Universiteit Leiden
The Netherlands

Opleiding Informatica

Finding the Sources of
Vulnerable Code Patterns

Joris Rijnfrank

Supervisors:
Dr. Olga Gadyatskaya & Jafar Akhoundali

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

17/04/2025

Abstract

Developers often reuse code by cloning it from external sources. It is a practice that can introduce vulnerabilities if the copied code is not reviewed. This study investigates the sources of vulnerable code patterns by analysing URLs found in the comments of vulnerable code obtained from a dataset. The methodology involves extracting and filtering the URLs to identify platforms like GitHub and Stack Overflow as potential sources of cloned code. While the results show that these platforms are referenced, the study also shows limitations in confirming direct code matches and code clones. The findings show insights into developer practices and suggest the need for better code clone attribution and verification to reduce security risks associated with code cloning.

Contents

1 Introduction

Developers often engage in code cloning, also known as code reuse. They copy existing code into new projects. One of the main reasons why developers copy code is because they need specific functionality [JRM25]. Copying the code saves time and effort. Because the cloning accelerates development, it can also reduce development costs [FK05]. Code cloning is also prevelant in deep learning where the majority of clones are linked to the model construction [JRKM21]. While developers sometimes acknowledge the original source through comments, code cloning can introduce significant security risks if the reused code contains vulnerabilities. When vulnerabilities are intentionally planted through reused code, this constitutes a code reuse attack [Ble11].

Software vulnerabilities are weaknesses in code that can be exploited to compromise system security. These vulnerabilities can range from simple bugs to complex flaws. Some effects of vulnerabilities include unauthorised access, data breaches, and systems being unresponsive. Vulnerabilities from cloned code can come from several reasons. Developers may not properly check the copied code, there may be too little time to look at the code, or the original source may seem trustworthy despite having vulnerabilities. Figure 1 shows a hypothetical example of vulnerable cloned code with a buffer overflow vulnerability. The buffer declared on line 6 allocates space for ten characters, but line 8 allows input beyond this limit. The comment explicitly says that this function was copied from Stack Overflow. This shows how developers may copy code without checking it for vulnerabilities.

```
1 /**
2 * I didn't write this. I copied this function from the source
3 * Source: https://stackoverflow.com/a/314159265
4 */
5 void copiedFunction() {
6     char buffer[10];
7     std::cout << "Enter some text: ";
8     std::cin >> buffer;
9     std::cout << "You entered: " << buffer << std::endl;
10 }
```

Figure 1: Hypothetical vulnerable code reused from Stack Overflow.

The reliability of code sources varies significantly. Platforms like Stack Overflow host community-contributed content of inconsistent quality. Studies recommend caution when reusing code from such sources [MLOS20], as they may contain vulnerabilities that propagate through cloning. This problem is not just in question-and-answer platforms, but also open-source projects, code snippets in documentation and other snippets may propagate vulnerabilities. Vulnerable code can easily spread because of the general availability of these platforms.

1.1 Motivation

Cybersecurity is a critical concern in today's increasingly digital world. Over the past few decades, the importance of computer systems has grown. This also increases the potential impact of cyber attacks. The cost of a cyber attack can be massive. According to the FBI's 2023 Internet Crime Report, cybercrime resulted in over $12.5 billion in reported losses in the United States alone [Fed24].

One way to improve security is with early detection and elimination of software vulnerabilities. A big source of vulnerabilities is code cloning, where developers copy code from external sources without reviewing it well. This practice can introduce outdated and vulnerable code into systems.

Understanding the sources of these cloned vulnerabilities can provide important insight into how vulnerable code spreads. This can inform security researchers and developers and help them mitigate similar vulnerabilities in the future. Focusing on code cloning as a vulnerability vector supports the larger goal of creating more secure and resilient software.

1.2 Problem

The focus of this thesis is to find the sources of vulnerable code patterns by analysing uniform resource locators (URLs) in the comments of code. To do this, we need a dataset of vulnerable code. This dataset can be used to extract comments and URLs for analysis.

MoreFixes is presumably the largest dataset of real-world vulnerabilities with 26,617 unique Common Vulnerabilities and Exposures (CVE) identifiers [ANRG24]. These vulnerabilities were collected from open-source GitHub projects. Given the size of the dataset, it may contain vulnerabilities introduced by reusing code. This dataset can be used to analyse the vulnerabilities and determine the sources of vulnerable code patterns.

To test the methodology of extracting comments and URLs to find sources of vulnerable code patterns, MoreFixes will be used to provide the data. First, a short overview of the relevant data in MoreFixes will be given. Then will be explained how comments are extracted from the vulnerable source code. After that will be shown how URLs are extracted from the comments and then what is stored in the final database. The extracted URLs from the different files will be analysed for interesting patterns and information.

1.3 Structure

First, we will look at related work on this subject. This will show us where we currently are and what has been done in this area. After the literature review, the methodology will be explained in Chapter 3. It will explain the steps and design to execute the research. The MoreFixes dataset will be explored and the useful data for this research will be isolated since the dataset contains more than is needed for this research. In the methodology, how comments are extracted from code and URLs are extracted from the comments will also be explained. Lastly, the design of the database that will store all the URLs will be explained. Once the methodology has been explained, the results will be discussed and a conclusion will be drawn. This conclusion is also combined with ideas for potential further research.

2 Related Work

This chapter reviews existing work in vulnerability detection, code clone analysis, and studies on insecure code sources such as Stack Overflow to determine the relevance of finding the sources of vulnerable code introduced by code cloning.

Software vulnerability detection is an important area of research because of the increasing complexity and frequency of attacks. State-of-the-art detection approaches use a variety of techniques, like traditional rule-based methods, static and dynamic analysis and advanced machine learning models. Tools like VulDeePecker [LZX+18] and Devign [ZLS+19] use deep learning to identify vulnerabilities in source code by learning patterns from large datasets. Traditional tools like Frama-C [KKP+15] and Valgrind [NS03] use static and dