets (**D1**: C/C++; **D3**: ✔) to discover insecure code snippets by looking for changes in security-sensitive APIs, control flow information, and/or security-related keywords in the surrounding context (**D2**: ✪, N; **D4**: ✔). However, the authors only considered the first and last revision of each snippet (**D5**: ✔).

Fischer et al. [31] investigated the reuse of insecure Java code snippets in Android apps (**D1**: Java). They employed a machine learning classifier to identify snippets with insecure usage of crypto APIs (**D2**: ✪, M; **D5**: ✔). The authors considered neither code evolution (**D3**: ✘) nor surrounding context in their methodology (**D4**: ✘).

Follow-up work by Fischer et al. [36] relied on flagging Java snippets with crypto API misuse and suggesting more secure snippets (**D1**: Java; **D2**: ✪, D; **D5**: ✔). As in their preceding work, the authors considered neither code evolution (**D3**: ✘) nor context in their methodology (**D4**: ✘).

Rahman et al. [82] analyzed Python code snippets (**D1**: Python) to identify insecure coding practices using string matching (**D2**: ✪, SM). They focused on snippets from answers attributed in GitHub project (**D5**: ✔) but did not consider code evolution or context (**D3:** ✘, **D4**: ✘).

Campos et al. [32] studied JavaScript code snippets (**D1**: JavaScript) to identify rule violations using ESLint, a JavaScript linter (**D2:** ✔). They relied exclusively on ESLint to detect security or code quality issues without considering the surrounding context (**D4**: ✘) or the evolution of snippets (**D3**: ✘). Further, they focused only on snippets with a minimum of 10 lines of code (**D5**: ✔).

Verdi et al. [110] examined the insecure reuse of C++ code snippets in open-source GitHub projects (**D1**: C/C++). They manually assessed the security of the snippets (**D2**: ✘) and focused only on those explicitly attributed within projects (**D5**: ✔). The study did not consider the evolution of these snippets (**D3**: ✘) or their surrounding contexts (**D4**: ✘).

Selvaraj et al. [90] extended Zhang et al.'s [118] examine if revisions to C/C++ IoT code snippets (**D1**: C) could reduce vulnerabilities on Stack Overflow, Arduino, and Raspberry Pi Stack Exchange sites. Their methodology closely mirrors Zhang et al.'s (**D2**: ✔, **D3**: ✔, **D4**: ✘) but focuses solely on IoT code snippets (**D5**: ✔).

Acar et al. [116] studied how Android app developers (**D1**: Java) use code snippets from Stack Overflow to solve programming challenges. There were no restrictions on the reusable snippets (**D5**: ✘), and their security was manually determined (**D2**: ✘). Neither code evolution nor the surrounding context was considered in the study (**D3**:

✘); (**D4**: ✘).

Chen et al. [21] studied the security of Java code snippets (**D1**: Java), focusing on crypto APIs (**D5**: ✔). They manually labeled snippets as secure/insecure (**D2**: ✘) without considering code evolution or context (**D3**, **D4**: ✘).

Meng et al. studied Java code snippets (**D1**: Java) to explore the challenges in implementing secure solutions. They manually assessed security properties (**D2**: ✘) and focused on posts with security tags (**D5**: ✔), considering the surrounding context (**D4**: ✔) but not code evolution (**D3**: ✘).

Ragkhitwetsagul et al. [81] examined Java snippets (**D1**: Java) copied from GitHub to see if they remain outdated on Stack Overflow. They focused on accepted answers (**D5**: ✔) and code evolution to assess outdatedness (**D3**: ✔), without considering surrounding context (**D4**: ✘) or using a code scanner (**D2**: ✘).

Verdi et al. [110] examined the insecure reuse of C++ code snippets in open-source GitHub projects (**D1**: C/C++). They manually assessed the security of the snippets (**D2**: ✘) and focused only on those explicitly attributed within projects (**D5**: ✔). The study did not consider the evolution of these snippets (**D3**: ✘) or their surrounding contexts (**D4**: ✘).

Selvaraj and Uddin [90] extended Zhang et al.'s [118] examine if revisions to C/C++ IoT code snippets (**D1**: C) could reduce vulnerabilities on Stack Overflow, Arduino, and Raspberry Pi Stack Exchange sites. Their methodology closely mirrors Zhang et al.'s (**D2**: ✔, **D3**: ✔, **D4**: ✘) but focuses solely on IoT code snippets (**D5**: ✔).

Acar et al. [116] studied how Android app developers (**D1**: Java) use code snippets from Stack Overflow to solve programming challenges. There were no restrictions on the reusable snippets (**D5**: ✘), and their security was manually determined (**D2**: ✘). Neither code evolution nor the surrounding context was considered in the study (**D3**: ✘); (**D4**: ✘).

Chen et al. [21] studied the security of Java code snippets (**D1**: Java), focusing on crypto APIs (**D5**: ✔). They manually labeled snippets as secure/insecure (**D2**: ✘) without considering code evolution or context (**D3**, **D4**: ✘).

Meng et al. studied Java code snippets (**D1**: Java) to explore the challenges in implementing secure solutions. They manually assessed security properties (**D2**: ✘) and focused on posts with security tags (**D5**: ✔), considering the surrounding context (**D4**: ✔) but not code evolution (**D3**: ✘).

Ragkhitwetsagul et al. [81] examined Java snippets (**D1**: Java) copied from GitHub to see if they remain outdated on Stack Overflow. They focused on accepted answers (**D5**: ✔) and code evolution to assess outdatedness (**D3**: ✔), without considering surrounding context (**D4**: ✘) or using a code scanner (**D2**: ✘).

Bai et al. [12] examined Java code snippets (**D1**: Java) to investigate the propagation of insecure code from Stack Overflow. They relied on manual analysis (**D2**: ✘) to classify code snippets from answers only (**D5**: ✔). The authors did not consider the evolution of code snippets (**D3**: ✘), nor the context surrounding them (**D4**: ✘).

Bagherzadeh et al. [11] analyzed 186 real-world Akka actor bugs in Java and Scala code snippets (**D1**: Java/Scala) to understand and classify the symptoms, root causes, and their associated API usages. The Akka-related bugs and vulnerabilities were detected manually (**D2** = ✘) and code evolution was not considered in their study (**D3**:

✘). However, the authors analyzed the context surrounding code snippets, as it includes analysis of developer discussions in Stack Overflow and GitHub (**D4**: ✔). The study focuses only on Akka-related code snippets (**D5**: ✔).

Chen et al. [20] analyzed Java code snippets (**D1**: Java) to detect insecure snippets from answer posts using a novel hierarchical attention-based sequence learning model (**D2**: ✪, M; **D5**: ✔). The authors did not consider the evolution of code snippets (**D3**: ✘), nor did they analyze the context surrounding them (**D4**: ✘).

Zhang et al. [119] analyzed Java code snippets (**D1**: Java) with the ExampleCheck custom tool (**D2**: ✪, P) to assess the reliability of code examples in terms of API misuse. The study did not consider the evolution of code snippets (**D3**: ✘) nor their surrounding context (**D4**: ✘) and excluded code snippets from question posts (**D5**: ✔).

Rahman et al. [84] investigated the impact of reusing Java code snippets (**D1**: Java) by manually labeling (**D2**: ✘) the reusability, stability, and bug-proneness of code snippets when reused in mobile applications. They did not consider code evolution (**D3**: ✘) nor the context surrounding the code snippets considered in their study (**D4**: ✘), and they considered only code snippets in answer posts (**D5**: ✔).

Reinhardt et al. [85] analyzed Java code snippets (**D1**: Java) to detect API misuse using ExampleCheck (**D2**: ✔). The study did not consider code evolution (**D3**: ✘) nor the context around code snippets (**D4**: ✘). The study only considered code snippets that used specific Java APIs (**D5**: ✔).

Licorish et al. [64] analyzed Java code snippets (**D1**: Java) for security vulnerabilities using FindBugs (**D2**: ✔). The study did not consider code evolution (**D3**: ✘) but did analyze the context surrounding the code snippets, including discussions and comments, to gauge awareness of security issues (**D4**: ✔). The study did not filter out specific code snippets, analyzing all relevant code from Stack Overflow posts (**D5**: ✘).

Schmidt et al. [88] analyzed JavaScript and PHP Code snippets (**D1**: JavaScript/PHP) using CopypastaVulGuard, a tool that primarily relies on SQL queries and regular expressions to identify specific vulnerabilities, such as SQL injection, remote code execution, and deprecated functions in source snippets (**D2**: ✘). The study did not focus on code evolution (**D3**: ✘) but considered the surrounding context of the posts (**D4**: ✔). The study considered all relevant code snippets from Stack Overflow without filtering (**D5**: ✘).

Liu et al. [66] analyzed Java code snippets (**D1**: Java) to recommend and detect annotation misuses using a custom tool based on deep learning and multi-label classification (**D2**: ✪,M). The study did not consider code evolution (**D3**: ✘) but incorporated both structural and textual context (e.g., code comments) for annotation recommendation and misuse detection (**D4**: ✔). The study analyzed all Java code snippets relevant to its focus without filtering any specific snippets (**D5**: ✘).

Ren et al. [86] analyzed Java code snippets (**D1**: Java) to identify API misuse scenarios and generate demystification reports using a custom tool(**D2**: ✪, M) based on text mining. Although code evolution was not part of their methodology (**D3**: ✘), they did analyze the surrounding context of the code snippets, including discussions and API usage directives, to create comprehensive reports (**D4**: ✔). The study filtered out specific discussion threads by focusing on those that contained relevant API mentions, excluding other threads (**D5**: ✔).

Licorish et al. [65] analyzed Java code snippets (**D1**: Java) to investigate the effects of mutating Java code to fix performance-related issues detected by PMD (**D2**: ✔). The study did not consider the evolution of the code (**D3**: ✘) nor its surrounding context (**D4**: ✘). However, the authors analyzed all relevant code snippets without additional filtering (**D5**: ✘).

Rangeet Pan [77] analyzed Python code snippets (**D1**: Python) to examine the impact of bug-fixing on the robustness of deep learning models. The authors relied on a manual classification scheme to categorize bug-fix posts (**D2**: ✘) to classify code snippets and did not consider the evolution of code snippets (**D3**: ✘). However, they leveraged user discussions (**D4**: ✔) in the classification and filtered out bug-fix posts related to deep learning (**D5**: ✔).

Ye et al. [117] analyzed Java code snippets (**D1**: Java) to detect insecure code using a heterogeneous information network (HIN) and a novel network embedding model (snippet2vec) (**D2**: ✪, M). The study did not focus on code evolution (**D3**: ✘) nor considered the context surrounding around code snippets (**D4**: ✘) and focused on Android (**D5**: ✔).

Chen et al. [19] analyzed Java code snippets (**D1**: Java) to develop a *crowd debugging* technique based on machine learning to identify defective code fragments (**D2**: ✪, M). The study did not consider code evolution (**D3**: ✘), but the technique relies on explanations and suggestions provided by users in answer posts (**D4**: ✔). Only code snippets with meaningful code explanations were selected (**D5**: ✔).

Zhang et al. [120] manually collected 87 TensorFlow-related bug reports from Stack Overflow TensorFlow-related code snippets (**D1**: Python, **D2**: ✘) to identify bugs and their root causes. The study did not consider the evolution of snippets (**D3**: ✘), nor their surrounding context (**D4**: ✘). However, they did filter out irrelevant discussions, focusing on bugs related to TensorFlow (**D5**: ✔).

Alhanahnah et al. [5] analyzed Java code snippets (**D1**: Java) to detect insecure SSL/TLS implementation patterns, specifically targeting certificate and hostname validation vulnerabilities. They used the PMD tool with a custom ruleset designed to detect SSL/TLS security flaws (**D2**: ✔). The study did not consider the evolution of code snippets (**D3**: ✘) but considered the context surrounding them to identify insecure implementation patterns (**D4**: ✔) and focused specifically on snippets related to SSL/TLS (**D5**: ✔).

Imai et al. [50] analyzed the impact of reusing potentially vulnerable Java snippets (**D1**: Java) into Android applications. The study uses a custom tool based on pattern recognition (**D2**: ✪,P) to detect vulnerable snippets. The study did not examine the evolution of code snippets (**D3**: ✘) nor their surrounding context (**D4**: ✘) and focused exclusively on Android snippets (**D5**: ✔).

Fischer et al. [35] analyzed Java code snippets (**D1**: Java) to assess how *Google Search rankings* affect the security of cryptographic code examples that developers reuse. The study introduced a security-based re-ranking system, based on machine learning, to improve the visibility of secure code by altering the ranking of search results (**D2**: ✪,M). The authors did not consider the evolution of the cryptographic code snippets (**D3**: ✘) nor their context (**D4**: ✘). Similar to Fischer et al. [31, 36], the study focused on cryptographic code examples (**D5**: ✔).

Almeida et al. [6] developed REXSTEPPER, a debugger for JavaScript regular expressions (**D1**: JavaScript) designed to troubleshoot common issues. The study utilized *REXREF*, an off-the-shelve tool for detecting issues in the regular expressions (**D2**: ✔). The study focused on debugging 18 faulty regular expressions sourced from Stack Overflow (**D5**: ✔) but did not consider code evolution (**D3**: ✘) or their surrounding context (**D4**: ✘).

Islam et al. [53] analyzed code snippets related to deep neural networks from both Stack Overflow and GitHub (**D1**: Python) to investigate bug fix patterns to understand the common challenges and solutions used by developers. The study relied on manual classification of bug fixes (**D2**: ✘) and did not consider code evolution (**D3**: ✘) nor their surrounding context (**D4**: ✘). The authors only considered Python code snippets that use deep learning libraries (**D5**: ✔).

Mahajan et al. [69] analyzed Java code snippets (**D1**: Java) to recommend relevant Stack Overflow posts for fixing runtime exceptions by using a custom-built tool based on Abstract Program Graphs (APGs) (**D2**: ✪, A). The study did not consider the evolution of code snippets (**D3**: ✘) nor their surrounding context (**D4**: ✘) but filtered relevant posts based on certain criteria, e.g., question post must have an accepted answer (**D5**: ✔).

Mahajan et al. [70] analyzed Java code snippets (**D1**: Java) to automatically recommend real-time fixes for runtime exceptions (**D2**: ✔). The study did not consider the evolution of code snippets (**D3**: ✘) nor their surrounding context (**D4**: ✘) but considered question posts with an accepted answer (**D5**: ✔).

Yadavally et al. [114] analyzed pre-labelled Java and C/C++ code snippets (**D1**: Java, C/C++) from Stack Overflow (**D2**: ✘) to detect vulnerable code snippets. The study did not consider code snippet evolution(**D3**: ✘) nor the context surrounding them (**D4**: ✘). Additionally, they filtered out incomplete code snippets and focused on analyzing method-level code with a maximum of 8 LoC (**D5**: ✔).

Firouzi et al. [34] analyzed C# code snippets that use the *unsafe* keyword on Stack Overflow (**D1**: C#) using regular expressions and manual checks to extract and analyze the snippets for potential security vulnerabilities (**D2**: ✘). The authors did not consider code evolution (**D3**: ✘), nor did it their surrounding context (**D4**: ✘), and focused on extracting only code snippets that used the unsafe keyword (**D5**: ✔).

Ghanbari et al. [39] analyzed deep neural networks (**D1**: Python) to locate faults using mutation-based fault localization (**D2**: ✪, F) to localize faults in pre-trained DNN models. The study did not consider the evolution of code snippets (**D3**: ✘), nor their surrounding context (**D4**: ✘) and focused solely on deep-learning-related code snippets (**D5**: ✔).

Gao et al. [38] proposed an automatic approach to fixing recurring crash bugs in Java code snippets (**D1**: Java). They used off-the-shelve tools for partial parsing, tree-based code differencing, and edit script generation to identify and apply fixes for crash bugs in code snippets (**D2**: ✔). They did not consider the evolution of code snippets (**D3**: ✘), nor their surrounding context (**D4**: ✘). Only code snippets containing fixes to crash bugs were sampled (**D5**: ✔).

M. Chakraborty[18] analyzed the reuse of the pre-trained BERT model in Python code snippets (**D1**: Python) to identify bugs related to the reuse of BERT. The authors used a manual analysis approach to identify those bugs (**D2**: ✘) and did not consider the evolution of these BERT-related buggy snippets (**D3**: ✘), nor their surrounding context (**D4**: ✘). Posts tagged with BERT or related keywords with a score *ge* 2 were considered (**D5**: ✔).

Moradi Moghadam et al. [73] proposed μAkka, a mutation testing framework for actor concurrency in the Akka system, using 130 real-world Akka bugs sourced from Java code snippets (**D1**: Java). The authors manually reviewed (**D2**: ✘) the 130 code snippets containing those bugs. The evolution of the snippets was not considered (**D3**: ✘), nor took into account their surrounding context (**D4**: ✘) and focused specifically on Akka actor bugs on Stack Overflow (**D5**: ✔).

Magnus Madsen et al. [68] analyzed JavaScript promises (**D1**: JavaScript) to detect promise-related errors, using a promise graph (**D2**: ✪, S). They did not examine the evolution of code snippets (**D3**: ✘), but they extensively studied the surrounding context by manually inspecting StackOverflow discussions to understand errors related to promise usage (**D4**: ✔). They also applied filters by focusing on StackOverflow posts that were directly related to promise errors rather than all JavaScript code snippets (**D5**: ✔).

Jhoo et al. [56] analyzed Python code snippets (**D1**: Python) to detect tensor shape errors in PyTorch snippets using a custom static analyzer tool called PyTea (**D2**: ✪, S). The authors did not consider code evolution (**D3**: ✘) nor the context surrounding them (**D4**: ✘). Only PyTorch-specific code snippets were considered (**D5**: ✔).

MQ1: What aspects of Stack Overflow affect the validity of prior research? Criteria **D1**–**D5** in Table 4.2 show that programming language-specific trends and evolution may affect the stability of results over time. Further, most of these works leverage some form of code classification, making their results susceptible to changes as the code evolves. Five works rely upon code evolution in their methodology, showing that ongoing evolution beyond the study's timeframe can influence their results. Additionally, two studies focus on the context surrounding code snippets, so any changes in context, like the addition of security-relevant comments, could also impact their findings.

4.4 Evolution of Stack Overflow

Based on the identified aspects of Stack Overflow that *may* affect prior research (MQ1), we now look at the global evolution of the programming languages and security-relevant contexts on Stack Overflow. Measuring the security of code snippets directly, however, is a challenging task and requires dedicated methodologies (e.g., [118, 49, 36, 31]). We address the evolution of code snippet security through case studies in Section 4.5 and focus here on programming languages and security-relevant edits and comments.

Programming Languages Figure 4.3 depicts the monthly addition of new code snippets in the programming languages considered in the 14 works from Table 4.2. The

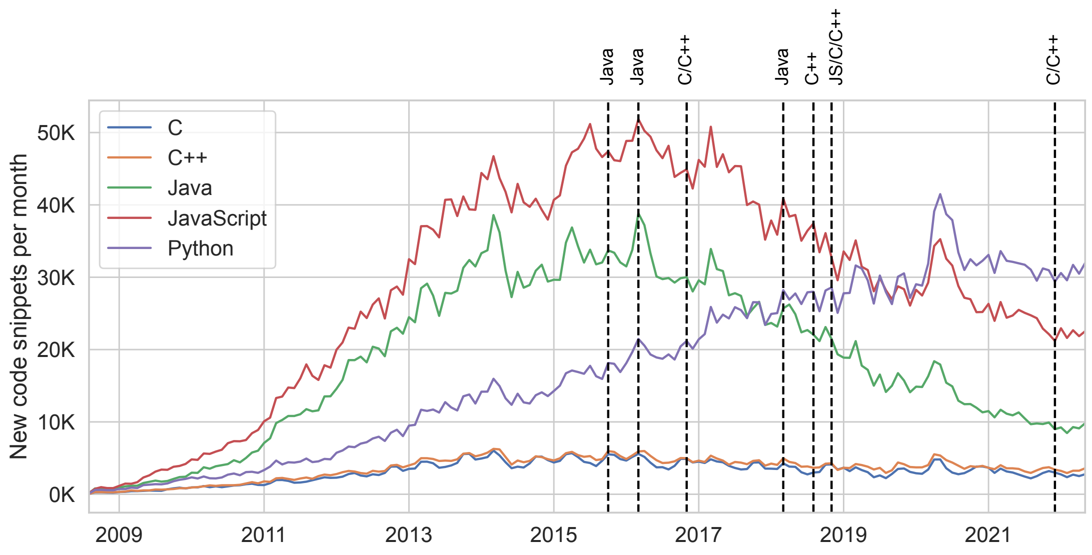


Figure 4.3: Added code snippets on Stack Overflow per month. Dashed lines indicate data collection points of snippets by the works in our systematization (see Table 4.2).

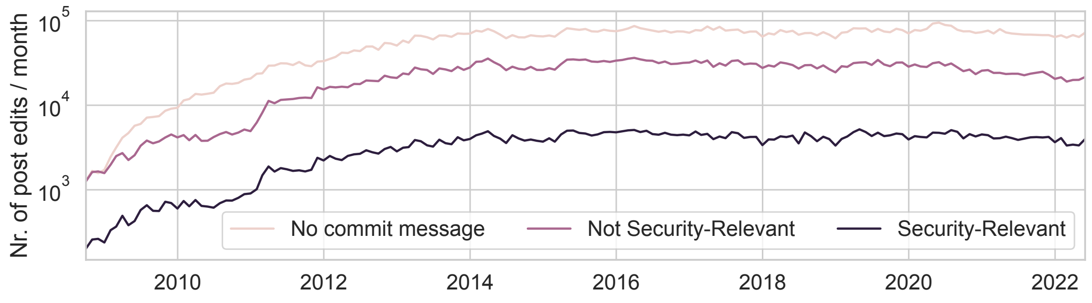
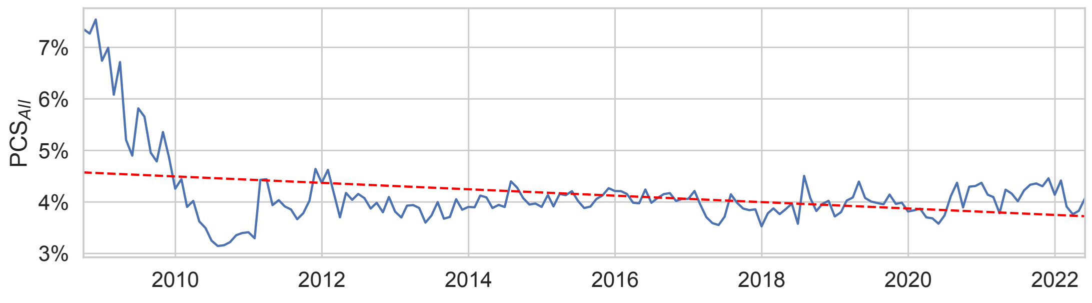


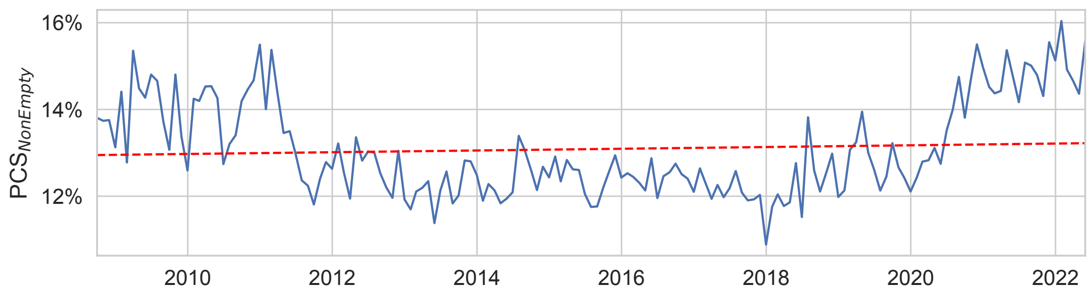
Figure 4.4: Number of monthly (30-day interval) post edits categorized by their security relevance.

data shows that while C/C++ is relatively stable over time but at a comparatively low rate, Java and JavaScript peaked between 2013 and 2018 and have been shrinking on Stack Overflow since 2018. We indicate with dashed lines when the papers from our systematization sampled their data.

Security-Relevant Edits and Comments We found that the average code snippet on Stack Overflow receives 1.66 edits (max = 754, $P_{75} = 2$, $P_{99} = 6$). A breakdown of average code snippet edits per language is provided in Table B.2 in Appendix B.2. Figure 4.4 shows the number of monthly *post* edits on Stack Overflow between 09/2008–06/2022, broken down by their security relevance and commit message. To identify security-relevant edits, we apply the publicly shared NLP-based classifier by Jallow et al. [P1, 8] on the commit messages of the edits. We found 9,443,509 code edits without a commit message, 3,731,935 non-security-relevant commit messages, and 549,863 commit messages indicating security relevance. Our data shows that 514,666 answer posts have



(a) Including empty commit messages.



(b) Excluding empty commit messages.

Figure 4.5: Percentage of security-relevant commits (PSC) in monthly intervals. Dashed lines are fitted linear regressions.

received at least one security-relevant commit (max = 27). We calculate the percentage of security-relevant commits (PSC):

$$PSC = \frac{\text{Number of security-relevant commits} * 100}{\text{Total number of commits}}$$

On June 20, 2020, Stack Overflow changed to CommonMark [111] and adjusted all 338,622 nonconforming posts with automated edits. For data sanity, we exclude these script-generated edits from the PSC calculation. Figure 4.5 depicts the PSC for all code edits on Stack Overflow, where Figure 4.5a counts empty commit messages to the total number of commits and Figure 4.5b excludes them. The monthly average PSC is $\text{PSC}_{\text{All}} = 4.2\% \pm 0.1$ and $\text{PSC}_{\text{NonEmpty}} = 13.1\% \pm 0.2$ ($CI = 95\%$). We use the Kwiatkowski–Phillips–Schmidt–Shin (KPSS) test [62] to test for stationarity of the PSC over the long term. Since KPSS is known to exhibit a high rate of Type-I errors, indicating non-stationary too often, we additionally test non-stationary of the PSC with the Augmented Dickey Fuller (ADF) test [37]. Table B.5 in Appendix B.2 details the results and illustrates the corresponding PSC. In summary, when focusing on non-empty commit messages, we find that the overall PSC is trend stationary (KPSS $p > 0.05$; ADF $p > 0.05$), the PSC for C code snippets is difference stationary (KPSS $p > 0.05$; ADF $p < 0.001$) with an increasing PSC (i.e., the difference between data points is stationary and the PSC has a linear upward trend over time), and the PSC for JavaScript snippets is non-stationary (KPSS $p < 0.05$; ADF $p > 0.05$). All but C++ and Python edits show a stationary PSC when considering non-empty commit messages. We also applied linear regressions to the PSC to highlight their long-term trends, with the results presented in Appendix B.2.

Regarding comments, we found that 3,991,533 (8.1%) of 49,087,103 comments were security-relevant. These relevant comments are for 3,128,208 posts, of which 98,139 also received a security-relevant commit message. The mean percentage of security-relevant comments $\mathrm{PSC}_{\mathrm{Comments}}$ is 7.78±0.143, where posts with C/C++ snippets have a significantly higher mean $\mathrm{PSC}_{\mathrm{Comments}}$ between 11–12% (see Table B.6 in Appendix B.2). Further, KPSS and ADF tests show that the overall $\mathrm{PSC}_{\mathrm{Comments}}$ is difference stationary and that the $\mathrm{PSC}_{\mathrm{Comments}}$ for C/C++, Java, JavaScript, and Python posts is non-stationary (see Table B.7 in Appendix B.2). A fitted linear regression confirms this increasing trend of $\mathrm{PSC}_{\mathrm{Comments}}$ ($R^2 = 0.93$, $p < 0.001$ for all comments). A potential explanation for this development could be the decreasing number of new code snippets (e.g., because simple questions are now posted to GenAI tools) and a continued community effort to curate the Stack Overflow content.

MQ2: How much do Stack Overflow code snippets and surrounding context evolve? Our data shows that programming languages trend differently in their overall number of added snippets and their ratio of security-relevant edits. Thus, studies focusing on particular programming languages will likely find a different landscape when conducted at different times. Further, many comments raised security-relevant issues but were largely not on posts that received a security-relevant edit. Over time, the ratio of security-relevant comments steadily increased, indicating that the community strives to improve content quality.

4.5 Replication Case Studies

We present replication case studies to answer **MQ3** as to whether the findings of prior research that relied on specific versions of the Stack Overflow dataset hold over time. In contrast to Section 4.3, we aim to find concrete evidence for the impact of Stack Overflow evolution on research results by replicating the findings of the six studies in Table 4.2.

Excluded papers. As shown in Table 4.2, we focused on six papers for replication and excluded the others. In 16 studies [12, 11, 84, 88, 77, 120, 53, 114, 34, 18, 73], the detection of security weaknesses and bugs in code snippets was not automated; instead, these studies relied on manual processes to label and identify security issues and/or bugs in code snippets. This approach typically involved human annotators reviewing and classifying the code for potential weaknesses, which is hard to compare in a replication study with different human evaluators. On the other hand, three studies[86, 19, 35] leveraged machine learning techniques to automate the detection of security vulnerabilities in code snippets. While these studies employed advanced algorithms to facilitate automated analysis, they failed to release the underlying source code and datasets, particularly the training data used for training their models, making replication hard. Additionally, nine studies [85, 66, 117, 20, 5, 6, 38, 68, 56] did not specify which version of the dataset they used to collect code snippets from Stack Overflow. Lastly, nine studies [116, 81, 12, 86, 35, 69, 19, 70, 84] were excluded in favor of non-user-driven studies.

4.5.1 Case Study 1: C/C++ Code Weakneses

Table 4.3: Comparison of results by Zhang et al. (118) and our replication study using SOTorrent22 and Cppcheck v2.13.

Based on SOTorrent18 (Original)				**Based on SOTorrent22 (Replication)**			
	Answer #	*Code Snippet #*	*Code Version #*		*Answer #*	*Code Snippet #*	*Code Version #*
SOTorrent	867,734	1,561,550	1,833,449	SOTorrent	1,096,380	1,944,378	2,340,975
LOC >= 5	527,932	724,784	919,947	LOC >= 5	695,326	938,643	1,234,443
Guesslang	490,778	646,716	826,520	Cppcheck 2.13	323,321	388,749	507,997
$Code_w$	**11,235**	**11,748**	**14,934**	$Code_w$	**28,521**	**30,254**	**38,248**

Zhang et al. [118] studied whether revisions to C/C++ snippets increase or decrease the snippets' security. Their work addressed the following questions: **RQ1**: *What are the types of code weaknesses that are detected in C/C++ code snippets on Stack Overflow?* **RQ2**: *How do code with weaknesses evolve through revisions?* **RQ3**: *What are the characteristics of the users who contributed code with weaknesses?*

4.5.1.1 Original Methodology

The authors employed a data-driven approach (**R4**: ✘), focusing exclusively on answer posts from the *SOTorrent18* dataset (released in 12/2018), without considering code reused from Stack Overflow (**R3**: ✘). The left-hand side of Table 4.3 shows the originally collected data. The authors extracted 867,734 answers containing 1,561,550 code snippets with C/C++ tags with 1,833,449 versions. From this data set, they filtered all code snippets with less than five LoC. This resulted in 724,784 code snippets (with 919,947 code versions) from 527,932 answers. However, the authors noticed that using *tags* alone is insufficient to determine the language of code snippets, i.e., not all the snippets contain valid C/C++ code. Using the *Guesslang* machine learning classifier [42] they filtered non-C/C++ code snippets (**R2**: ✪, **M**). This resulted in 646,716 code snippets (having 826,520 versions) from 490,778 answers. In a final step, the authors leveraged the *CppCheck* static analysis tool to scan all 826,520 code snippet versions to detect security weaknesses. We reuse the terms by the authors to denote code snippets, snippet versions, and answers with security weaknesses as $Code_w$, $Version_w$, and $Answer_w$, respectively. The authors' final dataset to answer their RQ1 and RQ2 are $Version_w = 14,934$ from $Code_w = 11,748$ in $Answer_w = 11,235$.

4.5.1.2 Re-Implementation

The authors did not publicly release their source and data artifacts (**R1**: ✘) but provided us on request with a CSV file of the $Answer_w = 11,235$. The unavailability of the code artifact forced us to re-implement their methodology for replication.

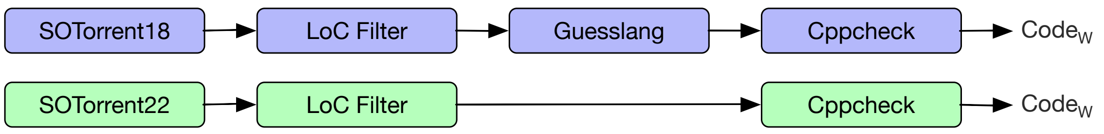


Figure 4.6: Comparison of Zhang et al.'s (118) methodology with the approach used in our replication study.

An exact re-implementation of Zhang et al.'s [118] methodology was impossible. The Guesslang version used by the authors is outdated and unavailable. Replacing the old version with the newest version, v2.2.1, yielded significantly different numbers for the same *SOTorrent18* data set (490,778 vs.249,451). Additionally, we noticed that Guesslang v2.2.1 misclassified 4,901 C/C++ answers (out of $Answer_w = 11,235$) from the authors' results as Java. Given the substantial differences of Guesslang v2.2.1, we devised an alternative methodology that skips Guesslang (see Figure 4.6). We relied solely on Cppcheck to identify valid C/C++ snippets and detect security weaknesses. Cppcheck attempts to compile code snippets, failing if the snippets are not valid C/C++ code. Thus, Cppcheck formed an additional language detection step in the original methodology, which is now the only such step. The intuition is that any invalid C/C++ code detected by Guesslang would be rejected by Cppcheck.

A second challenge for re-implementation was that the authors did not specify the exact Cppcheck version used in their study and did not respond to multiple inquiries. Ultimately, we resorted to brute force by testing 15 different versions of Cppcheck on the provided $Answer_w$ list against the reported results in the paper. We found version v1.86 to be closest to the original results. This version detected $Answer_w = 15,724$, of which 11,142 (99.2% of 11,235) were also identified by the authors. We surmise the additional 4,489 answers result from erroneous filtering with Guesslang in the original approach.

We evaluated our re-implementation with the *SOTorrent18* data set that was also used by the authors, aiming to verify that our approach is within an acceptable error margin from the authors' results and allowing us to replicate their study with a newer data set faithfully. Table B.8 in Appendix B.3 compares with the original findings and shows that our approach resembles the original methodology closely enough.

4.5.1.3 Replication

We replicate their findings using *SOTorrent22*, released four years after the original dataset. We use Cppcheck v2.13 to show how the results would differ if the study were conceived years later. Table B.10 in Appendix B.3 shows results for different combinations of Cppcheck and SOTorrent versions.

Table 4.3 compares the original results with the replication based on *SOTorrent22* and Cppcheck v2.13. We found that the number of code snippets with $LoC >= 5$ increased between 2018 and 2022 (724,784 $\nearrow$ 938,643), a growth rate of 29.5%. Among these, the $Code_w$ also increased (11,748 $\nearrow$ 30,254), a growth rate of 157.5%. With this $Code_w$ dataset based on the newer SOTorrent version, we can replicate the authors' findings for **RQ1** and **RQ2** to see if their conclusions still hold. Table 4.4 presents

Table 4.4: Side-by-side summary of the main claims in Zhang et al.'s paper (118) for **RQ1** and **RQ2** and claims based on the results of our replication using *SOTorrent22*.

Original Results Based on SOTorrent18	**Replication Results Based on SOTorrent22**
RQ1 *What are the types of code weaknesses that are detected in C/C++ code snippets on Stack Overflow?*	
The authors found 36% (i.e., 32 out of 89) of all the C/C++ CWE types in C/C++ code snippets on Stack Overflow.	We found 37% (i.e., 33 out of 89) of all the C/C++ CWE types in C/C++ code snippets on *SO*. CWE-476 is a newly introduced type for snippets after December 2018 and has 1,159 instances.
The authors identified 12,998 CWE instances within the latest versions of the 7,481 answers.	We identified 7,679 CWE instances within the latest versions of the 5,721 answers.
The authors found CWE-758 to be the sixth most prevalent CWE type in C/C++ code snippets with 482 (3.7%) instances.	We found CWE-758 to be the second most prevalent CWE type in C/C++ code snippets with 10,911 instances.
The authors found 10,533 CWE instances in the TOP-6 most prevalent CWE types affecting C/C++ code snippets on *SO*.	While we found 42,984 CWE instances in the TOP-6 most prevalent CWE types affecting C/C++ code snippets on *SO*.
RQ2 *How does code with weaknesses evolve through revisions?*	
The authors identified 12,998 CWE instances within the latest versions of 7,481 answers.	We identified 23,926 CWE instances within the latest versions of the 18,071 answers.
As the number of revisions increases from one to ≥ 3, the proportion of improved $Code_w$ increases from 30.1% to 41.8%.	As the number of revisions increases from one to ≥ 3, the proportion of improved $Code_w$ increases from 3.1% to 7.4%.
In $Code_w$ with different rounds of revisions, a larger proportion of code snippets have reduced rather than increased the number of security weaknesses.	We observed a smaller proportion of code snippets whose associated security weaknesses reduced with different rounds of code revisions.
The authors found 92.6% (i.e., 10,884) of the 11,748 $Code_w$ had weaknesses introduced when their code snippets were initially created on Stack Overflow. They found 10,884 $Code_w$ introduced in the snippets' first version.	We found 93.1% (i.e., 28,155) of the 30,254 $Code_w$ had weaknesses introduced in their first version. However, we discovered significantly more $Code_w$ introduced when code snippets were initially created: 28,155.
69% (i.e., 8,103 out of 11,748) of the $Code_w$ has never been revised.	80.6% (i.e., 24,388 out of 30,254) of $Code_w$ has never been revised.

a side-by-side comparison of the authors' claims and our findings based on the newer dataset for the authors' **RQ1** and **RQ2**. Below, we summarize the main points.

Revisiting RQ1 Findings: We found that an additional CWE type (CWE-476) has appeared since December 2018, which is now the sixth most prevalent CWE type. Similarly, the authors identified 12,998 CWE instances in the latest versions of 7,481 answers, whereas we found 7,679 instances in 5,721 answers. Further, we noticed a shift in the ranking of CWE types: CWE-758 climbed 6th $\nearrow$ 2nd; CWE-401 dropped 2nd $\searrow$ 3rd; CWE-775 fell 3rd $\searrow$ 7th.

We found that several of Zhang et al.'s [118] original conclusions regarding the types of code weaknesses are no longer valid. *SOTorrent22* contains proportionally more vulnerable snippets with different ratios for CWE types and the emergence of a new CWE type.

Revisiting RQ2 Findings: The authors noted that as the number of revisions to $Code_w$ increased from 1 to 3+, the proportion of *improved* $Code_w$ rose from 30.1%

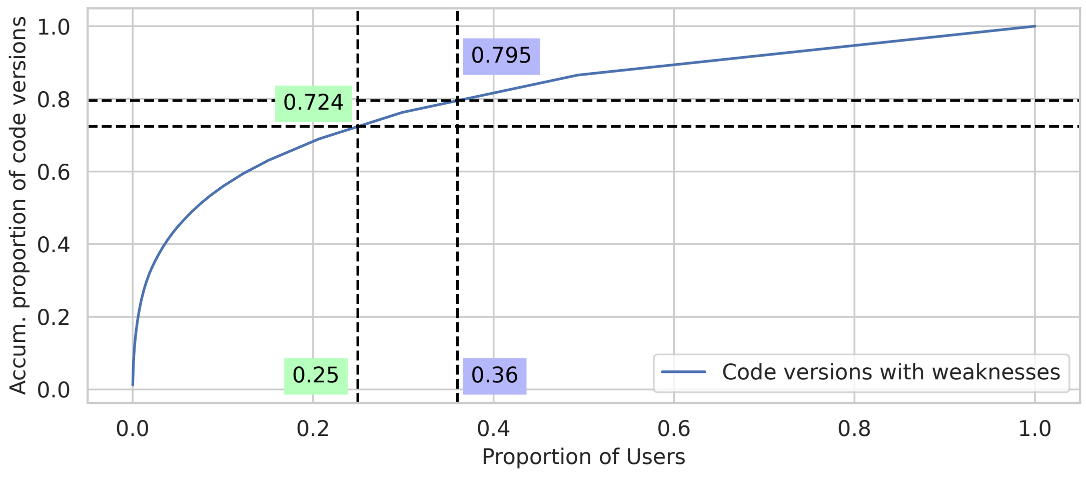


Figure 4.7: The accumulative proportion of $Version_w$ posted by the proportion of users. Annotations indicate the original (cf. Figure 9 in (118)) and replicated significant data points.

to 41.8% (see Table B.9 in Appendix B.3). As a result, the authors concluded that a larger proportion of $Code_w$ has reduced rather than increased, indicating that revisions improve code security. In contrast, our results showed significantly lower improvement rates. As revisions increased from 1 to 3 or more, the proportion of improved $Code_w$ only rose from 3.1% to 7.4%.

Our replication shows that if the authors had conducted their study 4 years later, they would have observed only a slight increase in the proportion of *improved* $Code_w$.

Revisiting RQ3 Findings: RQ3 deals with Stack Overflow users. The authors found that *"the majority of the C/C++ $Version_w$ were contributed by a small number of users"* and that *"72.4 percent (i.e., 10,652) of $Version_w$ were posted by 36 percent (i.e., 2,292) of users."* In our replication, we found a shift where an even smaller number of users contributed $Version_w$, see Figure 4.7. We found that 72.4 percent (i.e., 35,034) of $Version_w$ were posted by 25 percent (i.e., 3,205) of users. In our data set, 36 percent (i.e., 4,625) of users contributed 79.5 percent (i.e., 38,481) of the $Version_w$. Moreover, Zhang et al. reported that *"64.0 percent (i.e., 4,070) of the users who contribute $Version_w$ have contributed only one $Version_w$."* We found that 86.2 percent (i.e., 11,077) of users contributed only one $Version_w$. Further, they reported that *"among all the 85,165 users who posted C/C++ code snippets, only 7.5 percent (i.e., 6,361) of them posted code snippets that have weaknesses."* In contrast, in our replication study, 17.0 percent (i.e., 12,845) of 75,779 users contributed code snippets with weaknesses.

Next, Zhang et al. explored the connection between user activity and code weaknesses. They found that "more active users are less likely to introduce $Code_w$." We depict the same connection in Figure 4.8, adopting the authors' plot style. The authors concluded

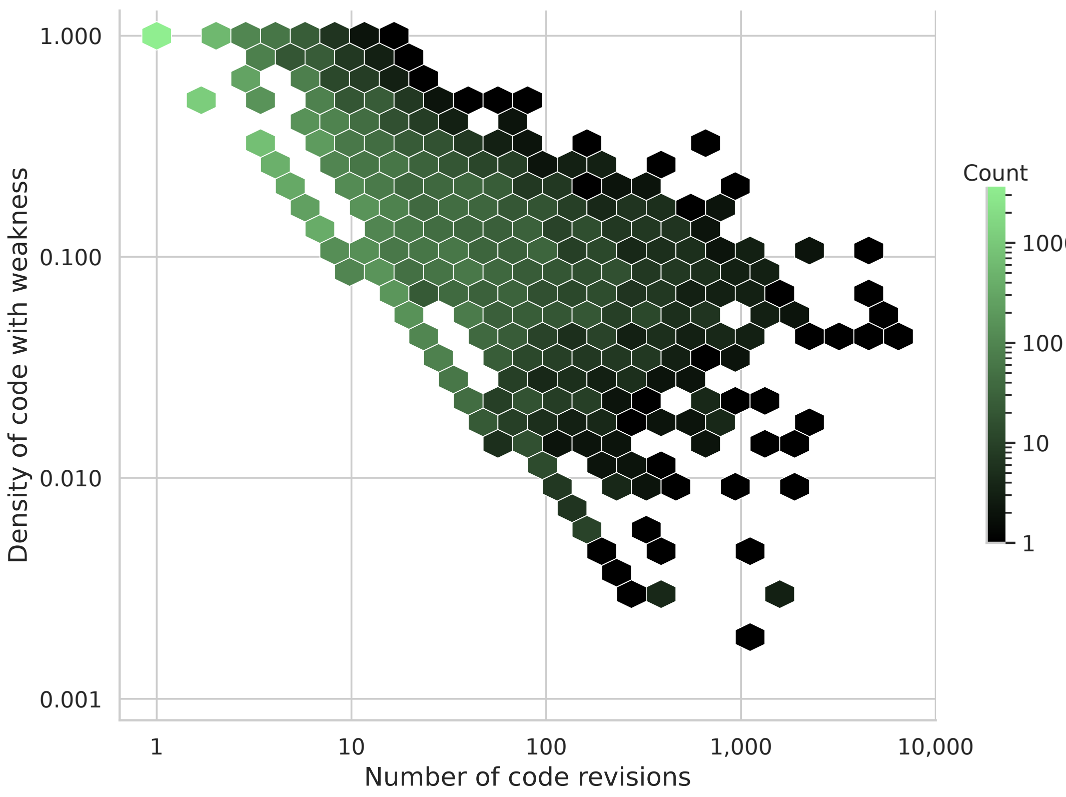


Figure 4.8: As the number of code revisions increases for a user, the density of contributed $Version_w$ by that user drops. Based on data from replication study. (cf. Figure 10 in (118))

that *"the weakness density of a user's code drops when the number of contributed code revisions by the user increases."* We explored the relation between the number of code revisions and the density of contributed $Version_w$ by users with statistical testing. A Pearson correlation analysis revealed a weak correlation, $r(4) = -0.190, p < 0.001$, suggesting that as the number of revisions increases, the density of code with weaknesses decreases slightly. A linear regression analysis was conducted to examine further the relationship between the number of revisions (independent variable) and the weakness density (dependent variable). The results indicated that the model was statistically significant $F(1, 12843) = 478.4$, $p < 0.001$, suggesting a significant inverse association of code revision count with weakness density. However, the model explained only a small proportion of the variance in weakness density $R^2 = 0.036$. These findings suggest that the weakness density is expected to decrease by 0.0005 for each additional revision. However, the low R^2 value indicates that the number of revisions explains only 3.6% of the variability in weakness density, suggesting that other factors may also play a significant role. Overall, the correlation and regression analyses support the conclusion that there is a statistically significant but weak inverse relationship between the number of revisions and weakness density. Further research is needed to explore additional variables and potential non-linear relationships that might better explain the variability in weakness density.

Additional details on the replication results for Zhang et al.'s RQ3 and a note about a possible multiplicity in counting CWE instances can be found in our extended version [55].

We found that the fraction of users with vulnerable C/C++ snippets more than doubled compared to the original findings. Moreover, the number of users that contributed just one vulnerable snippet also increased. Further, the authors reported that users who contributed multiple vulnerable snippet versions repeatedly contributed the same CWE type. We found that these users contribute different types of CWE with the same likelihood.

4.5.2 Case Study 2: Discovering Insecure Code Snippets

Hong et al. [49] built Dicos to discover insecure code snippets by examining snippet revisions for changes in security-sensitive APIs, security-related keywords, and control flows. A code snippet is classified as insecure if at least two changes occurred between its initial and most recent version. The authors used tags to identify the programming language of code snippets (**R2**: ✔). Although the Dicos source code is available on GitHub[48], it contains several bugs that required fixing. Further, their dataset of labeled snippets is not available, even on request (**R1**: ○).

The authors followed a data-driven approach (**R4**: ✘) to answer the following questions:

RQ1 *Are older posts more likely to provide insecure code snippets?*

RQ2 *Are accepted answer posts more secure than non-accepted posts?*

RQ3 *What types of insecure code snippets were discovered?*

RQ4 *What is the status of reusing insecure code snippets in popular open-source software?*

We replicate the study for **RQ2** and **RQ3** to determine if their results still hold today. We excluded **RQ1** because code evolution does not affect its findings, i.e., code evolution cannot retroactively *add* old posts, only evolve them. However, we used the results of **RQ1** to evaluate the released Dicos tool, which we will discuss later in this section. We excluded **RQ4** because it deals with code reuse from Stack Overflow in open-source projects on GitHub (**R3**: ✪, **S**). Though RQ4 is an interesting question for replication, our focus is on code evolution within Stack Overflow.

4.5.2.1 Original Methodology

The authors reported 93% precision, 94% recall, and 90% accuracy in discovering insecure **C/C++** code snippets with Dicos while for **Android** code snippets, it has 86% precision, 89% recall, and 86% accuracy. The evaluation was done using the *SOTorrent20* dataset, released in 12/2020, from which they extracted 668,520 posts containing 1,514,547 code snippets. For replication, we need to re-evaluate DICOS's precision, recall, and accuracy using the more recent SOTorrent22 dataset. The authors evaluated the accuracy of DICOS against the results by Fischer et al. [31] and Verdi et al. [110]. However, this methodology creates a barrier to replication: the authors only used the datasets provided by Fischer et al. and Verdi et al. rather than directly running DICOS and these tools on the same input. Specifically, they compared the insecure snippets identified by DICOS against the labeled code snippets from Fischer et al. and the insecure C++ snippets found by Verdi et al. To replicate this experiment, we would need to follow the same approach, using the labeled examples from related work to compare with DICOS's findings on a newer version of SOTorrent. Unfortunately, since these tools are unavailable and the labeled data from Fischer et al. and Verdi et al. only represent the old SOTorrent data set, this evaluation approach is infeasible when using a newer SOTorrent version.

4.5.2.2 Implementation

The DICOS code base is available on GitHub. We used this released source code for our study to replicate the authors' findings on a newer dataset version. However, we should note that we found several bugs in the code base for which we developed fixes for future work using DICOS. Further, some approaches discussed in their paper were only partially implemented. For example, the use of Joern[107] to discover changes to control flow statements was incomplete; the authors confirmed Joern was not used in the analysis. During testing, we found that their pairing technique based on the Jaccard index to match code snippets in the first and last versions of posts resulted in many false positive pairs. Clone detection is a better technique for this task, as it is more accurate for source code. We will publicly release a refactored version of DICOS with the option to use clone detection as a pairing technique.

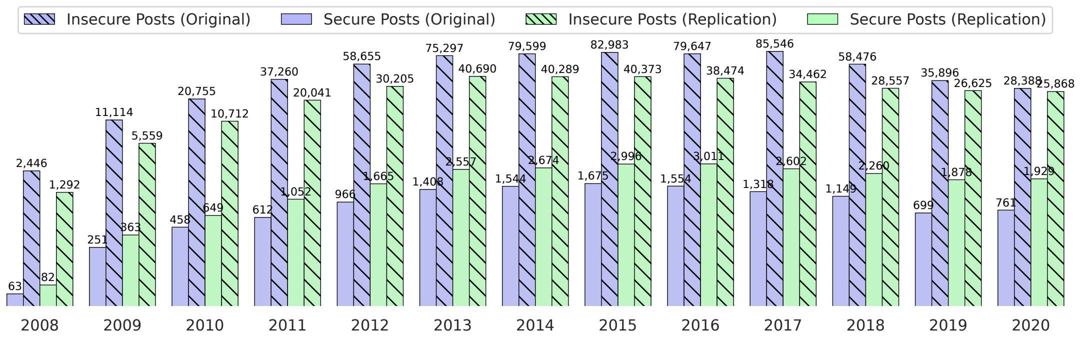


Figure 4.9: Comparison of yearly distribution of secure and insecure posts discovered by DICOS (logarithmic scale) as reported by the authors (Figure 6 in (49)) and found in our replication study.

We evaluated the authors' implementation of DICOS using the same dataset version of the original work to determine if we could reach the same conclusions with their source code and dataset, i.e., if we have a reproducible methodology as the basis for replication. However, we were unable to reproduce the number of Stack Overflow posts using the same SQL queries[47] and dataset as the authors. While the authors reported 987,367 C/C++ and 970,916 Android posts, we retrieved 867,962 C/C++ and 986,900 Android posts. Furthermore, we identified 26,550 insecure posts, 14,092 (or x2.13) more than the 12,458 insecure posts the authors reported. Unfortunately, we did not receive a response from the authors in an attempt to clarify this reproduction issue. Further, we reproduced the authors' findings for their ***RQ1***. The authors concluded that older posts are not more likely to introduce insecure snippets than newer ones. We observed the same trend but found different yearly numbers of secure/insecure posts from those reported by the authors. we reproduced the authors' results for **RQ1** using the same dataset version. Our comparative findings are shown in Figure 4.9. The authors concluded that older posts are less likely to introduce insecure posts than newer ones. While we observed similar trends, we found different yearly numbers of secure/insecure posts from those reported by the authors. For instance, in 2008, the authors observed 63 insecure and 2,446 secure posts while we observed 82 insecure and 1,292 secure posts.

A two-sample z-test for the *overall* proportions of the insecure to secure posts ratio between the original and our replication results shows a significant difference ($Z = -121.962$, $p < 0.001$). To evaluate whether the proportion of insecure posts differed significantly between the originally reported results and our replication study over the years, we conducted two-proportion z-tests for each year from 2008 to 2020. The Holm–Bonferroni correction [46] was applied to account for multiple comparisons across the 13 years. Our results indicate that for all years, the differences in the proportion of insecure posts between the two studies are statistically significant after the Holm–Bonferroni correction (adjusted α levels ranged from 0.0038 to 0.05, all p-values < 0.05). This suggests a consistent and significant disparity in the proportion of insecure posts across the entire study period.

While the obtained results may not reflect accurate values due to the tool's quality, they nevertheless allow us to measure how code evolution affects such a study.

4.5.2.3 Replication

We used *SOTorrent22*, which was released *two* years after the *SOTorrent20* dataset used by the authors. We followed the authors' approach to collect posts from *SOTorrent22*. The authors extracted 1,958,283 posts (987,367 C/C++ and 970,916 Android), whereas we extracted 2,858,003 posts (1,489,148 C/C++ and 1,368,855 Android posts). This indicates a 51% increase in C/C++ posts and a 41% increase in Android posts since December 2020. The authors filtered all single-version posts, resulting in 668,520 (34% of 1,958,283) multi-version posts. After filtering all single-version posts from the extracted *SOTorrent22* posts, we obtained 1,046,052 posts (36.6% of 2,858,003) with at least two versions.

Finally, the authors applied DICOS on their dataset of 668,520 posts and discovered 12,458 (1.9%) insecure posts (8,941 C/C++ and 3,517 Android). Of these, 788 (6.3%) insecure posts contained all three types of changes while the remaining 11,670 insecure posts contained two types of changes. Using the same approach on our dataset of 1,046,052 posts, we discovered 30,359 (2.9%) insecure posts, i.e., an **increase of 52%**.

Among these, 4,887 (16.1%) insecure posts contained all three features, i.e., an **increase of 155%**, while the remaining 25,472 posts contained two features.

***Accuracy of* Dicos**: To compute the accuracy, the authors manually verified a subset of the 12,458 discovered insecure posts using the following groups:

G1: All posts with three changes

G2: Top 200 posts with two changes

G3: Randomly selected 100 posts with two changes

G4: Top 200 posts with only one change

G5: Top 100 posts without changes

Groups **G1–G3** were used to measure true and false positive rates for detecting insecure posts. Groups **G4** and **G5** were used to measure the true and false negative rates. Two researchers manually verified the posts for each group, recording the positive and negative rates for C/C++ and Android. We replicated this approach to calculate the precision, recall, and accuracy of Dicos using *SOTorrent22*. We adhered to the authors' original method, verifying 788 insecure posts for **G1** by randomly selecting 788 posts for manual verification by two researchers.

	Original accuracy measurement results for C, C++, and Android posts based on *SOTorrent20*					**Replicated accuracy measurement results for C, C++, and Android posts based on *SOTorrent22***				
ID	**#Posts**	**#TP**	**#FP**	**#TN**	**#FN**	**#Posts**	**#TP**	**#FP**	**#TN**	**#FN**
G1	788	757	31	N/A	N/A	788	95	693	N/A	N/A
G2	400	346	54	N/A	N/A	400	33	367	N/A	N/A
G3	200	162	38	N/A	N/A	600	66	534	N/A	N/A
G4	400	N/A	N/A	318	82	400	N/A	N/A	379	17
G5	200	N/A	N/A	185	15	200	N/A	N/A	188	12
Total	1,988	1,265	123	503	97	2,388	194	1,594	567	29
Precision					**0.91**					**0.11**
Recall					**0.93**					**0.87**
Accuracy					**0.89**					**0.32**

Table 4.5: Comparison of precision, recall, and accuracy measured by authors (Table 5 in (49)) vs. our replication (SOTorrent22).

For **G3**, we faced the challenge that the authors conducted a one-time random selection of 100 posts with two features. In contrast, we performed three random selections of 100 posts and averaged the results.

Table 4.5 compares the original accuracy of Dicos reported by the authors with our findings using a newer dataset. We found that Dicos had an 11% precision (compared to the authors' 91%), 32% accuracy (versus 89%), and an 87% recall (versus 93%). These results indicate that the performance of Dicos has significantly decreased due to the code evolution on Stack Overflow. The replicated accuracy measurements for C/C++ and Android are in Appendix B.4.

We found that the code snippet evolution has adversely affected the precision and accuracy of DICOS's approach to detecting vulnerable snippets. Considering the stable recall, DICOS on newer Stack Overflow versions is better suited for detecting *secure* snippets.

Revisiting RQ2 Findings: The authors investigated the relationship between the security weaknesses of code snippets in accepted and non-accepted answers and found no difference between the ratios of insecure posts for accepted (1.67%) and non-accepted (1.99%) answers. Figure 4.10 compares the original results (refer to Figure 7 in [49]) and the replication findings. Using the newer version of the SOTorrent dataset, we found a significantly higher ratio of insecure posts in both accepted (1.67% ↗ 92.83%) and non-accepted answers (1.99% ↘ 93.7%). Nevertheless, the original conclusion that secure posts outnumber insecure posts in both categories still remains valid.

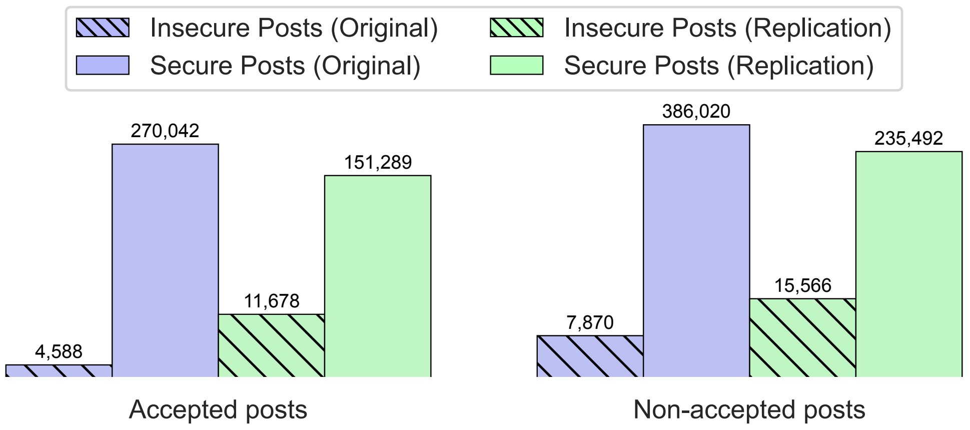


Figure 4.10: Ratio of insecure posts between accepted and non-accepted posts discovered by DICOS (logarithmic scale) as reported by the authors (Figure 7 in (49)) and found in our replication study.

Revisiting RQ3 Findings: The authors manually categorized the 788 insecure posts containing all three types of changes by their weakness types. Figure 4.11 presents a comparison between the authors' original results (refer to Figure 8 in [49]) and the replication findings for the types of insecure code snippets with all three types of changes (i.e., changes in security-sensitive APIs, security-related keywords, and control flows). They reported eight types of insecure code snippets, with undefined behavior (42%) being the most common. Overall, there is a significant decrease in the number of snippets, for instance, *Undefined behavior* (367 ↘ 16), *null-terminated string issue* (175 ↘ 10) and out-of-bounds error (17 ↘ 0).

The original conclusions about the weakness types no longer hold. For the newer data set version, the types of weaknesses and their frequencies have shifted.

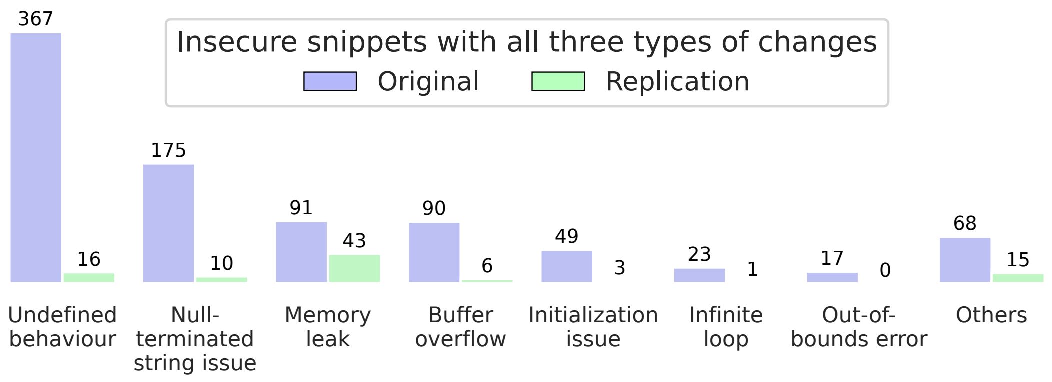


Figure 4.11: Types of discovered insecure code snippets with three types of changes discovered by DICOS as reported by the authors (Figure 8 in [49]) and found in our replication study.

4.5.3 Case Study 3 & 4: Stack Overflow Considered Harmful & Helpful

Fischer et al. [31] conducted a data-driven study (**R4**: ✘) of crypto API misuse in Java snippets. They classified snippets of the *March 2016* Stack Exchange data set into *secure* or *insecure* using a custom-built machine learning classifier. Through static analysis, they detected insecure snippets reused in Android apps (**R3**: ✪, **P**).

Unfortunately, the authors could only make their set of labeled snippets publicly available (**R1**: ◐) and did not mention how they detected the language of snippets (**R2**: **N/A**). Re-implementing their approach is challenging because their pipeline involves outdated and custom-built components. For instance, JavaBaker [106] is no longer functional and maintained, and we were unable to install the Partial Project Analysis tool [23]. In a follow-up study, Fischer et al. [36] proposed solutions to help developers use more secure code from Stack Overflow. The authors again only published their dataset of labeled code snippets (**R1**: ◐) and did not indicate how they detected the language of code snippets (**R2**: **N/A**). The study neither considered the participation of developers (**R4**: ✘) nor code reuse from Stack Overflow (**R3**: ✘). The authors used the *March 2018* Stack Exchange data set and a machine learning classifier to identify secure and insecure snippets. The training dataset and classifier are also unavailable, and the pipeline also includes outdated components, some identical to the previous study. As for their previous study, these challenges made re-implementing their approach infeasible.

Instead of re-implementation, we relied on the labeled datasets the authors made available to study **if** code evolution in those datasets could affect their findings. However, without their machine learning classifier, we cannot fully assess **how** their findings might be affected by code evolution since we cannot classify new snippets added on Stack Overflow after their study in a scalable, reasonable manner.

The first labeled dataset from the 2016 study includes 4,019 security-related snippets. The second dataset from the 2018 study includes 16,343 security-related snippets. We refer to these as DS_{2016} and DS_{2018} respectively (see Table 4.7). Unfortunately, DS_{2016} lacks the post IDs of the snippets, making it impossible to track snippet evolution. To

Dicos with SOTorrent20	**Dicos with SOTorrent22**
Data Collection	
The authors extracted 987,367 C/C++ posts and 970,916 Android posts, totaling 1,958,283 answer posts.	We extracted 1,460,627 C/C++ posts and 1,339,692 Android posts, totaling 2,800,319 answer posts. This is a 43% increase in the number of C/C++ and Android answer posts created on Stack Overflow after December 2020.
After filtering out single-version posts, the authors collected 668,520 multi-version answers, which they used to evaluate DICOS.	We collected 1,046,052 multi-version answers after filtering out single-version posts.
The authors found 12,458 insecure posts; 8,941 insecure C/C++ posts, and 3,517 insecure Android posts.	In contrast, we found 30,359 insecure posts; 22,167 insecure C/C++ posts and 8,192 insecure Android posts.
DICOS has 91% precision, 93% recall, and 89% accuracy.	We observed 11% precision, 87% recall and 32% accuracy.
RQ2 *Are accepted answer posts more secure than non-accepted posts?*	
The ratio of insecure posts was almost the same between accepted (1.67%) and non-accepted (1.99%) posts.	The ratio of insecure posts is almost the same between accepted (92.83%) and non-accepted (93.7%) posts.
RQ3 *What types of insecure code snippets were discovered?*	
The most prevalent type of insecure code snippets was *undefined behavior*, accounting for 42% of the total;	The most prevalent type of insecure code snippets was *memory leak*, accounting for 39.25% of the total;
Authors observed *Null-terminated string issue* as the second most prevalent security weakness.	In contrast, we found three security weaknesses as the second most prevalent weaknesses: *Undefined behavior*, *Out-of-bounds error*, and *Others*.

Table 4.6: Comparison of the results by Hong et al. (49) and our replication using *SOTorrent22*.

try resolving this, we indexed all Android posts and their snippet versions in Apache Solr [7], using the snippets as search strings to find matching posts. This process identified post IDs for 2,370 (59%) snippets of the original data set. The DS_{2018} dataset includes post IDs for the tracking of snippet evolution.

4.5.3.1 Evolution of Labeled Code Snippets

We used the *StackExchange23* data set to study the evolution of snippets in the labeled datasets of both studies. We collected all snippets from the DS_{2016} and DS_{2018} datasets that were revised in *StackExchange23* and manually labeled each edit. We tracked the evolution of the DS_{2016} snippets from 03/2016 to 09/2023 (denoted as I_{16-23}), i.e., seven years of changes. Figure 4.12 in exemplifies an answer post with two snippets that evolved twice in I_{16-23}. For the DS_{2018} dataset, we examined a period of one year earlier (U_{17-18}) and five years later (U_{18-23}).

	All	Secure	Insecure
	DS_{2016} $(N = 2,370)$		
Questions	1,147	813 (70.9%)	334 (29.1%)
Answers	1,223	870 (71.1%)	353 (28.9%)
	DS_{2018} $(N = 16,343)$		
Questions	1,0294	4,189 (40.7%)	6,105 (59.3%)
Answers	6,049	2,057 (34.0%)	3,992 (66.0%)

Table 4.7: Composition of the DS_{2016} and DS_{2018} datasets

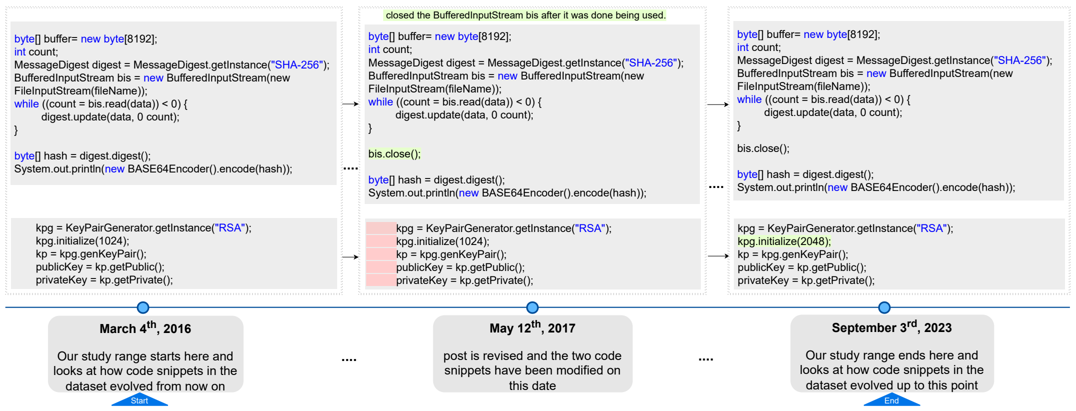
Table 4.8: Labels assigned during manual classification

Label	Meaning
1. Cosmetic	Does not affect the snippet functionality, e.g. identifier renaming, comments, code formatting
2. Functional	Affect functionality, but not security-relevant, e.g. adding test cases or refactoring
3. Security-relevant	Changes to code segments handling security-relevant data or interacting with a security API, e.g., changing security API, adding encryption, changing key size. Fixes to bugs and vulnerabilities also fall under this category.
4. Security label change	Changes to snippets that change the security classification from *insecure* to *secure* or vice versa according to the definition by Fischer et al. [31].
5. Bad classification	Coders disagree with the original security classification in the dataset by Fischer et al.
6. Post deleted	Post has been deleted from Stack Overflow

Labeling Two researchers independently assigned each edit one of four labels (see Table 4.8). They then discussed and resolved conflicts. We excluded posts assigned labels 5 or 6 for data sanitization. The researchers reached a high level of agreement (Krippendorff's Alpha [61] of $\alpha = 0.96$).

4.5.3.2 Results and Findings

Table 4.9 shows the results of classifying the posts containing code snippets that evolved within the selected time frames. Our analysis reveals that only **nine** code snippets from the DS_{2016} dataset labeled by Fischer et al. evolved between 2016 and 2023 with a security-relevant label change (see highlighted row in Table 4.9). Initially labeled insecure by Fischer et al., the vulnerabilities in these nine snippets were fixed during the seven years following Fischer et al.'s data collection. Similarly, if Fischer et al. [36] would have performed their data collection a year earlier (U_{17-18}), **three** snippet labels would have changed. If they would have crawled Stack Overflow five years later



```
byte[] buffer= new byte[8192];
int count;
MessageDigest digest = MessageDigest.getInstance("SHA-256");
BufferedInputStream bis = new BufferedInputStream(new
FileInputStream(fileName));
while ((count = bis.read(data)) < 0) {
        digest.update(data, 0 count);
}

byte[] hash = digest.digest();
System.out.println(new BASE64Encoder().encode(hash));
```

```
      kpg = KeyPairGenerator.getInstance("RSA");
      kpg.initialize(1024);
      kp = kpg.genKeyPair();
      publicKey = kp.getPublic();
      privateKey = kp.getPrivate();
```

closed the BufferedInputStream bis after it was done being used.

```
byte[] buffer= new byte[8192];
int count;
MessageDigest digest = MessageDigest.getInstance("SHA-256");
BufferedInputStream bis = new BufferedInputStream(new
FileInputStream(fileName));
while ((count = bis.read(data)) < 0) {
        digest.update(data, 0 count);
}

bis.close();

byte[] hash = digest.digest();
System.out.println(new BASE64Encoder().encode(hash));
```

```
      kpg = KeyPairGenerator.getInstance("RSA");
      kpg.initialize(1024);
      kp = kpg.genKeyPair();
      publicKey = kp.getPublic();
      privateKey = kp.getPrivate();
```

```
byte[] buffer= new byte[8192];
int count;
MessageDigest digest = MessageDigest.getInstance("SHA-256");
BufferedInputStream bis = new BufferedInputStream(new
FileInputStream(fileName));
while ((count = bis.read(data)) < 0) {
        digest.update(data, 0 count);
}

bis.close();

byte[] hash = digest.digest();
System.out.println(new BASE64Encoder().encode(hash));
```

```
kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(2048);
kp = kpg.genKeyPair();
publicKey = kp.getPublic();
privateKey = kp.getPrivate();
```

March 4th, 2016
Our study range starts here and looks at how code snippets in the dataset evolved from now on
Start
....
May 12th, 2017
post is revised and the two code snippets have been modified on this date
....
September 3rd, 2023
Our study range ends here and looks at how code snippets in the dataset evolved up to this point
End

Figure 4.12: The evolution of an answer post in DS_{2016} between March 4^{th}, 2016 and September 3^{rd}, 2023. Two additional versions have been created during this time interval. The first version shows the post on March 4^{th}, 2016, labeled by Fischer et al. as secure (top snippet) and insecure (bottom snippet). The post was revised on May 12^{th}, 2017, and the two code snippets were edited. The top code snippet undergoes a **security-relevant** edit because a resource leak bug was fixed. The edit to the bottom code snippet is merely **cosmetic**. In the final revision of the post, only the bottom code snippet is edited. This edit is a **label change** because the code snippet was originally labeled as insecure by Fischer et al. because the key size is considered weak, and the edit changes to a stronger key size in the final revision. In our classification, we label the top code snippet as a security-relevant change and the bottom code snippet as a security **label change** (see also Table 4.8).

(U_{18-23}), **eight** snippet labels would have been different. The limited number of fixes for insecure cryptographic API usage in these snippets suggests that such corrections are rare, potentially because of the particular niche expert topic of crypto API misuse in Java/Android code.

Snippets identified as vulnerable by Fischer et al. [31, 36] remained vulnerable despite code evolution. We postulate that this may be due to the particular niche topic of crypto API usage that requires domain experts to identify and fix such vulnerabilities.

However, since we do not have access to their code classifier, we cannot measure the impact of *new* snippets *added* to Stack Overflow after the data sets were collected. Therefore, while our analysis supports the stability of their results within the original datasets, the potential impact of new code snippets on the broader conclusions remains unknown, e.g., if the fraction of vulnerable snippets remains stable. To estimate the potential impact of new code snippets, we counted all posts added after the original studies that shared meaningful tags with those identified by Fischer et al. Since Fischer et al. studied the security of Java or Android code snippets, we counted the frequency of tags co-located with these two tags within the data set by Fischer et al. and two researchers identified the top 50 crypto-related co-located tags. These tags include *encryption*, *aes*, *ssl*, or *cryptography*. Figure B.22 and Figure B.23 in Appendix B.5 provide more details. Figure 4.13 depicts the number of posts created each month that

Table 4.9: Categories of posts containing code snippets that evolved in different periods in the two datasets.

Label	U_{17-18}	U_{18-23}	I_{16-23}
1. Cosmetic	107	443	256
2. Functionality	25	25	22
3. Security-relevant	35	24	44
4. Security-label	3	8	9
Σ	*170*	*500*	*331*
5. Wrong label	2	2	6
6. Post deleted	0	0	1

U_{17-18} and U_{18-23} for DS_{2018}; I_{16-23} for DS_{2016}

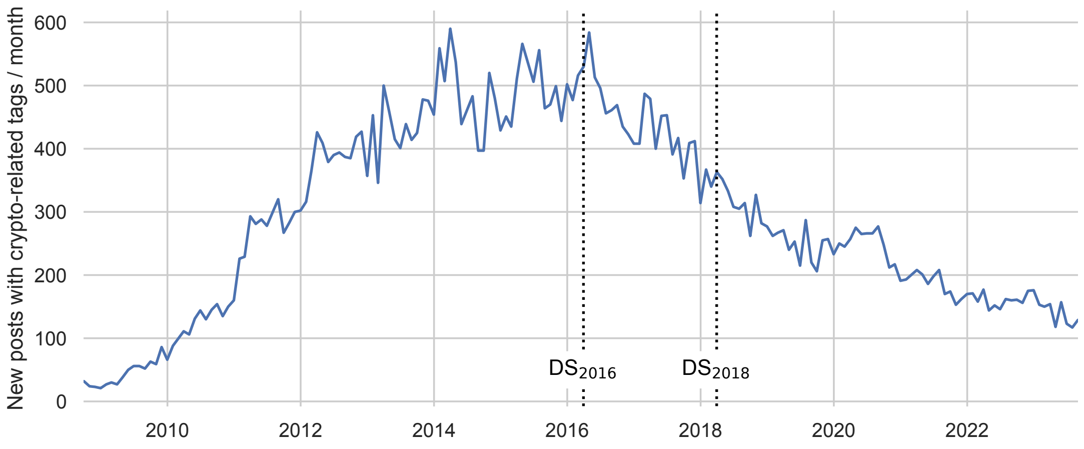


Figure 4.13: Nr. of monthly created posts with *Java* and/or *Android* tag plus at least one top-50 crypto-related tag. Vertical lines indicate data collection points by Fischer et al. (31, 36)

are tagged with *Android* and/or *Java* plus at least one of the top 50 crypto-related tags. We found 24,777 new answers containing those tags were added after DS_{2016}, corresponding to a growth of 87.6% to the 28,274 posts created before 03/2016. Using the crypto-tags as a proxy for new potentially vulnerable snippets, together with the stability of Fischer et al.'s original results, this growth indicates that Fischer et al.'s results likely form a lower bound for the number of vulnerable Java and Android snippets about crypto APIs in today's Stack Overflow. In our opinion, this is the most viable and fair approach to understanding the impact of code evolution on their findings.

Without code artifacts, we cannot measure whether the fraction of vulnerable snippets remains stable. Using tags as a proxy, we estimate that the number of vulnerable snippets has increased since Fischer et al.'s study.

4.5.4 Case Study 5: Snakes in Paradies

Rahman et al. [82] investigated Python code snippets posted on Stack Overflow, aiming to characterize the prevalence of insecure Python-related coding practices. The authors used the SOTorrent18 dataset, released in 09/2018, to empirically answer the following research questions: **RQ1**: *How frequently do insecure coding practices appear in Python-related Stack Overflow answers?* **RQ2**: *How does user reputation relate with the frequency of insecure Python-related coding practices?* **RQ3**: *What are the characteristics of Python-related questions that include answers with insecure code practices?*

The authors did not investigate the reuse of Python snippets from Stack Overflow (**R3**: ✘) and followed a purely data-driven approach to answer their research questions (**R4**: ✘).

4.5.4.1 Original Methodology

The authors focused exclusively on Python code snippets in answer posts in the SOTorrent18 dataset. If a question post, viewed more than once and with a score > 0, is attributed inside Python source files on GitHub, then all answers of the question are considered. Attributing a question post in Python projects on GitHub is the information the authors used to determine the language of snippets (**R2**: ✔). This resulted in 10,861 questions having 44,966 answers from which they extracted 529,054 code snippets, which forms their final dataset. To detect insecure coding patterns in code snippets, the authors used string matching to determine if a standard library or third-party Python API, which is known to be used in an insecure way, is found in a code snippet. The authors used six categories to group the insecure Python APIs they considered in their study. The six categories and their corresponding insecure coding pattern are describe in Table II of the original paper.

4.5.4.2 Implementation

The authors published the source artifacts of their study but not their data artifact (**R1**: ◑). We used the published source artifact[9] for replication, however, we needed to implemented means to collect the dataset from SOTorrent22 ourselves. This implementation was trivial since the authors extensively described their data collection approach in the paper. We tested this implementation using SOTorrent18, the same version used by the authors, and we were able to come to the same number of code snippets as reported in the original paper.

4.5.4.3 Replication

We replicated their findings using *SOTorrent22*, released four years after the original dataset version. After applying the authors filtering criteria, we obtained 12,095 questions containing 72,202 answers, of which 10,140 were accepted answers. This means that the number of code snippets matching the authors filtering criteria has dropped since 2018: 529,054 ↘ 239,575.

Revisiting RQ1 Findings: Our findings regarding the number of questions with at least one insecure answer differ significantly: 18.1% (out of 10,861) dropped to 4.9%

(out of 12,095). Similarly, the percentage of accepted answers containing at least one insecure snippet also decreased: 9.8% (out of 7,444) ↘ 2.2% (out of 10,139). Although the vulnerability rankings from the original study remain unchanged, we observed a shift in the number of affected snippets: *code injection* increased (2,319 ↗ 5,734), while *insecure cipher* (564 ↘ 356), *insecure connection* (624 ↘ 276), and *data serialization* (153 ↘ 140) all dropped. Like the original study, no snippets were impacted by XSS vulnerabilities.

> We found fewer insecure answers and accepted answers compared to the original, with percentages dropping significantly. Vulnerability rankings remained consistent, but there were notable shifts in affected snippets, including an increase in *code injection* cases and decreases in others. No XSS vulnerabilities were found, as in the original study.

Revisiting RQ2 Findings: The authors answered this question by computing the *normalized* reputation score of users that contributed at least one insecure code snippet with those that contributed answers with no insecure code snippets. The normalization was required to reduce bias since long-time Stack Overflow users tend to have higher reputation points. Using the Mann-Whitney U and Cliff's Delta non-parametric tests, the authors found no significant difference between answer providers with high and low reputation, suggesting that both are equally likely to introduce insecure code snippets. We also found no significant difference between the two user groups, however, we observed different $p-$value (0.9 ↗ 6.2) and cliff's delta (0.01 ↗ 0.03) values.

Revisiting RQ3 Findings: The authors employed Latent Dirichlet Allocation (LDA)-based topic modeling to group questions associated with an insecure answer. They found that answers to questions associated with **web**, **string** and **RNG** topics contains at least one insecure code snippet. In contrast, we found that answers to questions related to **web** topics are no longer associated with insecure code snippets but questions associated with **string** and **RNG** topics still have answers containing at least one insecure code snippet.

> We found that user reputation does not influence the likelihood of posting insecure code snippets, confirming the original observation. However, we discovered that the association between *web* topics and insecure code snippets is no longer valid.

4.5.5 Case Study 6: Mining Rule Violations in JavaScript

Campos et al. [32] investigated the prevalence of violations in JavaScript code snippets on Stack Overflow, aiming to answer the following research questions:

RQ1: *How commonplace are rule violations in JavaScript code snippets?*

RQ2: *What are the most common rules violated in JavaScript code snippets?*

RQ3: *Are JavaScript code snippets flagged with possible errors being reused in GitHub projects?*

We replicated the study for **RQ1** and **RQ2** to assess if the findings remain valid today. We omitted RQ3, as it concerns code reuse from Stack Overflow, while our research focuses on the evolution of code within Stack Overflow.

4.5.5.1 Original Methodology

The authors used a data-driven approach to answer their research questions (**R4**: ✘) and focused on accepted answers of questions with JavaScript tags (**R2**: ✔) in the SOTorrent18 dataset. In the event that a code snippet in an accepted answer has multiple versions, the latest version of the snippet is selected, resulting in 336,643 code snippets. Using the ESLint static analysis tool, the authors found that **all code snippets** in their data set contains rule violations, with stylistic issues been the most prevalent violation accounting for 82.9% of the total. Relying on attribution to detect code reuse (**R3**: ✔), they found 36 of the JavaScript code snippets containing violations to be reused in 845 GitHub projects.

4.5.5.2 Implementation

The authors made their source code and data publicly available (**R1**: ✪), enabling us to replicate their findings on a newer dataset. However, we found a discrepancy between their paper's data collection method and the released dataset. Although they claimed to discard snippets with fewer than 10 LoC, their dataset includes **42,158** snippets with nine LoC. Since the script used for data collection wasn't shared, reproducing the exact number of code snippets from SOTorrent18 using the approach discussed in the paper was not possible. Similarly, when we attempted to reproduce the authors findings for their **RQ1** using their own source code and dataset of code snippets, we could not come to their conclusion that *no code snippet in their dataset were free of violations*. Instead, we found 153,159 (45.5% of 336,643) code snippets that only contains parse errors without any rule violations.

We reached out to the authors for clarification and learned that they chose a minimum LoC of nine and included parse errors as violations. Since our replication study follows the same methodology as the original, we adhered to their clarification by setting the LoC threshold to nine and counting parse errors as violations.

4.5.5.3 Replication

We replicate their findings for **RQ1** and **RQ2** using *SOTorrent22*, released four years after the dataset version used in their study. The findings are summarized below.

Revisiting RQ1 Findings: The original study reported that no JavaScript code snippet was free of violations, but our replication found nine snippets without any violations. Additionally, the number of violations in JavaScript code snippets increased from 5,587,357 to 7,385,044, with the average violations per snippet rising from 11.94 to 28.8.

The observation that no single JavaScript code snippet on Stack Overflow is free of violations no longer holds true and we found more violations in code snippets than originally reported.

Revisiting RQ2 Findings: The original study grouped violations into six categories and reported the top three most common rule violation for each category. Below, we compare the authors' original findings with our replication results.

Stylistic Issues: Violations increased by 28.4% (from 4,632,348 to 5,946,283), with the three most common violations in this category being: (1) *semi*: 1,477,808 $\nearrow$ 1,990,461, (2) *quotes*: 700,770 $\nearrow$ 1,072,468, and (3) *no-trailing-spaces*: 374,012 to 473,294.

In the original study, the authors found and removed 3,461,739 violations related to indentation rule violations from this category, reasoning that multiple snippets were merged into a single file. We followed the same approach in the replication and removed 4,910,529 indentation violations.

Variable: Violations increased by 28.7% (from 787,824 to 1,013,612), with the top issues being: (1) *no-undef*: 719,679 $\nearrow$ 913,103, (2) *no-unused-vars*: 67,816 $\nearrow$ 100,124, and (3) *no-undef-init*: 150 $\nearrow$ 189.

Best Practices (BP): Violations increased by 216.5% (from 57,578 to 182,232), with the most common being: (1) *eqeqeq*: 53,321 $\nearrow$ 66,664, (2) *no-multi-spaces*: 54,768 $\nearrow$ 50,352, (3) *no-redeclare*: 18624, and (4) *curly*: 14,989 $\nearrow$ 18,292.

The *eqeqeq* violation is now the most common violation, while *curly* dropped to fourth place.

Possible Errors (PE): Violations increased by 39% (6,303 $\nearrow$ 8,762), with the most common being: (1) *no-irregular-whitespace*: 2,037 $\nearrow$ 2,196, (2) *no-cond-assign*: 910 $\nearrow$ 1,196, and (3) *no-dupe-keys*: 874, which replaced *no-unreachable* as the third most common violation.

Node.js/Common.js: Violations increased by 27.5% (from 3,304 to 4,211), with the most common violations being: (1) *handle-callback-err*: 2,855 $\nearrow$ 3,650, (2) *no-path-concat*: 444 $\nearrow$ 555, and (3) *no-new-require*: 5 $\nearrow$ 6.

ECMAscript 6: Violations increased by 139.8% (from 548 to 1,314), with the most common being: (1) *template-curly-spacing*: 164 $\nearrow$ 516, (2) *no-useless-constructor*: 154 $\nearrow$ 238, and (3) *no-const-assign*: 129, replacing *no-this-before-super* as the third most common violation.

Parse Errors: In our replication study, we found 267,795 code snippets with *parse errors* but no violations. While the original study included parse errors in the total violation count, we introduced a seventh category to separately list code snippets with only parse errors.

The number of violations in each category has risen, rendering the original findings obsolete. Additionally, the ranking of specific violations within certain categories has changed.

4.6 Limitations and Challenges

Before concluding with an overview of related works (Section 4.7) and recommendations for future works (Section 4.8), we briefly discuss the limitations of our study and discuss basic challenges for studying the security of Stack Overflow snippets.

Quality of original studies. This study replicates prior research using the original methods on a newer dataset. Consequently, any flaws or biases in the original studies are also reflected in our replication.

Generalizability of results. Our study specifically targeted research investigating the security properties of Stack Overflow code snippets. Thus, our findings may not apply to Stack Overflow-based studies that explore other aspects, such as user behavior, limiting the generalizability of our conclusions.

Dependence on available artifacts. The quality and availability of the artifacts by the original studies posed another limitation. If a published artifact inaccurately implemented a study's methodology, these inaccuracies carried over into our replication. Despite our best efforts to reuse the same source artifacts, any errors or bugs in the original implementation may have impacted our results. Therefore, the reliability of our replication study is inherently linked to the accuracy and completeness of the original research artifacts.

However, the biggest challenge we faced in our study was the lack of code and data artifacts from prior research (see Table 4.2), which forced us to re-implement the original methodology in various case studies. This was not always possible (e.g., training a machine learning-based classifier without access to the original data) and can be error-prone (e.g., replacing deprecated toolchains).

Language detection. A general challenge for studies on Stack Overflow is determining the programming language of code snippets. Related work relied on post tags or the Guesslang tool (see **R2** in Table 4.2). We evaluated these approaches briefly and found neither approach is reliable (see Appendix B.6 for details). In comparison, ChatGPT outperformed either tool, but it is neither scalable in data set size nor economically reasonable for our replication studies to create a ground truth of snippet languages. Thus, future work could create a fine-tuned LLM to replace Guesslang for the language detection task (see Section 5.1.3 for details).

Security classification. Several studies rely on a security classification of code snippets by either building/using dedicated tools for specific languages (**D2** in Table 4.2) or, in the absence of a generic code security classifier, leveraging the context of snippets (i.e., comments and commit messages; **D4** in Table 4.2). We used the tool we described in Chapter 3 from the latter category. However, this tool has a potentially high false positive rate due to its keyword-based detection. Future work could investigate a more reliable context-based tool, e.g., taking inspiration from other NLP-based solutions [45, 4] (see Section 5.1.2 for details).

4.7 Related Works

We briefly present related works that explore meta-research studies and investigate the evolution of data.

4.7.1 Meta-research

Meta-research is an important tool for evaluating and improving research practices. It helps to identify which research methods provide reliable and reproducible answers. Ioannidis [51] investigated prior research findings and found that most published research turned out to be false. One factor contributing to this crisis is the lack of code and data to verify research claims. Ioannidis et al. [52] introduced a framework that serves as a benchmark to holistically evaluate and improve research practices to make research results more reliable. They identified methods, reporting, reproducibility, evaluation, and incentives as areas of interest for research practices. Demir et al. [24] examined the reproducibility and replicability of web measurement studies. They discovered that many studies lacked proper documentation of their experimental setups, which is essential for accurately reproducing and replicating results. In particular, they found that even slight variations in experimental setup could lead to significant differences in results. The authors recommended proper documentation and adopting standardized practices to make web measurement findings more reliable. Weber et al. [112] conducted a study of benchmarking studies to develop guidelines to help computational scientists conduct better benchmarking studies. While their guidelines are for a different target group than ours, some translate to our work. For instance, the guideline on selecting datasets and reproducible research best practices can be transferred, as it shows that security studies focused on Stack Exchange datasets should be measured using different dataset versions and that the code and data for studies should be made publicly available to facilitate replication studies.

4.7.2 Dataset Evolution

Ceroni et al. [17] investigated Wikipedia's dynamic nature to examine how its content changes over time, specifically looking at page edits. Their research highlights the significance of understanding content evolution to evaluate the quality and accuracy of information on Wikipedia. In a follow-up, Tran et al. [108] leverage the evolution of data on Wikipedia to analyze the history of user edits to extract and represent complex events. Fetterly et al. [33] provides a detailed investigation of how web pages evolve over time to understand how that evolution might impact crawling and indexing processes for the web. Their study shed light on the importance of understanding the evolution of web pages for improving web crawling and indexing processes. Jallow et al. [P1] is closely related to us. They studied the impact of evolving Stack Overflow code snippets on software developers. In contrast, we focus on how code evolution might impact prior research findings.

Conducting time-series analysis to better understand data that can change over time is explained in the textbooks of several other research disciplines, such as economics [44], environmental science [113], medicine [91], finance [109], social sciences [16], or engineering [67].

4.8 Conclusion

Time-Series Analysis. Our systematization showed that security-focused Stack Overflow studies relied on specific dataset versions and specific, non-stationary aspects thereof. Among the 42 relevant papers we identified, only Ragkhitwetsagul et al. [81] discussed the potential impact of evolution on their results. Four works [49, 90, 118, 71] use code evolution in their methodology but do not discuss its consequences for their findings. With our six replication studies, we revealed that the findings of four studies are no longer valid for newer dataset versions. This does *not* mean that the results of prior research are wrong—but missing context. Researchers are advised to provide additional context around their results by treating the Stack Overflow dataset as the time-series data it is. Considering the temporal dimension of posts and conducting trend analysis can help better understand such results, e.g., whether issues are recurring or stable, and can address temporal dynamics, such as differentiating short-lived trends from long-term changes. While it is unrealistic to predict future changes with certainty, considering multiple dataset versions and their aspects' stationarity provides insights into how changes on Stack Overflow might impact the validity of findings. For example, we found that CWE types of C/C++ code snippets shifted over time while crypto API misuse in Java snippets seemed stable or even increasing. Trend analysis is common in other research disciplines, where drawing conclusions from a single observation would be insufficient, if not misleading. This is particularly true in fields such as economics [44] (studying variables like GDP, inflation, or stock prices), environmental science [113] (analyzing climate data or temperature changes), or social sciences [16] (examining voting patterns, crime rates, or demographic shifts). We suggest that our discipline adopt such a longitudinal analysis methodology when studying data that changes over time.

Open Science Best Practices. Although not a goal at the outset of our work, we want to re-iterate some best practices for open science and reproducibility that we found disregarded during our replication studies. We found that many artifacts were not or only partially available (on request) or non-functional. Making code and data available on request is often insufficient [112], and experience shows that paper authors can often ignore such requests. Thus, our work underlines the need to establish better efforts for open science, and we welcome the recent steps to introduce artifact evaluation committees and require artifacts for accepted papers. Moreover, we also found that, in most cases, the information provided in the papers was insufficient to allow others to re-implement the methods when artifacts were unavailable. Both the ACM SIGPLAN Empirical Evaluation Guidelines [92] and van der Kouwe et al.'s benchmarking flaws [60] remark that lack of such information leads to a lack of reproducibility and can hinder scientific progress. Thus, we urge researchers to report the software versions used in their methods. Containerization tools, like Docker, to encapsulate software environments and preserve package versions and dependencies could ease replication efforts and help with artifact releases.

5
Conclusion

When developing software, developers often reuse components from existing sources, such as software libraries, to save time. However, not every problem requires integrating an entire library, especially when the solution is a trivial one. An alternative approach used by developers is copying functional code snippets, often from platforms like Stack Overflow. While many studies have shown that developers reuse code snippets from Stack Overflow, including potentially vulnerable ones, it remains unclear whether they are aware of the evolution of these snippets to address security issues. Additionally, it is uncertain whether researchers using Stack Overflow data for studies consider the evolution of code snippets and their context, which could provide more meaningful insights around research findings.

In this dissertation, we presented a line of work that studied the evolution of code snippets and their surrounding context on Stack Overflow, examining how these changes impacts software developers (Chapter 3) and research findings (Chapter 4). Our work on code evolution and its effects on software developers reveals that developers are generally unaware that code snippets on Stack Overflow change over time. Furthermore, we discovered that developers do not track the updates to copied code snippets, even when the posts containing those snippets are properly attributed in their projects. We found that 2,405 code snippet versions reused in 22,735 distinct GitHub source files were outdated, affecting 2,109 GitHub projects. Among those outdated snippet versions, we verified 26 to have a security-relevant update on Stack Overflow that fixes a known vulnerability. The fixes to those vulnerabilities on Stack Overflow were not reflected in 43 highly popular, non-forked open-source projects to whose maintainers we disclosed our findings. Further analyzing the 43 affected projects reveals that it took, on average, 1,060±506 days (max. 3,303 days) from when an insecure Stack Overflow code snippet is committed to a GitHub project until the time a Stack Overflow comment raising a security warning is made. However, as soon as a security warning is raised, it takes, on average, 296±200 days (max. 1,516) for a security issue in a code snippet to be fixed. For a time difference this long, it would be non-trivial for developers to manually track Stack Overflow for updates or be aware of (or react to) security discussions around a piece of code they reused such a long time ago.

The second part of this dissertation focused on how prior research findings may be impacted by the evolution of code and surrounding context on Stack Overflow (Chapter 4). To understand whether prior findings are impacted, we surveyed the literature for papers studying the security properties of code snippets on Stack Overflow. This yielded 42 highly relevant papers, which we systematized according to the Stack Overflow aspects their methods relied on (e.g., programming language or the context of snippets). This systematization shows that the targeted programming language may affect the stability of results over time and that most works leverage some form of code classification, whose results can be immediately affected by code revisions. Further, we conducted a time series analysis to understand how code snippets and security- and privacy-related discussions on Stack Overflow evolve. Our data shows that programming languages trend differently regarding their overall number of added snippets and their ratio of security-relevant edits. Moreover, we found that the fraction of security-relevant comments on Stack Overflow steadily increased. As a result, studies focusing on particular programming languages will likely find a different landscape when conducted at various points in time. Together, these two insights provided an intuition about the reliability of prior cross-sectional studies in light of the evolution of Stack Overflow content. We conducted *six* replication studies of prior work [118, 49, 32, 82, 31, 36] using a more recent data set version to find concrete evidence for the impact of this evolution on research results. We found that the results of multiple works do not hold over time [118, 49, 32, 82] and provided recommendations for future research using Stack Overflow data. We suggest that researchers treat Stack Overflow data as time-series data and analyze their results as a trend model rather than a cross-sectional analysis, drawing inspiration from methodologies used in economics, environmental science, or medical research. This approach better contextualize findings, helping to distinguish whether the issues observed are short-term trends or long-term systemic problems.

5.1 Future research directions

The author of this dissertation envisions potential future research directions that build upon this line of work. Each of these directions is discussed below.

5.1.1 Better tool support for developers

While prior work [36, 110] made crucial contributions in providing tools to help developers reuse Stack Overflow code snippets more securely, those works treated code on Stack Overflow as *static* and thus only helped to mitigate current instances of insecure code reuse. For instance, available tools can flag a code snippet as secure. However, as our results show, the developer community on Stack Overflow can raise a security warning about such a snippet months or even years later. While software projects have tools (e.g., Dependabots) for managing library dependencies and keeping them up to date, there are no tools for managing dependencies on Stack Overflow snippets. A future research direction is to help software developers with tools to make it seamless for them to *monitor updates* on Stack Overflow by integrating search and monitoring functionality directly into their IDE. There are currently Visual Studio Code (VS Code) extensions [28, 27] that provide Stack Overflow search for code solutions inside VS Code. However, these tools only provide search functionality and neither record nor monitor Stack Overflow for bug/security fixes to copied code. Instead, they should support monitoring the Stack Overflow posts from which a developer copied code and inform developers about snippet updates and security discussions. One way to achieve this is for such a tool to record where a piece of code is copied from by attributing the original Stack Overflow post. Another option, which is a more involved, is for the tool to bridge the gap between a developer's code base and Stack Overflow to acquire insights from Stack Overflow to support developers whenever they use code that is present on Stack Overflow irrespective of whether the code was copied from Stack Overflow or not.

5.1.2 Security Classification

Regarding detecting security-relevant discussions and code changes, a more modern approach based on recent advances in natural language processing is required to alert developers better when security discussions and code fixes are available on Stack Overflow. This is because not all discussions and commit messages on Stack Overflow are security-relevant. The security filter used in this paper relies on keywords to identify security-relevant discussions and code fixes on Stack Overflow. However, this approach is limited since it errs on the safe side and over-reports security-relevant posts. A better, more recent approach to tackling this problem that is not keyword-restricted is Hark [45]. Hark leverages Natural Language Inference (NLI) together with classifier models based on Text-To-Text Transfer Transformer (T5) [80] to detect privacy and security-focused discussions from Google Play app reviews. Google Play app reviews and Stack Overflow post comments share some similarities since they are both short and directed towards addressing a specific issue in an app (in the case of the former) and in a code snippet (in the case of the latter). We believe a tool that leverages techniques such as Hark will enable developers to receive more fine-grained and meaningful security-focused discussions from Stack Overflow.

Finally, Baltes et al. [14] showed that not all code updates on Stack Overflow are meaningful, as they discovered that one-third of code updates on Stack Overflow are related to formatting. Their findings indicate a low noise-to-signal ratio for code updates related to bugs or security fixes. Accordingly, the tool should support tracking specific code changes so developers are only notified when certain code updates occur (e.g., when a bug or security issue is fixed). We believe this tool support will ultimately bring us closer to solving the problem of insecure code reuse from knowledge sharing platforms such as Stack Overflow.

5.1.3 Language Detection

During our analysis, we found that Guesslang is outdated and prone to false positives—for instance, it frequently misclassifies valid C/C++ snippets as Java. We briefly evaluated alternative methods, including post tags and Guesslang, and found that neither approach offers sufficient reliability (see Appendix B.6 for details). Although ChatGPT outperforms both in accuracy, it is not a scalable or cost-effective solution for large-scale language detection tasks.

We also examined `PLangRec` [101], a recent tool that performs better than Guesslang in language detection. While this tool is promising, it is only capable of detecting 10 programming languages, which restricts its applicability for generalized code snippet classification.

Given the central role of accurate language identification in studies involving Stack Overflow code snippets require a more robust and scalable solution. Future work should focus on developing fine-tuned large language models capable of high-accuracy, broad-coverage language detection. A scalable GenAI-based approach could serve as a replacement for current tools and dramatically improve the quality and efficiency of research relying on functional code snippets data.

5.1.4 Stack Overflow and Generative AI Tools

The rise of generative AI (GenAI) tools like GitHub Copilot and ChatGPT has changed how developers get help with programming assignments, leading to decreased traffic on platforms like Stack Overflow. This shift stems from GenAI's accessibility, rapid prototyping capabilities, and immediate suggestions [78]. A recent interview with developers reveals that GenAI tends to replace Stack Overflow primarily for resolving simple or repetitive programming issues. Conversely, developers continue to value Stack Overflow for addressing more intricate or edge-case scenarios, where community discussion and expert input are indispensable. As a result, Stack Overflow's role may evolve into a more specialized, high-quality knowledge base. Moreover, since GenAI models are trained on its content, the longitudinal value of its content may increase.

A promising direction for future work, therefore, concerns the reuse of outdated code in the age of GenAI. As demonstrated in Chapter 3, open-source developers on GitHub frequently reuse outdated code snippets sourced from Stack Overflow, sometimes introducing security vulnerabilities into their projects. This issue is compounded by the fact that GenAI models are trained on large volumes of code from GitHub repositories—many of which include such reused snippets. As a result, there is a risk that GenAI tools may inadvertently propagate insecure or obsolete coding practices. These observations underscore the importance of developing methods to identify and mitigate outdated code suggestions in AI-assisted development environments. Addressing this challenge is crucial for improving the reliability and security of software generated with the aid of GenAI technologies.

Bibliography

Author's Papers for this Thesis

[P1] Jallow, A., Schilling, M., Backes, M., and Bugiel, S. Measuring the effects of stack overflow code snippet evolution on open-source software security. In: *45th IEEE Symposium on Security and Privacy (SP'24)*. IEEE, 2024.

[P2] Jallow, A. and Bugiel, S. Stack overflow meets replication: security research amid evolving code snippets. In: *34th USENIX Security Symposium (USENIX Sec'25)*. USENIX Association, 2025.

Other references

[1] Abdalkareem, R., Shihab, E., and Rilling, J. On code reuse from stackoverflow: an exploratory study on android apps. *Information and Software Technology* 88 (2017), 148–158.

[2] Acar, Y., Stransky, C., Wermke, D., Weir, C., Mazurek, M., and Fahl, S. Developers need support, too: a survey of security advice for software developers. In: *Proc. Cybersecurity Development (SecDev'17)*. IEEE Computer Society, 2017.

[3] Ahmad, M. and Cinnéide, M. Ó. Impact of stack overflow code snippets on software cohesion: a preliminary study. In: *Proc. 16th International Conference on Mining Software Repositories (MSR'19)*. IEEE Press, 2019.

[4] Akgul, O., Peddinti, S. T., Taft, N., Mazurek, M. L., Harkous, H., Srivastava, A., and Seguin, B. A decade of Privacy-Relevant android app reviews: large scale trends. In: *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024.

[5] Alhanahnah, M. and Yan, Q. Towards best secure coding practice for implementing ssl/tls. In: *IEEE Conference on Computer Communications Workshops*. 2018.

[6] Almeida, L., Gonzaga, M., Santos, J. F., and Abreu, R. Rexstepper: a reference debugger for javascript regular expressions. In: *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings*. 2023.

[7] Apache Solr. *Apache Solr Website*. accessed 2019-12-09.

[8] *Artifact for Paper: Measuring the Effects of Stack Overflow Code Snippet Evolution.*

[9] *Artifact for Paper: Snakes in Paradies.*

[10] Backes, M., Bugiel, S., and Derr, E. Reliable third-party library detection in android and its security applications. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* CCS '16. Association for Computing Machinery, Vienna, Austria, 2016, 356–367.

[11] Bagherzadeh, M., Fireman, N., Shawesh, A., and Khatchadourian, R. Actor concurrency bugs: a comprehensive study on symptoms, root causes, api usages, and differences. *Proc. ACM Program. Lang.* 4, OOPSLA (Nov. 2020).

[12] Bai, W., Akgul, O., and Mazurek, M. L. A qualitative investigation of insecure code propagation from online forums. In: *2019 IEEE Cybersecurity Development (SecDev).* 2019, 34–48.

[13] Baltes, S. and Diehl, S. Usage and attribution of stack overflow code snippets in github projects. *Empirical Software Engineering* 24, 3 (June 2019), 1259–1295.

[14] Baltes, S. and Treude, C. Code duplication on stack overflow. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results.* ICSE-NIER '20. Association for Computing Machinery, Seoul, South Korea, 2020, 13–16.

[15] Baltes, S., Treude, C., and Diehl, S. Sotorrent: studying the origin, evolution, and usage of stack overflow code snippets. In: *Proc. 16th International Conference on Mining Software Repositories (MSR 2019).* 2019.

[16] Box-Steffensmeier, J. M., Freeman, J. R., Hitt, M. P., and Pevehouse, J. C. W. *Time Series Analysis for the Social Sciences.* Analytical Methods for Social Research. Cambridge University Press, 2014.

[17] Ceroni, A., Georgescu, M., Gadiraju, U., Naini, K. D., and Fisichella, M. Information evolution in wikipedia. In: *Proceedings of The International Symposium on Open Collaboration.* OpenSym '14. Association for Computing Machinery, Berlin, Germany, 2014, 1–10.

[18] Chakraborty, M. Does reusing pre-trained nlp model propagate bugs? In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM, 2021.

[19] Chen, F. and Kim, S. Crowd debugging. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ESEC/FSE 2015. ACM, 2015.

[20] Chen, L., Hou, S., Ye, Y., Bourlai, T., Xu, S., and Zhao, L. Itrustso: an intelligent system for automatic detection of insecure code snippets in stack overflow. In: *2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM).* 2019.

[21] Chen, M., Fischer, F., Meng, N., Wang, X., and Grossklags, J. How reliable is the crowdsourced knowledge of security implementation? In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* 2019, 536–547.

[22] Common Weakness Enumeration. *Common Weakness Enumeration.* 2022. URL: `https://cwe.mitre.org/`.

[23] Dagenais, B. *Partial Program Analysis for Eclipse.* 2022. URL: https://www.sable.mcgill.ca/ppa/ppa_eclipse.html.

[24] Demir, N., Große-Kampmann, M., Urban, T., Wressnegger, C., Holz, T., and Pohlmann, N. Reproducibility and replicability of web measurement studies. In: *Proceedings of the ACM Web Conference 2022.* 2022, 533–544.

[25] Derr, E., Bugiel, S., Fahl, S., Acar, Y., and Backes, M. Keep me updated: an empirical study of third-party library updatability on android. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* CCS '17. Association for Computing Machinery, Dallas, Texas, USA, 2017, 2187–2200.

[26] Diamantopoulos, T., Sifaki, M.-I., and Symeonidis, A. L. Towards mining answer edits to extract evolution patterns in stack overflow. In: *Proc. 16th International Conference on Mining Software Repositories (MSR '19).* IEEE Press, 2019.

[27] Extensions for Visual Studio Code. *StackFinder.* 2023. URL: https://marketplace.visualstudio.com/items?itemName=mark-fobert.stackfinder.

[28] Extensions for Visual Studio Code. *Stackoverflow Instant Search.* 2023. URL: https://marketplace.visualstudio.com/items?itemName=Alexey-Strakh.stackoverflow-search.

[29] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., and Smith, M. Why eve and mallory love android: an analysis of android ssl (in)security. In: *Proc. 19th ACM Conference on Computer and Communication Security (CCS '12).* ACM, 2012.

[30] Fahl, S., Harbach, M., Perl, H., Koetter, M., and Smith, M. Rethinking ssl development in an appified world. In: *Proc. 20th ACM Conference on Computer and Communication Security (CCS '13).* ACM, 2013.

[31] Felix, F., Böttinger, K., Huang, X., Christian, S., Yasemin, A., Michael, B., and Fahl, S. Stack overflow considered harmful? the impact of copy&paste on android application security. In: *Proc. 38th IEEE Symposium on Security and Privacy (SP '17).* IEEE Computer Society, 2017.

[32] Ferreira Campos, U., Smethurst, G., Moraes, J. P., Bonifácio, R., and Pinto, G. Mining rule violations in javascript code snippets. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).* 2019, 195–199.

[33] Fetterly, D., Manasse, M., Najork, M., and Wiener, J. A large-scale study of the evolution of web pages. In: *Proceedings of the 12th International Conference on World Wide Web.* WWW '03. Association for Computing Machinery, Budapest, Hungary, 2003, 669–678.

[34] Firouzi, E., Sami, A., Khomh, F., and Uddin, G. On the use of c# unsafe code context: an empirical study of stack overflow. In: *Proc. 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement.* 2020.

[35] Fischer, F., Stachelscheid, Y., and Grossklags, J. The effect of google search on software security: unobtrusive security interventions via content re-ranking. In: *ccs21*. 2021.

[36] Fischer, F., Xiao, H., Kao, C.-Y., Stachelscheid, Y., Johnson, B., Razar, D., Fawkesley, P., Buckley, N., Böttinger, K., Muntean, P., and Grossklags, J. Stack overflow considered helpful! deep learning security nudges towards stronger cryptography. In: *Proc. 28th USENIX Security Symposium (SEC' 19)*. USENIX Association, 2019.

[37] Fuller, W. *Introduction to statistical time series*. A Wiley publication in applied statistics. Wiley, 1976.

[38] Gao, Q., Zhang, H., Wang, J., Xiong, Y., Zhang, L., and Mei, H. Fixing recurring crash bugs via analyzing q&a sites (t). In: *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015.

[39] Ghanbari, A., Thomas, D.-G., Arshad, M. A., and Rajan, H. Mutation-based fault localization of deep neural networks. In: *Proc. 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 2024.

[40] Gharehyazie, M., Ray, B., and Filkov, V. Some from here, some from there: cross-project code reuse in github. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017.

[41] Gousios, G. The ghtorent dataset and tool suite. In: *Proc. 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, 2013.

[42] Guesslang. *Guesslang Documentation*. 2021. URL: `https://guesslang.readthedocs.io/en/latest/`.

[43] Haidar, O., Mircea, L., and Oscar, N. Mining frequent bug-fix code changes. In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 2014, 343–347.

[44] Hamilton, J. D. *Time Series Analysis*. Princeton University Press, 1994.

[45] Harkous, H., Peddinti, S. T., Khandelwal, R., Srivastava, A., and Taft, N. Hark: A deep learning system for navigating privacy feedback at scale. In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, 2469–2486.

[46] Holm, S. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics* 6, 2 (1979), 65–70.

[47] Hong, H. *Data Collection SQL Script*. 2022. URL: `https://github.com/hyunji-Hong/Dicos-public/blob/main/src/sql/collecting_allhisotyPost.sql`.

[48] Hong, H. *DICOS GitHub Repository*. 2022. URL: `https://github.com/hyunji-Hong/Dicos-public`.

[49] Hyunji, H., Seunghoon, W., and Heejo, L. Dicos: discovering insecure code snippets from stack overflow posts by leveraging user discussions. In: *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2021.

[50] Imai, H. and Kanaoka, A. Time series analysis of copy-and-paste impact on android application security. In: *13th Asia Joint Conference on Information Security (AsiaJCIS)*. 2018.

[51] Ioannidis, J. P. A. Why most published research findings are false. *PLOS Medicine* 2, 8 (Aug. 2005), null.

[52] Ioannidis, J. P. A., Fanelli, D., Dunne, D. D., and Goodman, S. N. Meta-research: evaluation and improvement of research methods and practices. *PLOS Biology* 13, 10 (Oct. 2015), 1–7.

[53] Islam, M. J., Pan, R., Nguyen, G., and Rajan, H. Repairing deep neural networks: fix patterns and challenges. In: *Proc. ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020.

[54] Jallow, A. and Bugiel, S. *SOTorrent Dataset Version 2020-12-31*. 2020.

[55] Jallow, A. and Bugiel, S. Stack overflow meets replication: security research amid evolving code snippets (extended version) (2025). arXiv: `2501.16948 [cs.CR]`.

[56] Jhoo, H. Y., Kim, S., Song, W., Park, K., Lee, D., and Yi, K. A static analyzer for detecting tensor shape errors in deep neural network training code. In: *Proc. ACM/IEEE 44th International Conference on Software Engineering (ICSE)*. 2022.

[57] Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. Do code clones matter? In: *2009 IEEE 31st International Conference on Software Engineering*. 2009.

[58] Kai, P., Sunghun, K., and James, W. E. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.* 14, 3 (June 2009), 286–315.

[59] Kitchenham, B. and Charters, S. Guidelines for performing systematic literature reviews in software engineering. 2 (Jan. 2007).

[60] Kouwe, E. van der, Heiser, G., Andriesse, D., Bos, H., and Giuffrida, C. Sok: benchmarking flaws in systems security. In: *2019 IEEE European Symposium on Security and Privacy (EuroSP)*. IEEE Computer Society, 2019.

[61] Krippendorff, K. *Content Analysis: An Introduction to Its Methodology (second edition)*. Sage Publications, 2004.

[62] Kwiatkowski, D., Phillips, P. C. B., Schmidt, P., and Shin, Y. Testing the null hypothesis of stationarity against the alternative of a unit root: How sure are we that economic time series have a unit root? *Journal of Econometrics* 54, 1-3 (1992), 159–178.

[63] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., and Kirda, E. Thou shalt not depend on me: analysing the use of outdated javascript libraries on the web. In: *Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA, Feb. 2017.

[64] Licorish, S. A. and Nishatharan, T. Contextual profiling of stack overflow java code security vulnerabilities initial insights from a pilot study. In: *21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE Computer Society, 2021.

[65] Licorish, S. A. and Wagner, M. Dissecting copy/delete/replace/swap mutations: insights from a gin case study. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '22. ACM, 2022.

[66] Liu, Y., Yan, Y., Sha, C., Peng, X., Chen, B., and Wang, C. Deepanna: deep learning based java annotation recommendation and misuse detection. In: *IEEE International Conference on Software Analysis, Evolution and Reengineering*. 2022.

[67] Ljung, L. *System identification: theory for the user*. Prentice-Hall, Inc., 1986.

[68] Madsen, M., Lhoták, O., and Tip, F. A model for reasoning about javascript promises. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017).

[69] Mahajan, S., Abolhassani, N., and Prasad, M. R. Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. ACM, 2020.

[70] Mahajan, S. and Prasad, M. R. Providing real-time assistance for repairing runtime exceptions using stack overflow posts. In: *IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2022.

[71] Manes, S. S. and Baysal, O. Studying the change histories of stack overflow and github snippets. In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 2021.

[72] Meng, N., Nagy, S., Yao, D., Zhuang, W., and Arango-Argoty, G. Secure coding practices in java: challenges and vulnerabilities. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, 372–383.

[73] Moradi Moghadam, M., Bagherzadeh, M., Khatchadourian, R., and Bagheri, H. μAkka: mutation testing for actor concurrency in akka using real-world bugs. In: *Proc. 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2023.

[74] Nadi, S., Krüger, S., Mezini, M., and Bodden, E. Jumping through hoops: why do java developers struggle with cryptography apis? In: *Proc. 38th International Conference on Software Engineering (ICSE'16)*. ACM, 2016.

[75] Nasehi, S. M., Sillito, J., Maurer, F., and Burns, C. What makes a good code example?: a study of programming q&a in stackoverflow. In: *Proc. 28th International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2012.

[76] Nishi, M. A., Ciborowska, A., and Damevski, K. Characterizing duplicate code snippets between stack overflow and tutorials. In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR '19. IEEE Press, Montreal, Quebec, Canada, 2019, 240–244.

[77] Pan, R. Does fixing bug increase robustness in deep learning? In: *Proc. ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings (ICSE)*. 2020.

[78] Peng, S., Kalliamvakou, E., Cihon, P., and Demirer, M. *The Impact of AI on Developer Productivity: Evidence from GitHub Copilot*. 2023. arXiv: `2302.06590 [cs.SE]`.

[79] PMD Source Code Analyzer Project. *Finding duplicated code with CPD*. 2022. URL: `https://pmd.github.io/latest/pmd_userdocs_cpd.html`.

[80] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* 21, 1 (Jan. 2020).

[81] Ragkhitwetsagul, C., Krinke, J., Paixao, M., Bianco, G., and Oliveto, R. Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering* 47, 3 (2021), 560–581.

[82] Rahman, A., Farhana, E., and Imtiaz, N. Snakes in paradise? insecure python-related coding practices in stack overflow. In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR '19. IEEE Press, Montreal, Quebec, Canada, 2019, 200–204.

[83] Rahman, M. An empirical case study on Stack Overflow to explore developers' security challenges. MA thesis. Kansas State University, 2016.

[84] Rahman, M. S. and Roy, C. K. An insight into the reusability of stack overflow code fragments in mobile applications. In: *IEEE 16th International Workshop on Software Clones*. 2022.

[85] Reinhardt, A., Zhang, T., Mathur, M., and Kim, M. Augmenting stack overflow with api usage patterns mined from github. In: *Proc. 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018.

[86] Ren, X., Sun, J., Xing, Z., Xia, X., and Sun, J. Demystify official api usage directives with crowdsourced api misuse scenarios, erroneous code examples and patches. In: *IEEE/ACM 42nd International Conference on Software Engineering*. 2020.

[87] Roy, C. K. and Cordy, J. R. Nicad: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: *Proc. 16th IEEE International Conference on Program Comprehension*. 2008.

[88] Schmidt, H., Aerssen, M. van, Leich, C., Benni, A., Al Ali, S., and Tanz, J. Copypastavulguard – a browser extension to prevent copy and paste spreading of vulnerable source code in forum posts. In: *Proc. 17th International Conference on Availability, Reliability and Security (ARES)*. ACM, 2022.

[89] Sebastian, B. and Markus, W. An annotated dataset of stack overflow post edits. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. GECCO '20. Association for Computing Machinery, Cancún, Mexico, 2020, 1923–1925.

[90] Selvaraj, M. and Uddin, G. Does collaborative editing help mitigate security vulnerabilities in crowd-shared iot code examples? In: *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '22. Association for Computing Machinery, Helsinki, Finland, 2022, 92–102.

[91] Shumway, R. and Stoffer, D. *Time Series Analysis and Its Applications With R Examples*. Vol. 9. 2011.

[92] *SIGPLAN Empirical Evaluation Guidelines.*

[93] Soni, A. and Nadi, S. Analyzing comment-induced updates on stack overflow. In: *Proc. 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, 2019.

[94] Stack Exchange, I. *Stack Exchange Data Dump).* accessed 2023-06-13.

[95] Stack Exchange Meta. *Academic Papers Using Stack Exchange Data.* 2022. URL: `https://meta.stackexchange.com/questions/134495/academic-papers-using-stack-exchange-data`.

[96] Stack Overflow. *How do I trim leading/trailing whitespace in a standard way?* 2022. URL: `https://stackoverflow.com/a/122721/8462878`.

[97] Stack Overflow. *How to make a copy of a file in android?* 2022. URL: `https://stackoverflow.com/a/9293885/8462878`.

[98] Stack Overflow. *How to pretty print XML from Java?* 2022. URL: `https://stackoverflow.com/a/1264912/8462878`.

[99] Stack Overflow. *Standard concise way to copy a file in Java?* 2022. URL: `https://stackoverflow.com/a/19542599/8462878`.

[100] Stack Overflow. *Stack Overflow Trends.* 2023. URL: `https://insights.stackoverflow.com/trends`.

[101] Stack Overflow. *Character-level deep-learning model to recognize the programming language of source code.* 2024. URL: `https://github.com/ComputationalReflection/PLangRec.git`.

[102] Stack Overflow. *Registered Stack Overflow Users.* 2024. URL: `https://data.stackexchange.com/stackoverflow/query/1873310/total-number-of-users`.

[103] Stack Overflow. *Revisions of Answer Post 14424800.* 2024. URL: `https://stackoverflow.com/posts/14424800/revisions`.

[104] Stack Overflow. *Stack Overflow Post History types.* 2024. URL: `https://data.stackexchange.com/stackoverflow/query/1892930/supported-post-history-types`.

[105] Stack Overflow. *Stack Overflow Post Types.* 2024. URL: `https://data.stackexchange.com/stackoverflow/query/1877252/posttypes`.

[106] Subramanian, S. *Java Snippet Parser.* 2022. URL: `https://github.com/siddhukrs/java-baker`.

[107] tool, O.-s. *Joern: The Bug Hunter's Workbench.* 2022. URL: `https://joern.io/`.

[108] Tran, T., Ceroni, A., Georgescu, M., Djafari Naini, K., and Fisichella, M. Wikipevent: leveraging wikipedia edit history for event detection. In: *International Conference on Web Information Systems Engineering.* Springer. 2014, 90–108.

[109] Tsay, R. *Analysis of financial time series.* 2. ed. Wiley series in probability and statistics. Wiley-Interscience, 2005.

[110] Verdi, M., Sami, A., Akhondali, J., Khomh, F., Uddin, G., and Motlagh, A. K. An empirical study of c++ vulnerabilities in crowd-sourced code examples. *arXiv:1910.01321* (2019). eprint: `arXiv:1910.01321`.

[111] Vocke, H. *We're switching to CommonMark.* `https://meta.stackexchange.com/questions/348746/were-switching-to-commonmark`. 2020.

[112] Weber, L. M., Saelens, W., Cannoodt, R., Soneson, C., Hapfelmeier, A., Gardner, P. P., Boulesteix, A.-L., Saeys, Y., and Robinson, M. D. Essential guidelines for computational method benchmarking. *Genome biology* 20 (2019), 1–12.

[113] Wilks, D. S. *Statistical methods in the atmospheric sciences.* Elsevier Academic Press, 2011.

[114] Yadavally, A., Nguyen, T. N., Wang, W., and Wang, S. (partial) program dependence learning. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* 2023.

[115] Yang, D., Martins, P., Saini, V., and Lopes, C. Stack overflow in github: any snippets there? In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR).* 2017.

[116] Yasemin, A., Michael, B., Sascha, F., Doowon, K., L, M. M., and Christian, S. You get where you're looking for: the impact of information sources on code security. In: *Proc. 37th IEEE Symposium on Security and Privacy (SP '16).* IEEE Computer Society, 2016.

[117] Ye, Y., Hou, S., Chen, L., Li, X., Zhao, L., Xu, S., Wang, J., and Xiong, Q. Icsd: an automatic system for insecure code snippet detection in stack overflow over heterogeneous information network. In: *Proceedings of the 34th Annual Computer Security Applications Conference.* ACSAC '18. ACM, 2018.

[118] Zhang, H., Wang, S., Li, H., Chen, T., and Hassan, A. E. A study of c/c++ code weaknesses on stack overflow. *IEEE Transactions on Software Engineering* 48, 07 (July 2022), 2359–2375.

[119] Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., and Kim, M. Are code examples on an online q a forum reliable?: a study of api misuse on stack overflow. In: *Proc. 40th International Conference on Software Engineering (ICSE'18)*. 2018.

[120] Zhang, Y., Chen, Y., Cheung, S.-C., Xiong, Y., and Zhang, L. An empirical study on tensorflow program bugs. In: *Proc. 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2018.

A
Appendix: Effects on Developers

A.1 Calibrating NiCad clone detector

We needed to find the optimal NiCad parameters for working with Stack Overflow code snippets. To this end, we calibrated NiCad's *threshold* and *minsize* configuration parameters, i.e., the minimum number of LoC and the type of clone (type-1, type-2, or type-3).

To measure the performance of different configuration settings, we extracted $3,824$ posts that are *directly attributed* in the SOTorrent dataset as a ground truth. We removed all attributed question posts from this set for two reasons: 1) developers are more likely to copy code snippets of answer posts [13]; 2) we found that many developers tend to wrongly attribute the question of the answer that contains the snippet. Of the $3,824$ answers, 364 did not contain any code snippets. From the remaining $3,460$ answers, we extracted $4,999$ code snippets after filtering out all snippets with less than five lines of code—the minimum code length for NiCad—resulting in a total of $2,656$ code snippets. Figure A.1 shows the distribution of the LoC of code snippets in the resulting attributed dataset. For each answer post in the dataset, we downloaded all the Java files on *GH* in which the corresponding answer was attributed. With this set of Stack Overflow posts and Java files, we measure the performance of our clone detection based on *precision* and *recall*.

Table A.1 summarizes the results of running NiCad for the different configuration parameters, where C_{so} is the set of snippet clones attributed in Java files and C_{nicad} is the set of snippet clones identified by NiCad. We manually verified all reported clones and recorded each run's true and false positives. Although a $minsize = 5$ would theoretically be able to detect all $2,6565$ clones, we found that NiCad has a very low recall for this parameter. This is rooted in a conservative classification of true positives: in most cases, it was very hard to tell, with so few code lines, whether the snippet was reused from Stack Overflow or the developer actually wrote the snippet.

Table A.1: Effectiveness of different NiCad configurations in detecting attributed code snippets in GitHub source files.

Configuration			**Clone Detection**				
$minsize$	C_{SO}	type	C_{nicad}	TP	FP	Prec	Rec
5	2,656	$type1$	317	317	0	100%	12%
		$type2$	374	368	6	98%	14%
		$type3$	667	646	21	97%	24%
10	770	$type1$	196	196	0	100%	26%
		$type2$	227	227	0	100%	30%
		$type3$	473	464	9	98%	60%
15	452	$type1$	108	108	0	100%	24%
		$type2$	126	126	0	100%	28%
		$type3$	297	295	2	99%	65%

The $minsize = 15$ does not work well either, as the number of potentially detectable clones is very small ($C_{SO} = 452$ or 17% of all possible clones). The reason is that code snippets on Stack Overflow are, on average, 12 lines long [15], and setting a minimum threshold of 15 lines will exclude many snippets. Thus, we selected $minsize = 10$, where the best balance between precision and recall is achieved when detecting type-3 clones. We suspect the reason to be that most code snippets on Stack Overflow are not compilable, and the import statements for libraries are rarely included in the snippets, necessitating code adaptions of type-3. Thus, we selected $< (minsize = 10, type3 >$ as final calibration for NiCad.

We use the selected parameters and ran the clone detector on the Java and C code snippets sample shown in Table 3.1. Figure A.2 shows the cumulative frequently distribution for the NiCad similarity values.The vast majority (80,665 or 95.4%) of the clones are identical (NiCad score of 100). To also include highly similar but not identical clones, we need to determine a cutoff point for the similarity score. We manually inspected some of the reported clones for each language and noticed that some *Java* clone pairs with a 70% similarity score looked structurally similar but semantically different. Since this may skew our results, we resorted to manually verifying a random sample of the reported clones and selected clones belonging to five similarity groups (i.e., 70%, 75%, 80% and 85%). The aim of the manual verification process was to select an optimal *similarity threshold* with which we can be confident that a detected clone is a true clone. The random sample consisted of 30 clone pairs for each similarity group—split equally among question and answer posts—and two researchers manually verified the clone pairs for each group. We found that 21 of the clone pairs in the 70% similarity group were false positives, 13 clone pairs in the 75% similarity group were false positives, 8 clone pairs in the 80% similarity group were false positives and 1 clone

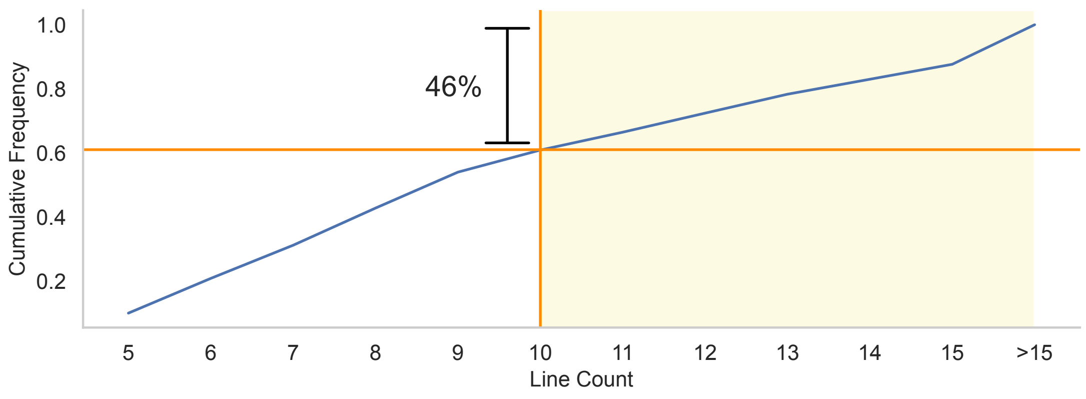


Figure A.1: Distribution of the LoC for code snippets in the attributed dataset. The shaded area indicates the snippet sizes our NiCad configuration covers.

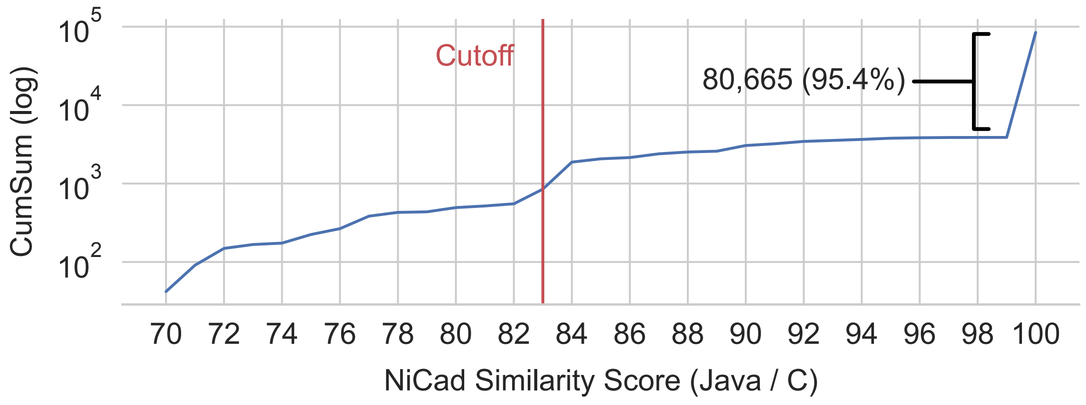
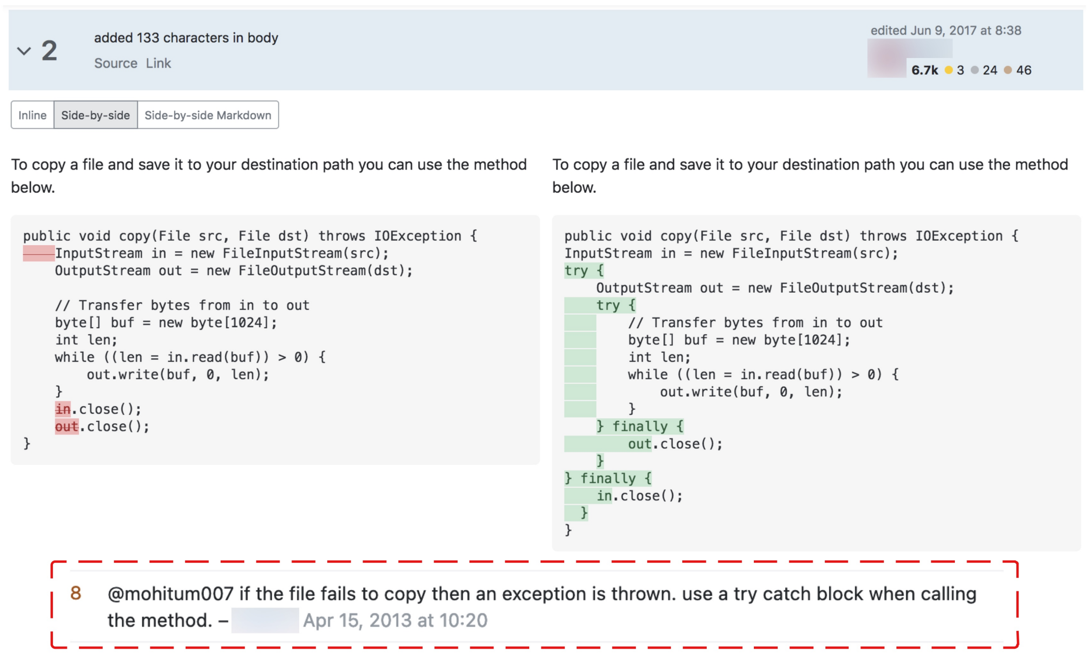


Figure A.2: Cumulative sum of detected clone pairs for NiCad similarities above 70%.

pair in the 85% similarity group was a false positive. We decided to additionally select a random sample of clone pairs in the 83% similarity group, manually verified those, and found 2 false positive clones. As a result, we selected 83% as a similarity threshold for clones of *Java* code snippets. We performed the same experiment for the reported clone pairs for the *C* language but found no false positives for the 70% similarity group. Thus we selected 70% as a similarity threshold for clones of *C* code snippets.

A.2 Additional Examples of Insecure Snippets

In Section 3.1, we provided an example showing how a fix to an *XML eXternal Entities (XXE)* injection vulnerability was missed in the Apache Lucene-Solr and the Apache Chemistry projects. Here, we provide three additional examples of other issues we found.



To copy a file and save it to your destination path you can use the method below.

```
public void copy(File src, File dst) throws IOException {
    InputStream in = new FileInputStream(src);
    OutputStream out = new FileOutputStream(dst);

    // Transfer bytes from in to out
    byte[] buf = new byte[1024];
    int len;
    while ((len = in.read(buf)) > 0) {
        out.write(buf, 0, len);
    }
    in.close();
    out.close();
}
```

To copy a file and save it to your destination path you can use the method below.

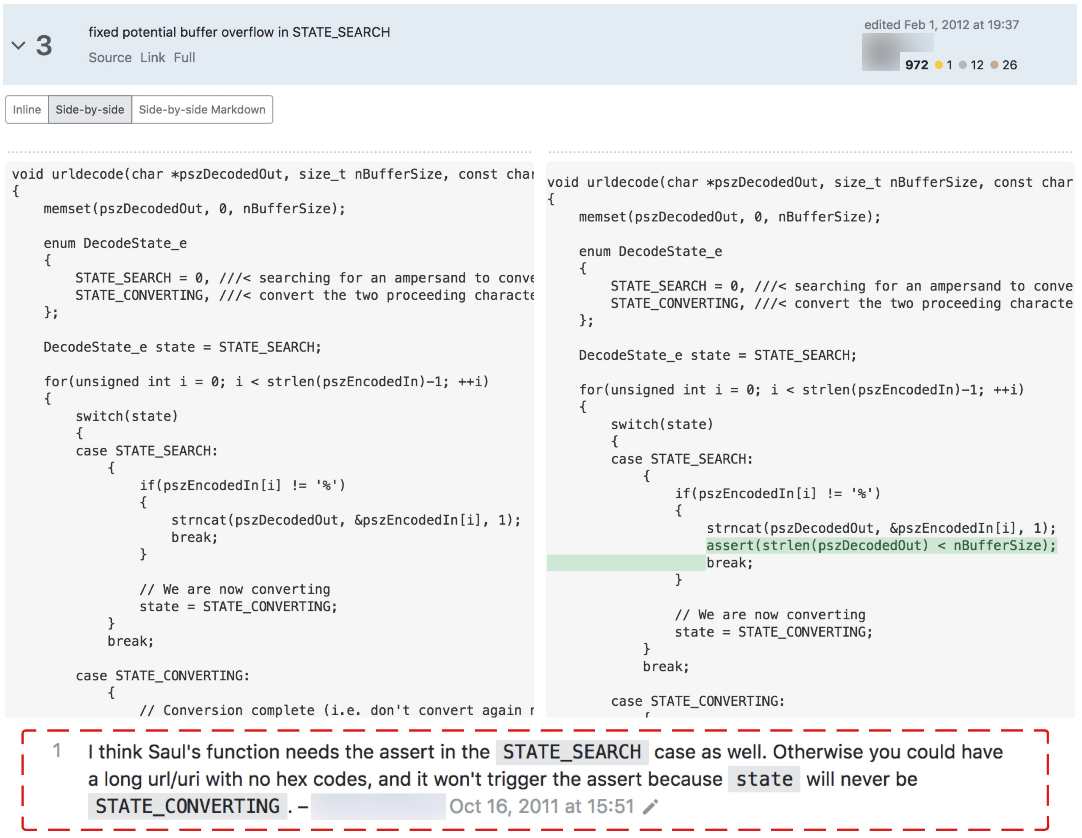
```
public void copy(File src, File dst) throws IOException {
InputStream in = new FileInputStream(src);
try {
    OutputStream out = new FileOutputStream(dst);
    try {
        // Transfer bytes from in to out
        byte[] buf = new byte[1024];
        int len;
        while ((len = in.read(buf)) > 0) {
            out.write(buf, 0, len);
        }
    } finally {
        out.close();
    }
} finally {
    in.close();
  }
}
```

8 @mohitum007 if the file fails to copy then an exception is thrown. use a try catch block when calling the method. – [redacted] Apr 15, 2013 at 10:20

Figure A.3: Answer `9293885` with CWE-404.

A.3 Projects Missing Improvements

Table A.2 shows the list of GitHub projects missing various improvements to code reused from Stack Overflow. Java projects (17/20) were mainly affected by those and were concerned with the release of Java 8 in which functional APIs were added to the standard library. Together, these projects were watched on average 179.2 times (max. 783), received an average of 3,790.9 stars (max. 12,361), and were forked on average 94.6 times (max. 809).



3 fixed potential buffer overflow in STATE_SEARCH

Source Link Full

edited Feb 1, 2012 at 19:37

[redacted] 972 ● 1 ● 12 ● 26

```
void urldecode(char *pszDecodedOut, size_t nBufferSize, const char
{
    memset(pszDecodedOut, 0, nBufferSize);

    enum DecodeState_e
    {
        STATE_SEARCH = 0, ///< searching for an ampersand to conve
        STATE_CONVERTING, ///< convert the two proceeding characte
    };

    DecodeState_e state = STATE_SEARCH;

    for(unsigned int i = 0; i < strlen(pszEncodedIn)-1; ++i)
    {
        switch(state)
        {
        case STATE_SEARCH:
            {
                if(pszEncodedIn[i] != '%')
                {
                    strncat(pszDecodedOut, &pszEncodedIn[i], 1);
                    break;
                }

                // We are now converting
                state = STATE_CONVERTING;
            }
            break;

        case STATE_CONVERTING:
            {
                // Conversion complete (i.e. don't convert again r
```

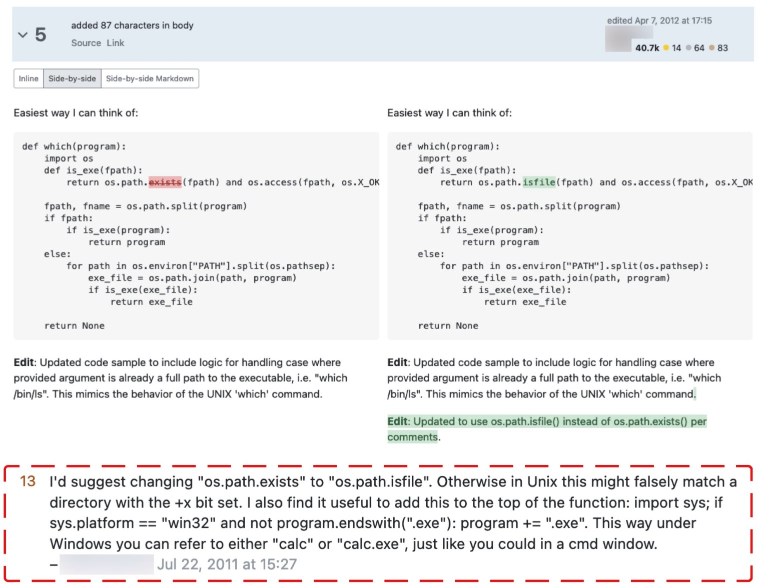
```
void urldecode(char *pszDecodedOut, size_t nBufferSize, const char
{
    memset(pszDecodedOut, 0, nBufferSize);

    enum DecodeState_e
    {
        STATE_SEARCH = 0, ///< searching for an ampersand to conve
        STATE_CONVERTING, ///< convert the two proceeding characte
    };

    DecodeState_e state = STATE_SEARCH;

    for(unsigned int i = 0; i < strlen(pszEncodedIn)-1; ++i)
    {
        switch(state)
        {
        case STATE_SEARCH:
            {
                if(pszEncodedIn[i] != '%')
                {
                    strncat(pszDecodedOut, &pszEncodedIn[i], 1);
                    assert(strlen(pszDecodedOut) < nBufferSize);
                    break;
                }

                // We are now converting
                state = STATE_CONVERTING;
            }
            break;

        case STATE_CONVERTING:
```

1 I think Saul's function needs the assert in the `STATE_SEARCH` case as well. Otherwise you could have a long url/uri with no hex codes, and it won't trigger the assert because `state` will never be `STATE_CONVERTING`. – [redacted] Oct 16, 2011 at 15:51

Figure A.4: Buffer Overflow issue fixed in version 3 of the post.

Project	Language	Watch	Fork	Star	Contributors	Last Commit (year)
OsmAnd (OSM Automated Navigation Directions)	Java	138	875	3,115	809	2022
Azure SDK for Java	Java	280	1,431	3,998	447	2022
F-Droid Client	Java	57	104	706	353	2022
GeoTools (The Open Source Java GIS Toolkit)	Java	110	1,008	1,274	235	2022
J2ObjC (Java to Objective-C Translator and Runtime)	Java	307	920	5,877	82	2022
sshj (SSHv2 library for Java)	Java	118	517	2,094	69	2022
Eclispe Deeplearning4J (DL4J)	Java	783	4,923	12,361	57	2022
MGit (Git client for Android)	Java	45	121	594	56	2022
GassistPi (Google Assistant for Single Board Computers)	Python	77	309	977	16	2022
Apache POI	Java	76	593	1,401	15	2022
PictureSelector	Java	199	2,635	11,450	5	2022
WebRTC Chrome Extensions	JavaScript	80	483	894	5	2022
Open Event Website App Generator	JavaScript	38	870	1,990	89	2021
React Designer	JavaScript	42	241	1,794	8	2021
LitePal for Android	Java	293	1,587	7,835	4	2021
w3af (Web Application Attack and Audit Framework)	Python	193	1,151	3,917	64	2020
WebRTC-Experiment	JavaScript	669	3,850	10,603	11	2020
Matisse	Java	241	2,011	12,258	28	2019
MusicDNA	Java	93	590	2,737	8	2019
libstreaming	Java	267	1,036	3,215	6	2019
Friend Spell	Java	11	74	415	6	2017
Pretty Curved Privacy (PCP)	C	7	5	125	3	2017
InstaMaterial	Java	307	1,498	5,028	3	2016
VOMS Java API	Java	10	7	5	4	2016
Livestreamer	Python	208	617	3,844	75	2016
Breeze	C	9	18	57	1	2013
Average		**179.2**	**1056.7**	**3790.9**	**94.6**	
Max		**783**	**4,923**	**12,361**	**809**	

Table A.2: GitHub projects missing fixes to code improvements on Stack Overflow



5 added 87 characters in body

Source Link

edited Apr 7, 2012 at 17:15

[redacted] 40.7k ● 14 ● 64 ● 83

Inline | Side-by-side | Side-by-side Markdown

Easiest way I can think of:

```
def which(program):
    import os
    def is_exe(fpath):
        return os.path.~~exists~~(fpath) and os.access(fpath, os.X_OK

    fpath, fname = os.path.split(program)
    if fpath:
        if is_exe(program):
            return program
    else:
        for path in os.environ["PATH"].split(os.pathsep):
            exe_file = os.path.join(path, program)
            if is_exe(exe_file):
                return exe_file

    return None
```

Edit: Updated code sample to include logic for handling case where provided argument is already a full path to the executable, i.e. "which /bin/ls". This mimics the behavior of the UNIX 'which' command.

Easiest way I can think of:

```
def which(program):
    import os
    def is_exe(fpath):
        return os.path.isfile(fpath) and os.access(fpath, os.X_OK

    fpath, fname = os.path.split(program)
    if fpath:
        if is_exe(program):
            return program
    else:
        for path in os.environ["PATH"].split(os.pathsep):
            exe_file = os.path.join(path, program)
            if is_exe(exe_file):
                return exe_file

    return None
```

Edit: Updated code sample to include logic for handling case where provided argument is already a full path to the executable, i.e. "which /bin/ls". This mimics the behavior of the UNIX 'which' command.

Edit: Updated to use os.path.isfile() instead of os.path.exists() per comments.

13 I'd suggest changing "os.path.exists" to "os.path.isfile". Otherwise in Unix this might falsely match a directory with the +x bit set. I also find it useful to add this to the top of the function: import sys; if sys.platform == "win32" and not program.endswith(".exe"): program += ".exe". This way under Windows you can refer to either "calc" or "calc.exe", just like you could in a cmd window.
– [redacted] Jul 22, 2011 at 15:27

Figure A.5: The most reused, outdated, buggy (no CWE) snippet with a subsequent fix. The changes in the fix were summarized in the post body and not using a commit message

B Appendix: Effects on Research Results

B.1 Literature Search

B.1.1 OpenAI GPT4o Evaluation

Listing B.1: "Final GPT4o prompt"

```
{
  "messages": [
    {
      "role": "system",
      "content": "You are an expert in analyzing academic research focused on ↪
          the security or correctness (e.g., buggy or faulty) of code snippets ↪
          on Stack Overflow. Do not consider studies that focus on general ↪
          security discussions or questions about security, unless they ↪
          specifically analyze the security of the code snippets themselves, ↪
          or identify and address issues such as bugs or faults in the code ↪
          snippets."
    },
    {
      "role": "user",
      "content": "Does this paper study the security of code snippets on Stack ↪
          Overflow or identify and address issues such as bugs or faults in ↪
          the code snippets? Provide a 'Yes' or 'No' and a brief justification ↪
          in the following JSON format: {'SecurityRelevant': 'True' or ↪
          'False', 'Justification': 'brief justification' }:\n\n{paper_abstract}"
    }
  ]
}
```

To assess GPT4o's performance in screening papers based on their relevance to the security and bugs of code snippets, we tested it with two sets of manually labeled studies: 10 true positive cases fitting these criteria and 10 false positive cases. We carefully selected the negative samples to include studies that almost matched the inclusion criteria, helping us evaluate how well the model distinguishes between relevant and irrelevant studies. We iteratively refined our LLM prompt until we were confident it would correctly identify studies that meet our criteria and exclude those that do not. The results of this evaluation are provided in Table B.1 and show that GPT4o performed well in distinguishing the papers. The optimized LLM prompt is listed in Listing B.1.

Table B.1: Papers used to test the GPT4o performance

Paper Title	**Truth Label**	**GPT4o Label**
Dicos: Discovering Insecure Code Snippets from Stack Overflow Posts by Leveraging User Discussions	True	True
Toxic Code Snippets on Stack Overflow	True	True
Does Collaborative Editing Help Mitigate Security Vulnerabilities in Crowd-Shared IoT Code Examples?	True	True
Mining Rule Violations in JavaScript Code Snippets	True	True
Snakes in Paradise?: Insecure Python-Related Coding Practices in Stack Overflow	True	True
How Reliable is the Crowdsourced Knowledge of Security Implementation?	True	True
Secure Coding Practices in Java: Challenges and Vulnerabilities	True	True
You Get Where You're Looking for: The Impact of Information Sources on Code Security	True	True
Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security	True	True
Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography	True	True
Understanding Privacy-Related Questions on Stack Overflow	False	False
Exploring Technical Debt in Security Questions on Stack Overflow	False	False
Crowd-GPS-Sec: Leveraging Crowdsourcing to Detect and Localize GPS Spoofing Attacks	False	False
An Investigation of Security Conversations in Stack Overflow: Perceptions of Security and Community Involvement	False	False
An Anatomy of Security Conversations in Stack Overflow	False	False
Is Reputation on Stack Overflow Always a Good Indicator of Users' Expertise? No	False	False
Mobile App Security Trends and Topics: An Examination of Questions from Stack Overflow	False	False
Challenges in Docker Development: A Large-Scale Study Using Stack Overflow	False	False
What Security Questions Do Developers Ask? A Large-Scale Study of Stack Overflow Posts	False	False
Hurdles for Developers in Cryptography	False	False

B.2 Evolution of Stack Overflow

Table B.2 provides an overview of the edits per code snippet on Stack Overflow, broken down by the snippets' programming languages. Table B.3 lists the number of commit messages by programming language and by their type (i.e., no commit message, not security-relevant, security-relevant). Figure B.1 and B.2 depict the normalized confusion matrices for Table B.3. A $\chi^2_{\text{All}}(10, N = 19,311,100) = 16288$, $p_{\text{All}} < 0.001$, Cramér's $V_{\text{All}} = 0.021$ for all commit messages and $\chi^2_{\text{NonEmpty}}(5, N = 6,134,834) = 6624$, $p_{\text{NonEmpty}} < 0.001$, Cramér's $V_{\text{NonEmpty}} = 0.023$ shows a significant relationship between programming languages and type of commit message. Table B.4 shows the mean monthly PSC and a fitted linear regression per programming language. Figures B.3 to B.7 illustrate the corresponding distributions of commit messages and PSC per language. Table B.5 shows the Kwiatkowski–Phillips–Schmidt–Shin (KPSS) and Augmented Dickey-Fuller (ADF) test statistics for the PSC with and without empty commit messages. Table B.6 shows the mean monthly percentage of newly added security-relevant comments and the fitted linear regression on this percentage, broken down by the programming language of the code snippets in the commented post. Table B.7 presents the KPSS and ADF test statistics for the time series of this monthly percentage. Figure B.8 to B.13 depicts the time series data for comments.

Table B.2: Edits per code snippet in different languages.

Language	Count	Mean	Min	75%	99%	Max
C	82,456	1.474±0.007	1	2	5	36
C++	90,264	1.480±0.007	1	2	5	34
Java	359,257	1.307±0.002	1	1	4	27
JavaScript	700,592	1.311±0.002	1	1	4	53
Python	455,606	1.410±0.003	1	2	5	58

Table B.3: Nr. of commit messages by language and type.

Language	Empty Message	Not Security-Relevant	Security-Relevant
All	12,492,209	4,875,276	688,157
C	222,398	85,173	19,133
C++	239,386	81,796	15,663
Java	931,999	419,725	59,261
Python	1,004,514	352,041	62,442
JavaScript	1,463,354	558,779	78,482

B.3 Case Study 1: C/C++ Code Weakneses

Table B.8 presents a detailed, side-by-side comparison of the results from the original methodology by Zhang et al. [118] and our re-implementation, which utilized Cppcheck 1.86 for both language and security weakness detection (see Figure 4.6 for details). We identified 15,724 C/C++ answers containing security weaknesses. Of these, 11,142 answers were also flagged as containing weaknesses in the original study's 11,235 answers. This indicates that our approach missed only 93 answers with weaknesses

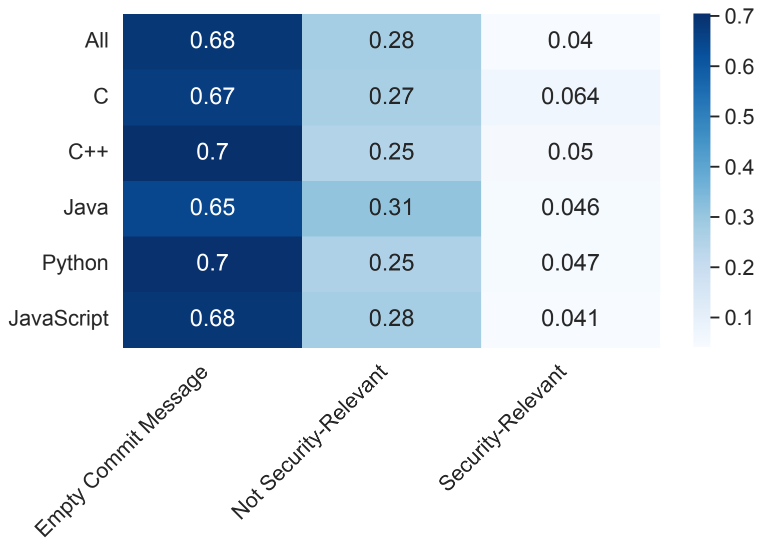


Figure B.1: Normalized confusion matrix for Table B.3

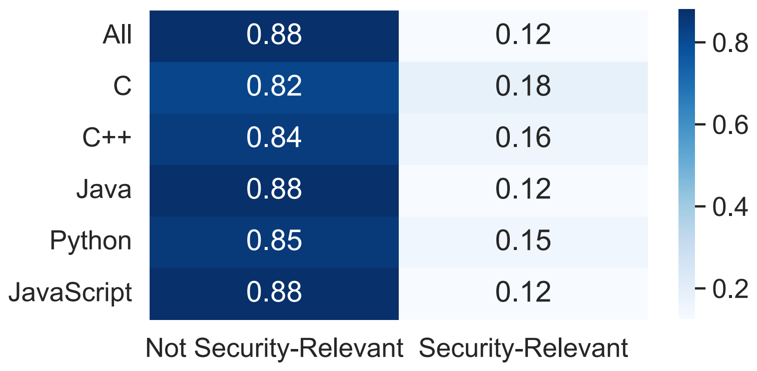


Figure B.2: Normalized confusion matrix excluding empty commit messages.

from the original study, demonstrating that our approach captures the same snippets as the original methodology. We surmise that the increase in detected snippets and snippet versions in our methodology is rooted in Guesslang's performance in the original methodology, i.e., many C/C++ snippets were misclassified as another language and hence not analyzed with Cppcheck by the authors. However, since the exact Cppcheck version used by the authors is unknown and we empirically estimated it to be version 1.86, we cannot exclude that our guessed version differs and performs differently from the authors' version.

Table B.9 compares the proportion of $Code_w$ versus the number of code revisions that were detected with the original methodology by Zhang et al. [118] for SOTorrent18 and our re-implementation based on SOTorrent22.

Table B.10 lists the vulnerable answers, snippets, and versions detected with different versions of Cppcheck on different versions of SOTorrent. We examined with a paired t-test whether the effect of changing the SOTorrent dataset version was stable across

Table B.4: PSC categorized by language. Monthly average PSC (with $CI = 95\%$) plus R^2 and significance for a fitted linear regression.

	All Commit Messages		**Non-Empty Commit Messages**	
Language	Mean PSC	R^2	Mean PSC	R^2
All	4.136±0.107	0.128***	12.863±0.167	0.002
C	6.331±0.242	0.031*	18.699±0.457	0.069***
C++	5.069±0.169	0.207***	16.570±0.325	0.000
Java	4.729±0.163	0.027*	13.372±0.309	0.008
Python	5.118±0.152	0.382***	15.852±0.249	0.006
JavaScript	4.144±0.147	0.016	13.10 ±0.24	0.024*

* $p < 0.05$, ** $p < 0.01$, ***$p < 0.001$

Table B.5: Kwiatkowski–Phillips–Schmidt–Shin (KPSS) and Augmented Dickey-Fuller (ADF) test statistics for PSC.

	All Commit Messages			**Non-Empty Commit Messages**		
	KPSS	ADF	Stationary	KPSS	ADF	Stationary
All	0.417	−4.848***	Stationary	0.386	−2.004	Trend Stationary
C	0.192	−5.918***	Stationary	0.780*	−4.508***	Difference Stationary
C++	0.876*	−4.409***	Difference Stationary	0.074	−11.060***	Stationary
Java	0.211	−6.152***	Stationary	0.356	−3.525**	Stationary
Python	1.139*	−4.130***	Difference Stationary	0.306	−5.173***	Stationary
JavaScript	0.171	−5.367***	Stationary	0.488*	−2.122	Non-Stationary

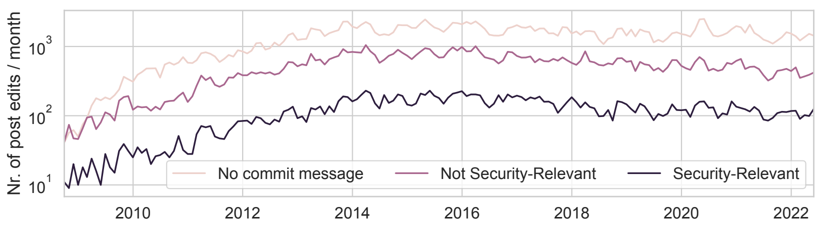
* $p < 0.05$, ** $p < 0.01$, ***$p < 0.001$

different versions of Cppcheck. Shapiro-Wilk tests confirmed the normality of the distribution of differences between paired observations for the SOTorrent versions for Cppcheck v1.86 ($0.832, p = 0.193$) and v2.13 ($0.852, p = 0.247$), respectively. The t-test indicates that the effect of upgrading the SOTorrent dataset is not consistent across Cppcheck versions ($t = -8.90, p < 0.05$). The change from SOTorrent18 to SOTorrent22 significantly impacted the results when using Cppcheck v2.13 compared to v1.86, implying that changes in the dataset can have a different impact depending on Cppcheck version. Although using Cppcheck v1.86 on SOTorrent22 would better isolate the effect of code evolution, we still decided to report the results of Cppcheck v2.13 in our replication study. We reason that if the authors had conducted their experiment later, they would have used the newer tool version.

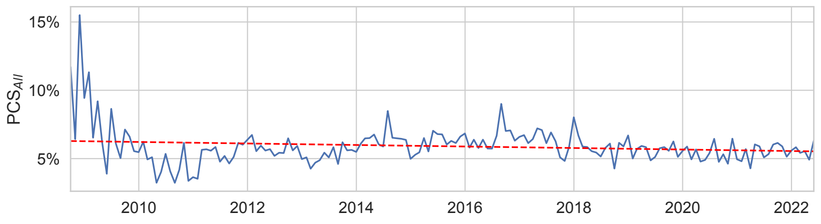
B.3.1 Details about Replicating RQ3

Our data set based on SOTorrent22 and Cppcheck v2.13 comprises 495,330 C/C++ code snippets from 75,779 users. This number is smaller than the original data (85k users), which we surmise is due to a larger fraction of non-C/C++ snippets falsely identified as such by Guesslang in the original methodology. See Figure 4.6 for a comparison of the original methodology with the approach used in the replication study.

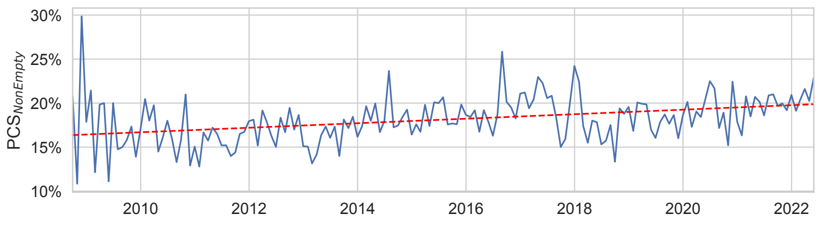
Table B.11 compares the results related to RQ3 and we elaborate on these in the



(a) Nr. of monthly post edits categorized by their security relevance.



(b) Including empty commit messages.



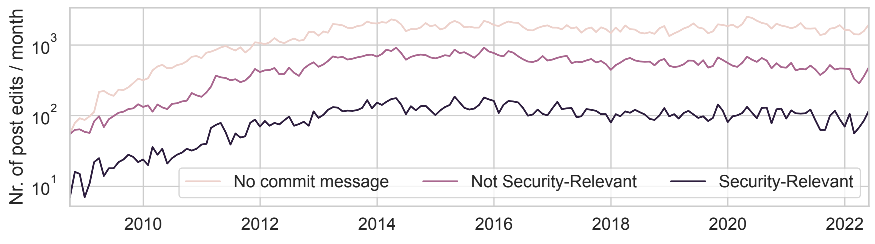
(c) Excluding empty commit messages.

Figure B.3: C language: PSC in monthly intervals. Dashed line is the fitted linear regression.

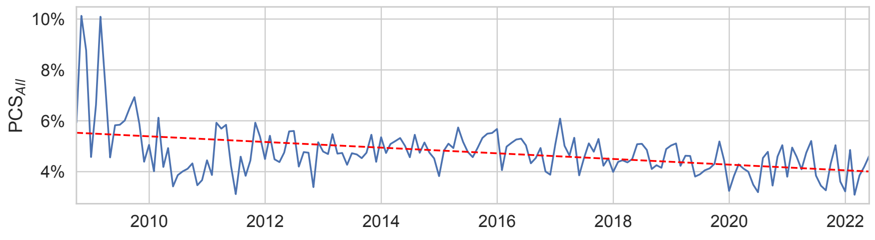
following.

Zhang et al. [118] found that *"the majority of the C/C++ $Version_w$ were contributed by a small number of users"* and that *"72.4 percent (i.e., 10,652) of $Version_w$ were posted by 36 percent (i.e., 2,292) of users."* In our replication, we found a shift where an even smaller number of users contributed $Version_w$, see Figure 4.7. We found that 72.4 percent (i.e., 35,034) of $Version_w$ were posted by 25 percent (i.e., 3,205) of users. In our data set, 36 percent (i.e., 4,625) of users contributed 79.5 percent (i.e., 38,481) of the $Version_w$. Moreover, Zhang et al. reported that *"64.0 percent (i.e., 4,070) of the users who contribute $Version_w$ have contributed only one $Version_w$."* We found that 86.2 percent (i.e., 11,077) of users contributed only one $Version_w$. Further, they reported that *"among all the 85,165 users who posted C/C++ code snippets, only 7.5 percent (i.e., 6,361) of them posted code snippets that have weaknesses."* In contrast, in our replication study, 17.0 percent (i.e., 12,845) of 75,779 users contributed code snippets with weaknesses.

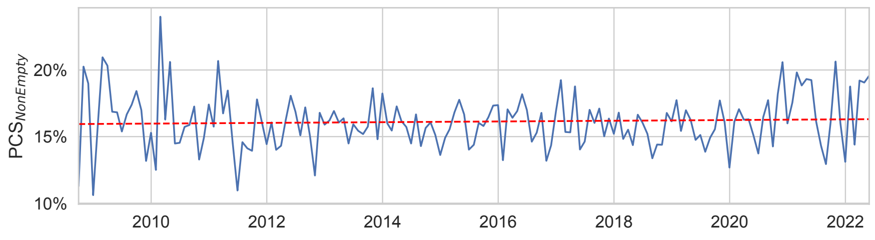
Next, Zhang et al. explored the connection between user activity and code weaknesses. They found that "more active users are less likely to introduce $Code_w$." We depict the same connection in Figure 4.8, adopting the authors' plot style. The authors concluded that *"the weakness density of a user's code drops when the number of contributed code*



(a) Nr. of monthly post edits categorized by their security relevance.



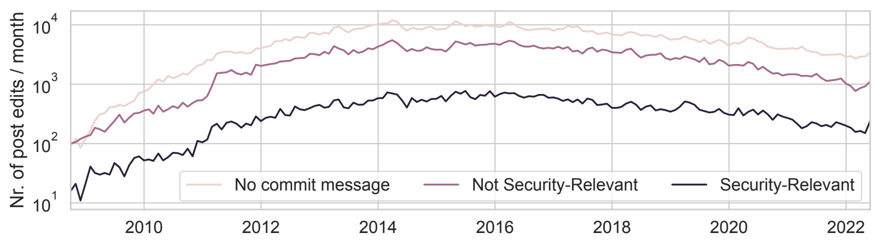
(b) Including empty commit messages.



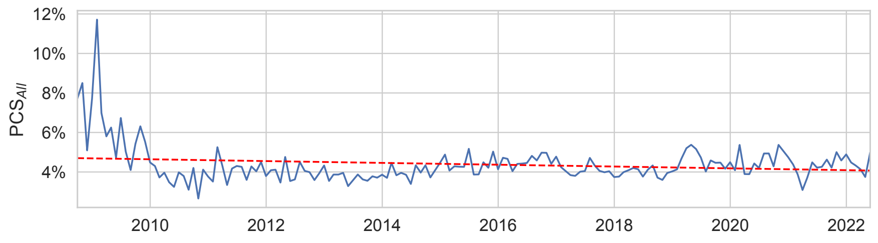
(c) Excluding empty commit messages.

Figure B.4: C++ language: PSC in monthly intervals. Dashed line is the fitted linear regression.

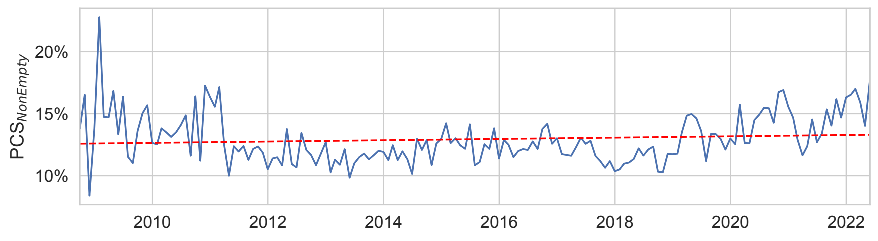
revisions by the user increases." We explored the relation between the number of code revisions and the density of contributed $Version_w$ by users with statistical testing. A Pearson correlation analysis revealed a weak correlation, $r(4) = -0.190, p < 0.001$, suggesting that as the number of revisions increases, the density of code with weaknesses decreases slightly. A linear regression analysis was conducted to examine further the relationship between the number of revisions (independent variable) and the weakness density (dependent variable). The results indicated that the model was statistically significant $F(1, 12843) = 478.4$, $p < 0.001$, suggesting a significant inverse association of code revision count with weakness density. The model explained, however, only a small proportion of the variance in weakness density, $R^2 = 0.036$. These findings suggest that the weakness density is expected to decrease by 0.0005 for each additional revision. However, the low R^2 value indicates that the number of revisions explains only 3.6% of the variability in weakness density, suggesting that other factors may also play a significant role. Overall, the correlation and regression analyses support the conclusion that there is a statistically significant but weak inverse relationship between the number of revisions and weakness density. Further research is needed to explore additional variables and potential non-linear relationships that might better explain the variability



(a) Nr. of monthly post edits categorized by their security relevance.



(b) Including empty commit messages.



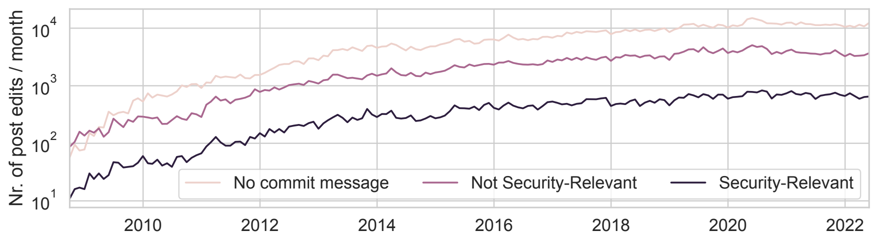
(c) Excluding empty commit messages.

Figure B.5: Java language: PSC in monthly intervals. Dashed line is the fitted linear regression.

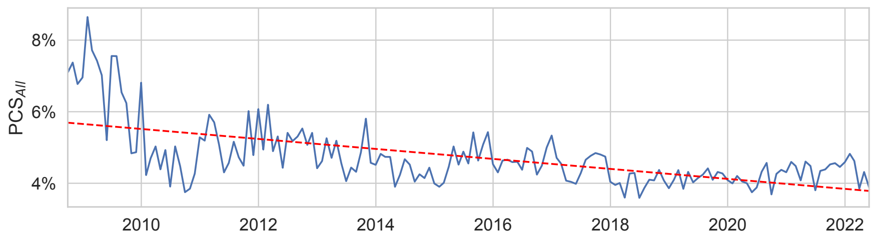
in weakness density.

Zhang et al. [118] also compared the reputation of the contributor for a code version with different numbers of CWE instances in the code. Their results showed *"that users with higher reputation tend to introduce fewer CWE instances in their contributed code versions."* We also explored the relationship between the users' reputation and the number of CWE instances. Figure B.14 depicts the relation between user's reputation and the number of CWE instances. Both linear regression and the Pearson correlation analysis suggest that while a statistically significant relationship ($p < 0.001$) exists between reputation and the number of CWE instances by a user, the effect size is very small. The linear regression had a practically negligible effect on the number of CWE, with an $R^2 = 0.001$. The Pearson correlation coefficient is -0.0227, indicating a very weak inverse relationship. This implies that reputation alone is not a strong predictor of the number of CWEs. Further research with additional variables or different analytical approaches might be needed to understand better the factors influencing the number of CWEs.

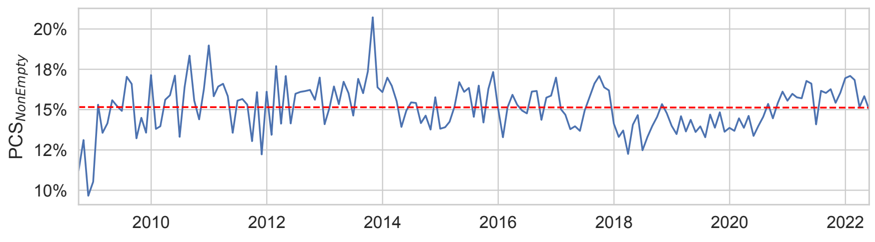
Next, Zhang et al. found that *"78.0 percent of users contribute code with only one CWE type."* Figure B.15 depicts the number of users vs. number of distinct CWE types



(a) Nr. of monthly post edits categorized by their security relevance.



(b) Including empty commit messages.



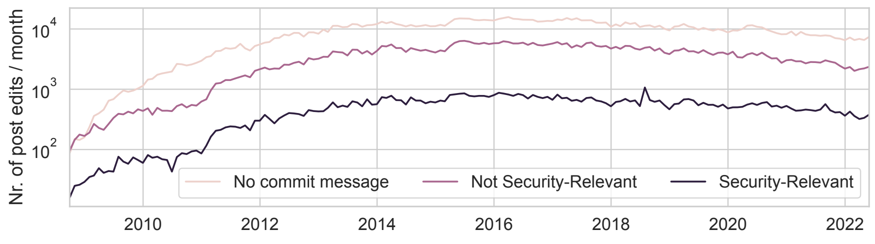
(c) Excluding empty commit messages.

Figure B.6: Python language: PSC in monthly intervals. Dashed line is the fitted linear regression.

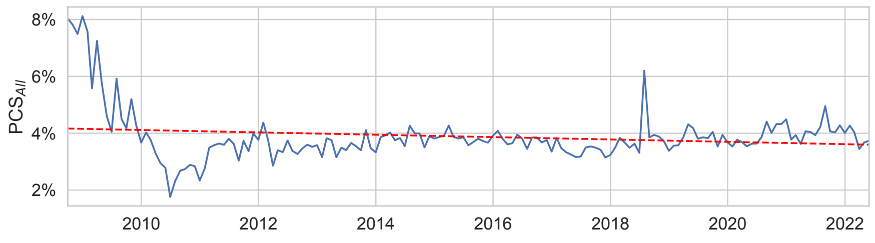
by the user in our replication study. We found that 74.6 percent (i.e., 9,582) of users contribute only one CWE type.

The next result reported by Zhang et al. is that *"42.2 percent (i.e., 2,686) of the users contribute only one CWE instance in all their $Version_w$."* and that *"81.8 percent (i.e., 5,206) of the users contribute less than five CWE instances in all their $Version_w$."* In Figure B.16, we compare the authors' results with ours. We find that 50.9 percent (i.e., 6,533) of the users contribute only one CWE instance in all their $Version_w$ and that 96.7 percent of the users contribute less than five CWE instances in all their $Version_w$ We compare the two distributions in Figure B.16 with a χ^2 test and find that there is a statistically significant difference with moderate effect ($\chi^2 = 146.623, p < 0.001$, Cramér's $V = 0.086$).

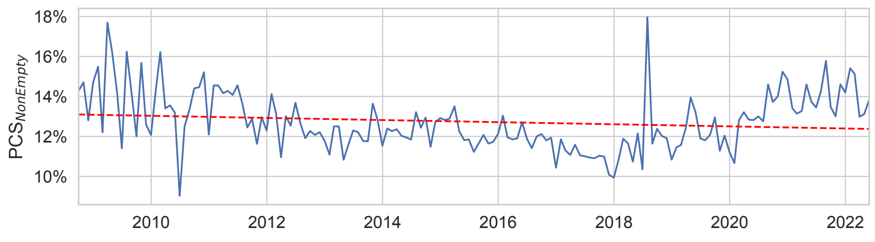
Zhang et al. further found that "users tend to commit the same types of CWE instances repeatedly" by studying the distribution of the number of contributed CWE instances by different users. We depict this distribution for our replication study in Figure B.17. We also find that certain CWE types have a higher frequency than others—however, the types shifted between the original study and the replication study. Zhang et al. found that users highly frequently contributed CWE-401/775/908, while



(a) Nr. of monthly post edits categorized by their security relevance.



(b) Including empty commit messages.



(c) Excluding empty commit messages.

Figure B.7: JavaScript language: PSC in monthly intervals. Dashed line is the fitted linear regression.

we found that CWE-457/476/758/788 are significantly more frequent, while CWE 908 disappeared.

Lastly, Zhang et al. calculated the normalized entropy of the CWE types each user posted in $Code_w$ to better understand how users contribute different CWE types. Here, an entropy value of 0 indicates that the user only contributed a single type of CWE in their code snippets. Figure B.18 compares the distribution of the normalized entropy for the original results and our replication study. The authors observed that *"37.7 percent of users are likely to introduce a single type of CWE instances in their posted code versions."* In contrast, we found that most users (82.5 percent) had a normalized entropy $> .5$, and only 15.6 percent had an entropy value of 0. A χ^2 test for homogeneity between the entropy in the original paper and our replication shows that there is a statistically significant difference between the distributions ($\chi^2(8, N = N = 3,059) = 243.501, p < 0.001$, Cramér's $V = 0.278$).

B.3.2 A remark on the methodology for RQ3

While replicating the results for Zhang et al.'s RQ3, we noticed a potential mistake in how CWE instances are counted for users. Concretely, the authors stated in their paper:

Table B.6: Monthly average percentage of created security-relevant comments (with $CI = 95\%$) as well as R^2 and significance for a fitted linear time series regression .

	Mean PSC	R^2
All	7.781 ± 0.143	0.935***
C	12.134 ± 0.286	0.874***
C++	11.125 ± 0.237	0.799***
Java	8.959 ± 0.212	0.844***
Python	7.091 ± 0.161	0.846***
JavaScript	6.971 ± 0.167	0.917***

* $p < 0.05$, ** $p < 0.01$, ***$p < 0.001$

Table B.7: Kwiatkowski–Phillips–Schmidt–Shin (KPSS) and Augmented Dickey–Fuller (ADF) test statistics for the percentage of security-relevant comments.

	KPSS	ADF	Stationary
All	1.851*	−3.778**	Difference Stationary
C	1.904*	−1.341	Non-Stationary
C++	1.831*	−1.467	Non-Stationary
Java	1.845*	−2.177	Non-Stationary
Python	1.858*	−2.781	Non-Stationary
JavaScript	1.882*	−0.323	Non-Stationary

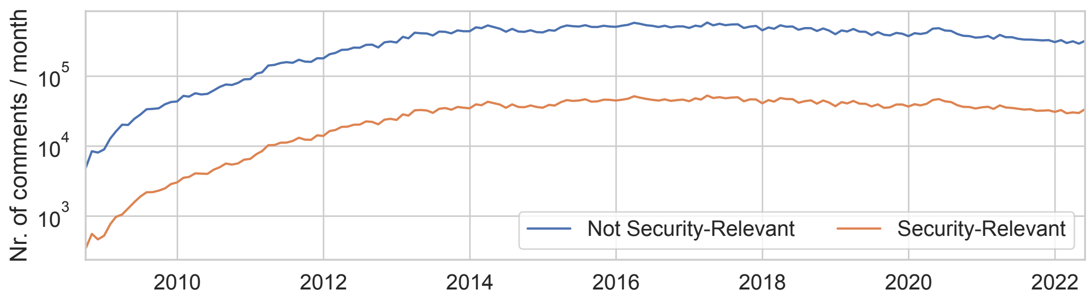
* $p < 0.05$, ** $p < 0.01$, ***$p < 0.001$

one user has contributed CWE-775, i.e., missing release of file descriptor or handle after effective lifetime, for 79 times.[1]

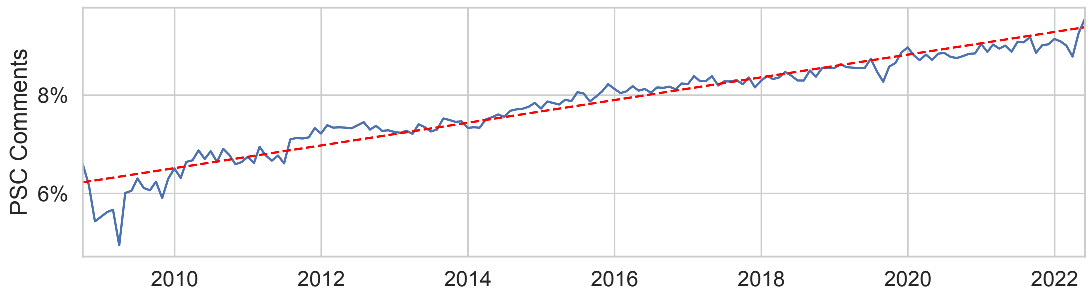
Our data set based on SOTorrent22 found that this user has 104 code revisions with this particular CWE. However, only in 59 (57%) of these revisions did this user *add* the CWE-755 to the code; in the other cases, the author posted a revision to a snippet that already contained CWE-755 and the revision did not fix this CWE. This particular way of counting "inherited CWE instances" bears the risk of over-reporting CWE instances for users, thus inflating the observed dangerous behavior of users by "punishing" them for not fixing CWEs during revisions.

We briefly explored the impact of this approach by re-running our replication but only counting *added* CWE instances. While we did not find a significant drop in the number of users who contributed CWEs—indicating that this "punishment" affected users who contributed CWEs anyway—we found that the distribution of CWEs and their entropy changed. Figure B.19 depicts the number of users contributing different numbers of CWE instances. When only considering *added* CWE instances, the number of users with only one CWE instance is higher, while the number of users with multiple CWE instances drops. A χ^2 test shows this difference between the two flavors of replication studies to be significant ($\chi^2(5, N = 25,396) = 169.420, p < 0.001$, Cramér's $V = 0.080$). Figure B.20 shows the CWE instance distribution based on

[1] `https://stackoverïňĆow.com/users/3422102/`



(a) Nr. of monthly comments categorized by their security relevance.



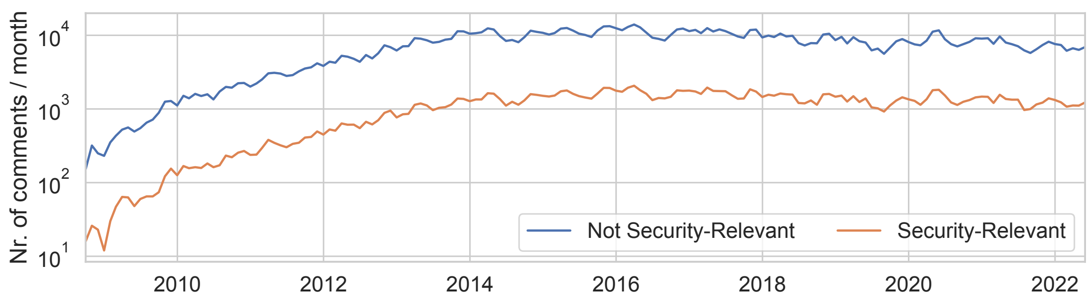
(b) Percentage of new security-relevant comments per monthly interval. Dashed line is the fitted linear regression.

Figure B.8: Security-relevant comments on Stack Overflow

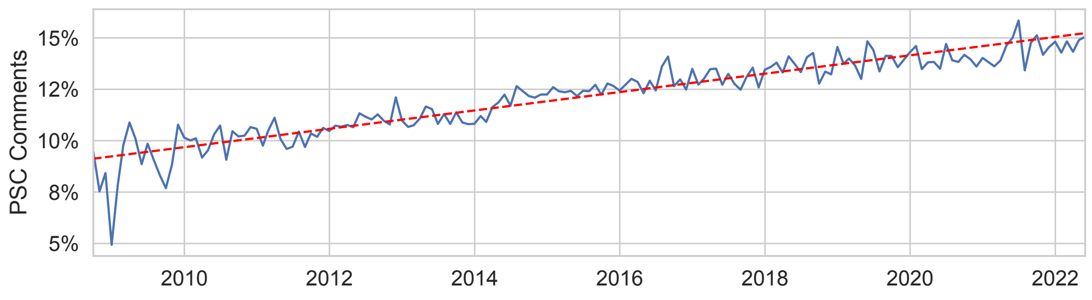
SOTorrent18 (Original)				**SOTorrent18 (Evaluation)**			
	Answer #	*Code Snippet #*	*Code Version #*		*Answer #*	*Code Snippet #*	*Code Version #*
SOTorrent	867,734	1,561,550	1,833,449	SOTorrent	867,734	1,561,550	1,833,449
LOC >= 5	527,932	724,784	919,947	LOC >= 5	527,932	724,784	919,947
Guesslang	490,778	646,716	826,520	Cppcheck $v1.86$	141,215	170,974	206,582
$Code_w$	**11,235**	**11,748**	**14,934**	$Code_w$	**15,724**	**16,533**	**20,664**

Table B.8: Comparison of results by Zhang et al. (118) and our evaluation using Cppcheck v1.86 for both language and security weakness detection. (cf. Table 1 in (118))

this alternative methodology. Compared to Figure B.17, several CWEs have a lower instance count. Further, Figure B.21 compares the entropy per user between the original study, our replication with the same methodology as the original, and our replication considering only added CWEs. When considering only *added* CWE instances, the entropy per user is higher, indicating that users add different CWE types with similar probabilities in their code revisions. The difference in entropy between our replication studies has been confirmed as statistically significant with a χ^2 test for homogeneity ($\chi^2(8, N = 3,329) = 31.031, p < 0.001$, Cramér's $V = 0.083$).



(a) Nr. of monthly comments categorized by their security relevance.

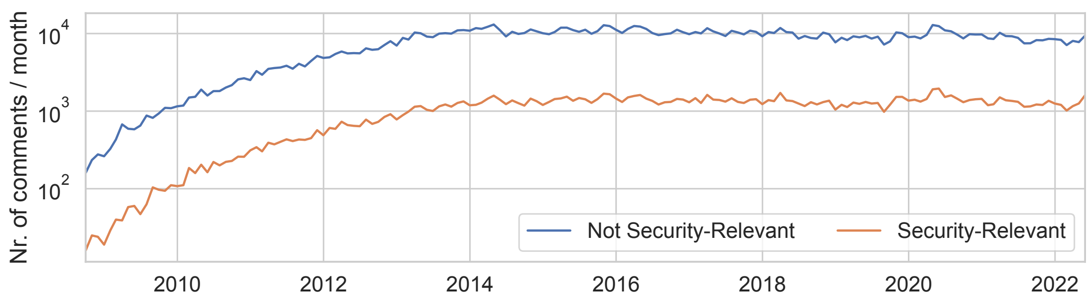


(b) Percentage of new security-relevant comments per monthly interval. Dashed line is the fitted linear regression.

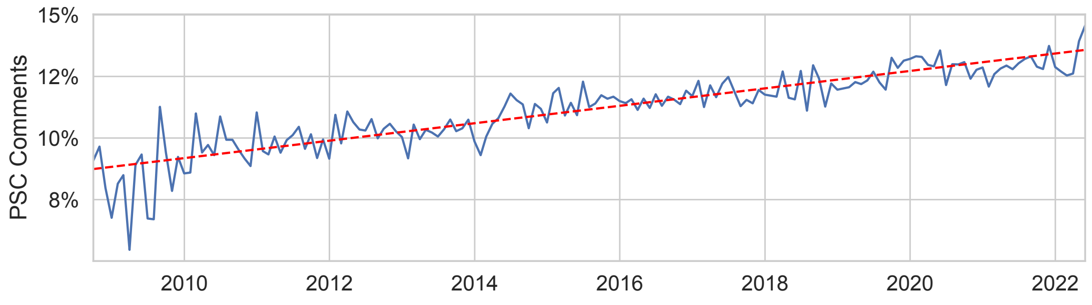
Figure B.9: Security-relevant comments on Stack Overflow for posts with snippets in the **C language**.

B.4 Case Study 2: DICOS: Discovering Insecure code snippets

We replicated the accuracy measurements for C/C++ and Android posts as presented in Table 3 and Table 4 of the original paper [49]. Our comparative findings for C/C++ posts are shown side-by-side in Table B.12 and for Android in Table B.13. For C/C++ posts, we observed an accuracy of **11%** (authors reported 93%), a recall of 92% (vs. 94%), and a precision of **27%** (vs. 90%). Similarly, for Android posts, we found a significant drop in precision (12% vs. 86%) and accuracy (41% vs. 86%) while recall remained high (78% vs. 89%).



(a) Nr. of monthly comments categorized by their security relevance.



(b) Percentage of new security-relevant comments per monthly interval. Dashed line is the fitted linear regression.

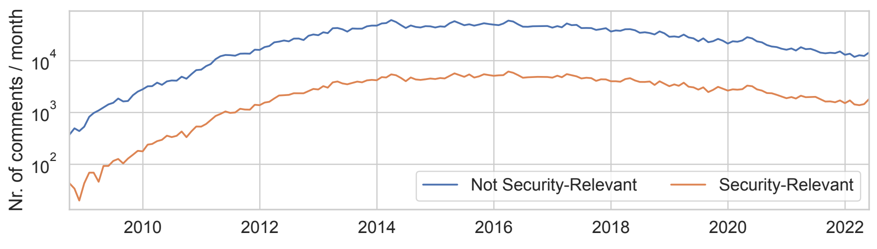
Figure B.10: Security-relevant comments on Stack Overflow for posts with snippets in the **C++ language**.

B.5 Case Study 3&4: Stack Overflow Considered Harmful and Helpful

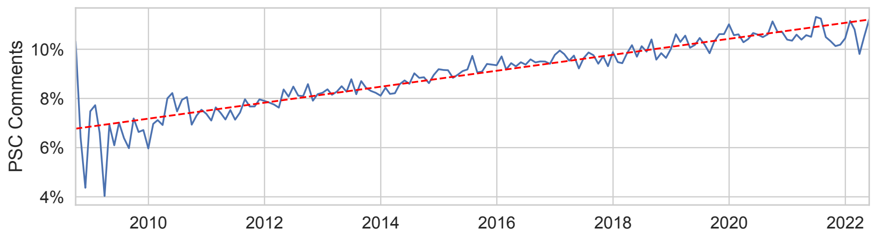
Table B.14 lists all tags from the top-100 (see Figure B.22) that are not crypto-related and were filtered for computing the frequency of co-location with the tags *android* and *java* (see Figure B.23). Table B.15 shows the top-20 crypto-related tags co-located with *android* or *java* among all 23,876,743 questions on Stack Overflow. Table B.16 shows the top 10 co-located tags. This shows that crypto-related questions are a narrow topic among developers.

B.6 Programming Language Detection

During our replication of the work by Zhang et al. [118], we recognized the issue of detecting the correct programming language of code snippets. Considering that various works focus on snippets in a specific language (see Table 4.2) and rely on *Tags* or machine learning, e.g., Guesslang [42], we were interested in their efficacy. We randomly sampled 385 code snippets from the 20,158,096 snippets on Stack Overflow. This forms a representative sample size with a 95% confidence interval and 5% margin of error. We used Guesslang v2.0.1 and GPT4o (via the batch API of OpenAI) to determine the programming language of the snippet. GPT4o was prompted to determine the most



(a) Nr. of monthly comments categorized by their security relevance.



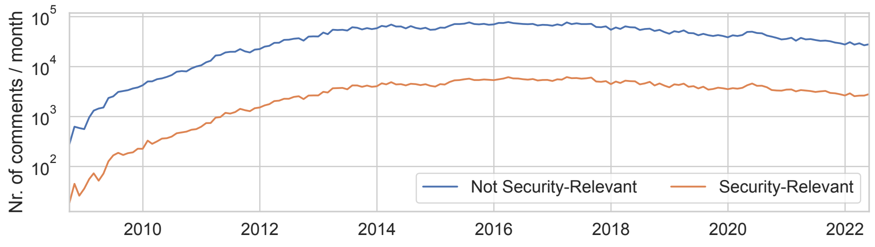
(b) Percentage of new security-relevant comments per monthly interval. Dashed line is the fitted linear regression.

Figure B.11: Security-relevant comments on Stack Overflow for posts with snippets in the **Java language**.

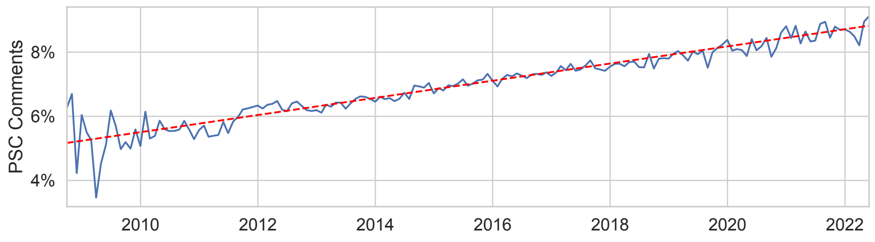
	Authors Results based on SOTorrent18				Results based on SOTorrent22			
#revisions	*Snippets*	*Unchanged*	*Improved*	*Deteriorated*	*Snippets*	*Unchanged*	*Improved*	*Deteriorated*
0	8,103	NA	NA	NA	24,388	NA	NA	NA
≥ 1	3,645	1,886 (51.7%)	1218 (33.4%)	541 (14.8%)	5,866	5,511 (93.9%)	221 (3.8%)	134 (2.3%)
1	2,369	1,340 (56.6%)	**714 (30.1%)**	315 (13.3%)	4,391	4,179 (95.2%)	**136 (3.1%)**	76 (1.7%)
2	774	349 (45.1%)	**294 (38.0%)**	131 (16.9%)	1,058	969 (91.6%)	**54 (5.1%)**	35 (3.3%)
≥ 3	502	197 (39.2%)	**210 (41.8%)**	95 (18.9%)	417	363 (87.1%)	**31 (7.4%)**	23 (5.5%)

Table B.9: The proportion of $Code_w$ versus the number of code revisions by Zhang et al. (118) and our replication study using SOTorrent22 and Cppcheck v2.13. (cf. Table 2 in (118))

likely programming language in a given code snippet. Two researchers manually verified the detected languages and marked the true positives (correct language detected) and negatives (wrong language detected). The researchers also checked if the post's tags contained the correct language. We found that GPT4o achieved an accuracy of 0.93 and Guesslang of 0.55. Only 247 (64.16%) posts listed the programming language in their tags. We observed that the tags often refer to concepts or frameworks, e.g., code snippets in Java were tagged with *Android* but not *Java*. However, since Android encompasses more than Java code, one cannot reliably deduce Java snippets from Android tags. Further, tags are applied exclusively to question posts, each of which can include one or more answers. A single post may contain multiple snippets written in various programming languages.. The correct language was only in three (0.78%) cases in the tags when neither of the two tools could correctly detect it. In 117 (30.39%) cases, GPT4o detected the correct language while the tags did not contain it. However, in 21 (5.45%) cases, neither the two tools nor the tags detected the correct language,



(a) Nr. of monthly comments categorized by their security relevance.



(b) Percentage of new security-relevant comments per monthly interval. Dashed line is the fitted linear regression.

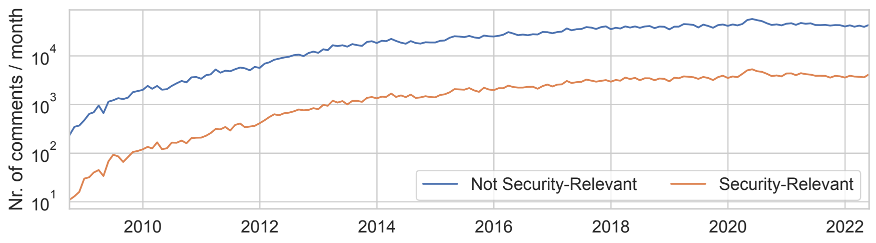
Figure B.12: Security-relevant comments on Stack Overflow for posts with snippets in the **JavaScript language**.

	SOTorrent18			**SOTorrent22**		
	Answer #	*Code Snippet #*	*Code Version #*	*Answer #*	*Code Snippet #*	*Code Version #*
Cppcheck *v*1.86	15,724	16,533	20,664	19,485	20,450	25,832
Cppcheck *v*2.13	23,253	24,699	30,923	28,521	30,254	38,248

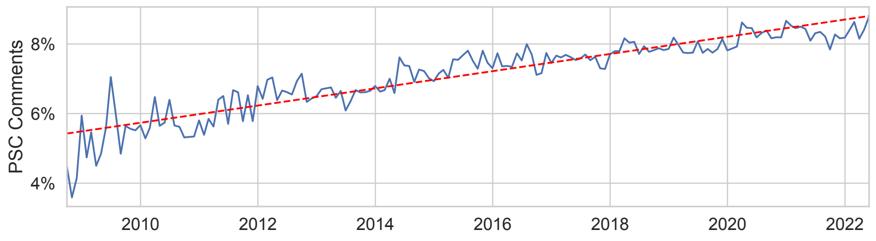
Table B.10: The results of Cppcheck v1.86 and Cppcheck v2.13 on different versions of the SOTorrent dataset.

indicating that even with a combination of these three approaches, there is a residual risk of missing the correct language. We only found two cases in which Guesslang (and Tags) were correct while GPT4o was wrong, underlining GTP4o's better reliability. Figure B.24 illustrates the intersection between the three tools.

Since the works studied in this paper focus on snippets in Java, JavaScript, Python, and C/C++, we further considered how well GuessLang and GPT4o detected these selected languages. To this end, we computed their precision and recall on our sample, where a positive case is a snippet written in either of these languages (see Table B.17). Our random sample contained 156 positive cases and 229 negative cases. GPT4o achieved the best performance, with a precision of 0.93, recall of 0.99, and F1 score of 0.96. Unfortunately, GPT4o is also the least scaling and economically reasonable of these three approaches, creating a call for action to improve the community's Guesslang tool (e.g., with new fine-tuned LLM).



(a) Nr. of monthly comments categorized by their security relevance.



(b) Percentage of new security-relevant comments per monthly interval. Dashed line is the fitted linear regression.

Figure B.13: Security-relevant comments on Stack Overflow for posts with snippets in the **Python language**.

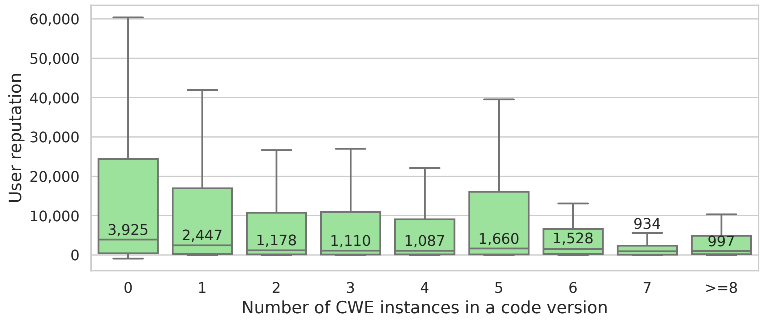


Figure B.14: The distribution of user reputation points for users who contribute code versions both without and with different numbers of CWE instances. (cf. Figure 11 in (118))

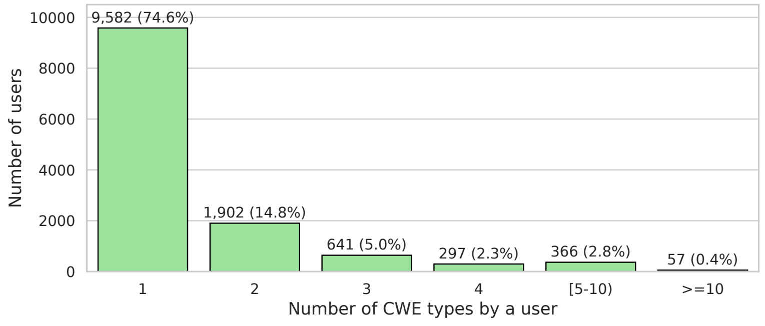


Figure B.15: Number of distinct CWE types introduced by users.

Table B.11: Side-by-side summary of the main claims in the original paper by Zhang et al. (118) for their **RQ3** and claims based on the results of our replication using *SOTorrent22* and Cppcheck v2.13.

Original (SOTorrent18)	Replication (SOTorrent22)
RQ3 *What are the Characteristics of the Users who Contributed to Code With Weaknesses?*	
The majority of the C/C++ $Version_w$ was contributed by a small number of users. 72.4 percent (i.e., 10,652) of $Version_w$ were posted by 36 percent (i.e., 2,292) of users. 64.0 percent (i.e., 4,070) of the users who contribute $Version_w$ have contributed only one $Version_w$.	The majority of the C/C++ $Version_w$ were contributed by a small number of users. 79.5 percent (i.e., 38,481) of $Version_w$ were posted by 36 percent (i.e., 4,625) of users. 86.2 percent (i.e., 11,077) of the users who contribute $Version_w$ have contributed only one $Version_w$. (See Figure 4.7)
Among all the 85,165 users who posted C/C++ code snippets, only 7.5 percent (i.e., 6,361) of them posted code snippets that have weaknesses.	Among all the 75,779 users who posted C/C++ code snippets, 17.0 percent (i.e., 12,845) of them posted code snippets that have weaknesses.
More active users are less likely to introduce $Code_w$. The weakness density of a user's code drops when the number of contributed code revisions by the user increases. Users with higher reputations tend to introduce fewer CWE instances in their contributed code versions.	A Pearson correlation analysis and a linear regression indicate only a statistically significant but weak negative relationship between the number of revisions and weakness density. A Pearson correlation analysis and a linear regression show an extremely small effect size for reputation as a predictor for the number of CWE instances.
78.0 percent of users contribute code with only one CWE type. Furthermore, 42.2 percent (i.e., 2,686) of the users contribute only one CWE instance in all their $Version_w$. 81.8 percent (i.e., 5,206) of the users contribute less than five CWE instances in all their $Version_w$.	74.6 percent of users contribute code with only one CWE type. Furthermore, 50.9 percent (i.e., 6,533) of the users contribute only one CWE instance in all their $Version_w$. 85.3 percent (i.e., 10,938) of the users contribute less than five CWE instances in all their $Version_w$.
Users tend to commit the same types of CWE instances repeatedly. We observe that 37.7 percent of users are likely to introduce a single type of CWE instances in their posted code versions.	Users tend to commit different types of CWE instances with similar probabilities. We observe that 15.6 percent of users are likely to introduce a single type of CWE instances in their posted code versions, but 32.7 percent of users introduce several types of CWE. Further, there is a shift in the CWE types: CWE 457, 476, 758, and 788 are significantly more frequent, while CWE 908 disappeared.

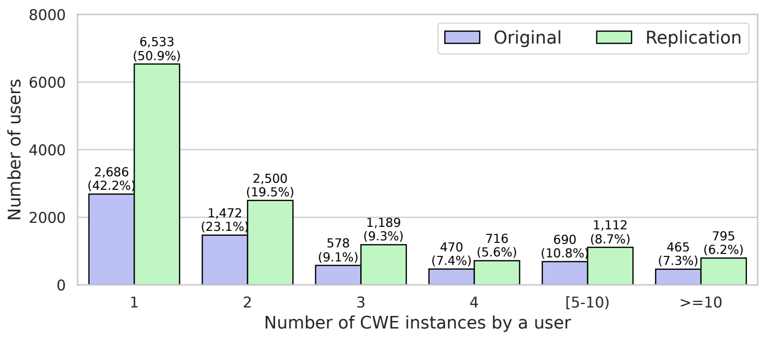


Figure B.16: The number/proportion of users contributing a different number of CWE instances. (cf. Figure 12 in (118))

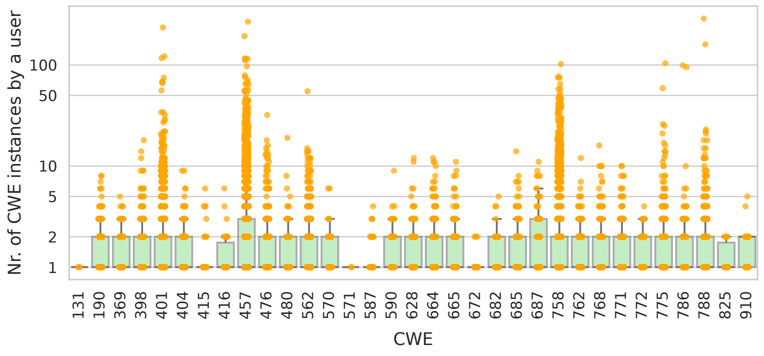


Figure B.17: The distribution of contributed CWE instances by different users for different CWE types. (cf. Figure 13 in (118))

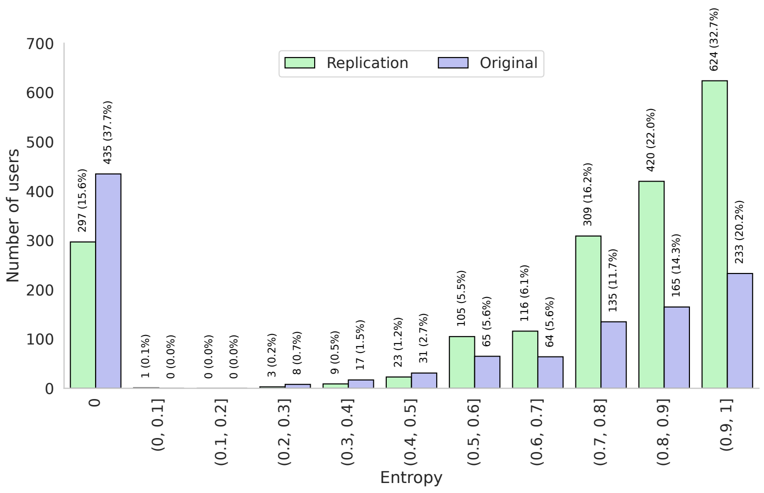


Figure B.18: The distribution of entropy for CWE instances of different types. (cf. Figure 14 in (118))

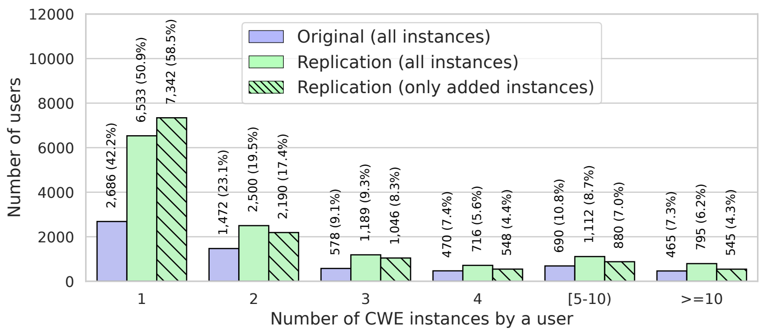


Figure B.19: The number/proportion of users contributing a different number of CWE instances, including a replication *that only considers newly added CWE instances for code revisions.*

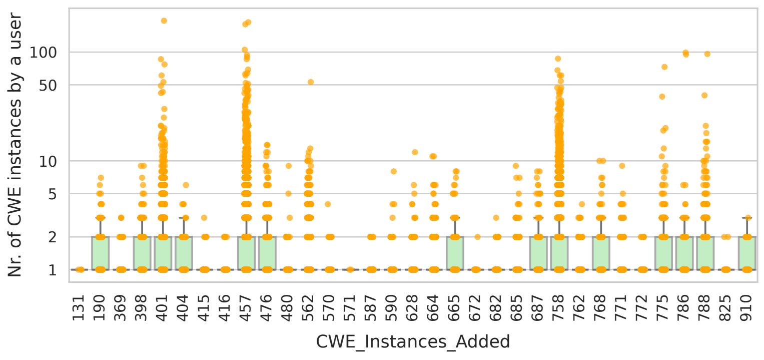


Figure B.20: The distribution of contributed CWE instances by different users for different CWE types *when only considering newly added CWE instances for code revisions.*

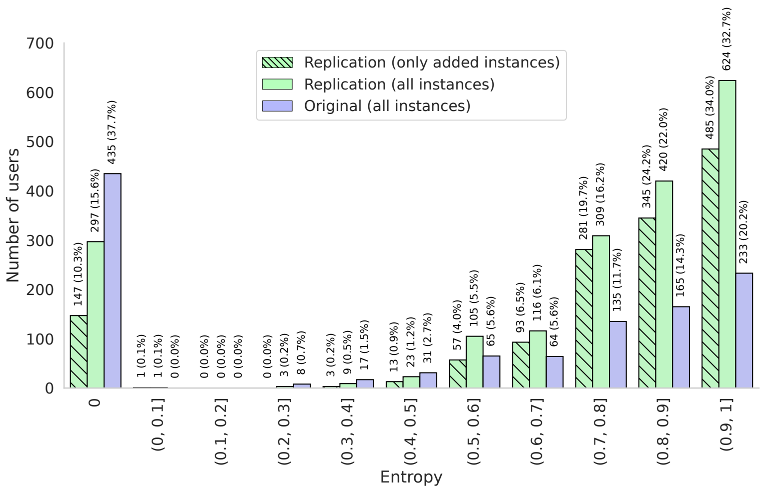


Figure B.21: The distribution of entropy for CWE instances of different types, including a replication *that only considers newly added CWE instances for code revisions.*

	Accuracy measurement result of Dicos for C/C++ *SOTorrent20* (***Original***)					Accuracy measurement results of Dicos for C/C++ *SOTorrent22* (***Replication***)				
ID	**#Posts**	**#TP**	**#FP**	**#TN**	**#FN**	**#Posts**	**#TP**	**#FP**	**#TN**	**#FN**
G1	731	704	27	N/A	N/A	746	92	654	N/A	N/A
G2	200	171	29	N/A	N/A	200	13	187	N/A	N/A
G3	100	82	18	N/A	N/A	300	26	274	N/A	N/A
G4	200	N/A	N/A	151	49	200	N/A	N/A	191	9
G5	100	N/A	N/A	92	8	100	N/A	N/A	98	2
Total	1,331	957	74	243	57	1,546	131	1115	289	11
Precision					**0.91**					**0.11**
Recall					**0.93**					**0.92**
Accuracy					**0.89**					**0.27**

Table B.12: Comparison of DICOS Discovery Accuracy for C/C++ posts: Precision, Recall, and Accuracy metrics measured by authors (Table 3 in (49)) vs. our Replicated Metrics (SOTorrent22)

	Accuracy measurement result of Dicos for Android *SOTorrent20* (***Original***)					Accuracy measurement results of Dicos for Android *SOTorrent22* (***Replication***)				
ID	**#Posts**	**#TP**	**#FP**	**#TN**	**#FN**	**#Posts**	**#TP**	**#FP**	**#TN**	**#FN**
G1	57	53	4	N/A	N/A	42	3	39	N/A	N/A
G2	200	175	25	N/A	N/A	200	20	180	N/A	N/A
G3	100	80	20	N/A	N/A	300	40	260	N/A	N/A
G4	200	N/A	N/A	167	33	200	N/A	N/A	188	8
G5	100	N/A	N/A	93	7	100	N/A	N/A	90	10
Total	657	308	49	260	40	842	63	479	278	18
Precision					**0.86**					**0.12**
Recall					**0.89**					**0.78**
Accuracy					**0.86**					**0.41**

Table B.13: Comparison of DICOS Discovery Accuracy for Android posts: Precision, Recall, and Accuracy metrics measured by authors (Table 4 in (49)) vs. our Replicated Metrics (SOTorrent22)

c#, .net, facebook, sockets, base64, javascript, php, web-services, string, exception, node.js, ios, python, encoding, spring, objective-c, file, padding, mysql, apache-httpclient-4.x, ruby, soap, rest, bytearray, algorithm, eclipse, android-asynctask, android-volley, arrays, json, performance, okhttp, hex, byte, android-studio, facebook-graph-api, http, jsp, xml, servlets, tomcat, hibernate, post, multithreading, apache, okhttp3, c, scala, jakarta-ee, amazon-web-services, groovy, websocket, iphone, amazon-s3

Table B.14: List of filtered tags

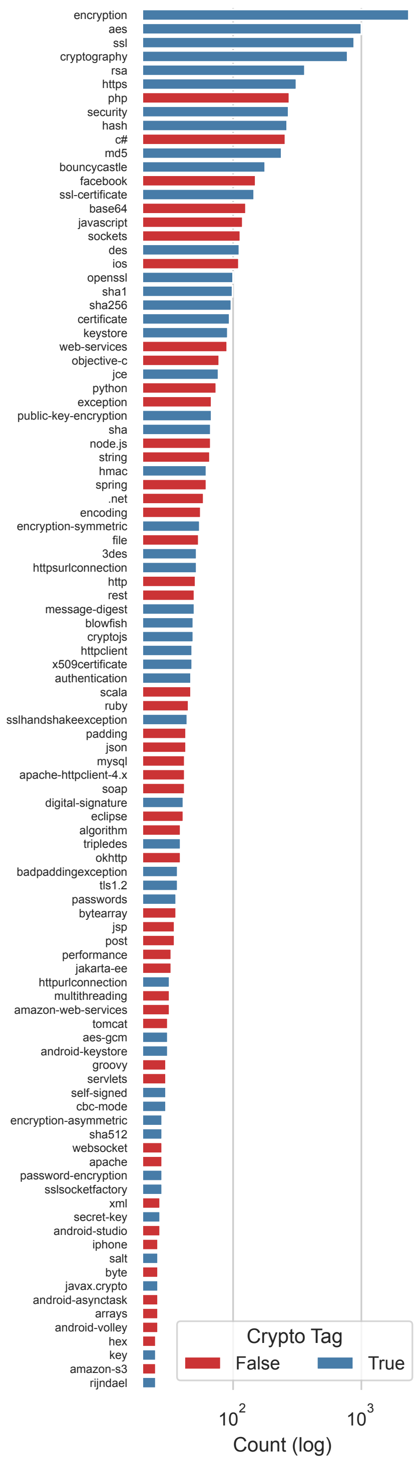
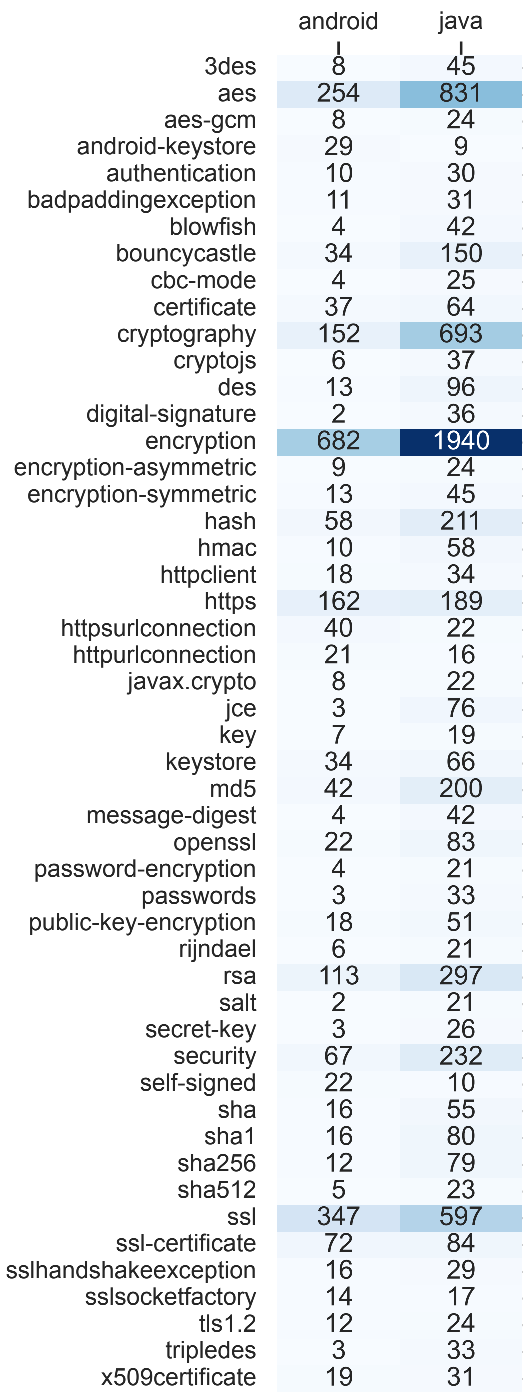


Figure B.22: Top-100 tags co-located with *Java* or *Android* tag in the data set by Fischer et al. Top-50 crypto-related tags are indicated.



	android	java
3des	8	45
aes	254	831
aes-gcm	8	24
android-keystore	29	9
authentication	10	30
badpaddingexception	11	31
blowfish	4	42
bouncycastle	34	150
cbc-mode	4	25
certificate	37	64
cryptography	152	693
cryptojs	6	37
des	13	96
digital-signature	2	36
encryption	682	1940
encryption-asymmetric	9	24
encryption-symmetric	13	45
hash	58	211
hmac	10	58
httpclient	18	34
https	162	189
httpsurlconnection	40	22
httpurlconnection	21	16
javax.crypto	8	22
jce	3	76
key	7	19
keystore	34	66
md5	42	200
message-digest	4	42
openssl	22	83
password-encryption	4	21
passwords	3	33
public-key-encryption	18	51
rijndael	6	21
rsa	113	297
salt	2	21
secret-key	3	26
security	67	232
self-signed	22	10
sha	16	55
sha1	16	80
sha256	12	79
sha512	5	23
ssl	347	597
ssl-certificate	72	84
sslhandshakeexception	16	29
sslsocketfactory	14	17
tls1.2	12	24
tripledes	3	33
x509certificate	19	31

Figure B.23: Frequency of co-location between top-50 crypto-related tags and *Java* and *Android* tags in the data set by Fischer et al.

Count	Tag1	Tag2	Rank
744	java	ssl	120
522	java	encryption	178
429	java	security	222
382	java	authentication	253
258	android	authentication	366
214	java	cryptography	435
210	android	security	445
207	java	bouncycastle	450
201	java	https	461
171	java	aes	556
170	java	ssl-certificate	562
156	android	ssl	605
153	java	httpclient	614
151	android	encryption	625
143	java	rsa	661
133	java	keystore	700
124	java	hash	757
116	java	certificate	796
116	java	digital-signature	796
110	java	httpurlconnection	837

Table B.15: Top-20 crypto-related tags co-located with *Java* and *Android* among all questions on Stack Overflow. Rank among all co-located tags

Count	Tag1	Tag2	Rank
25,006	java	spring-boot	1
24,173	android	kotlin	2
18,851	java	spring	3
11,363	android	android-studio	4
10,701	android	flutter	5
7,115	android	android-jetpack-compose	6
6,233	java	maven	7
6,002	android	firebase	8
5,591	android	react-native	9
5,449	java	hibernate	10

Table B.16: Top-10 tags co-located with Java and Android among all questions on Stack Overflow. Rank among all co-located tags.

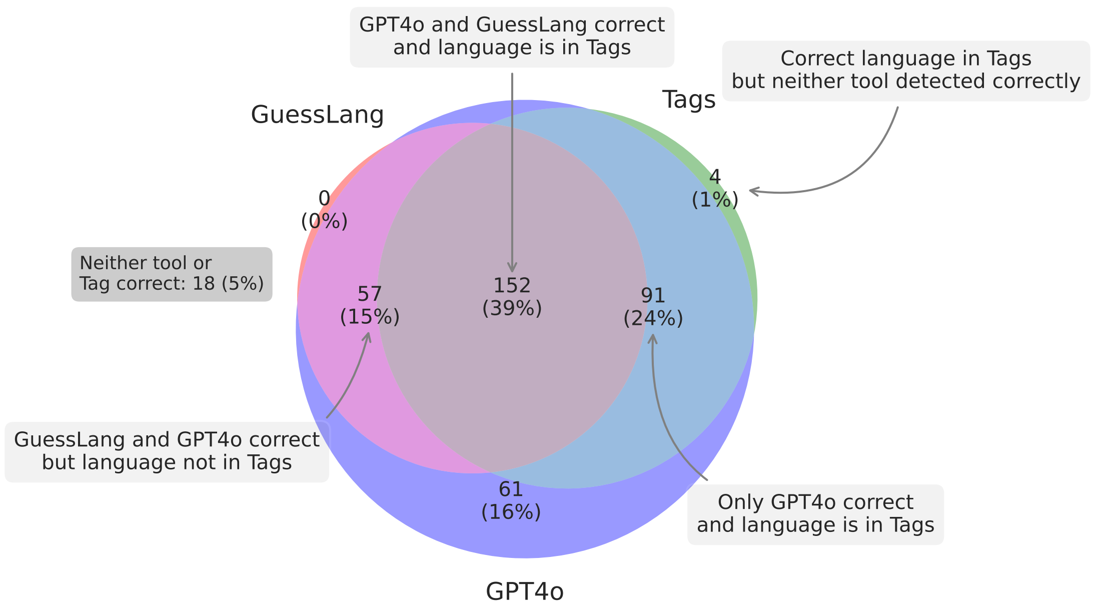


Figure B.24: Intersection between the manually verified language predictions by Guess-Lang, GPT4o, and post Tags.

Tool	Precision	Recall	F1
GPT4o	0.93	0.99	0.96
Guesslang	0.89	0.69	0.78
Tags	0.43	0.67	0.52

Table B.17: Performance of different approaches in detecting Python, JavaScript, Java, and C/C++ code snippets.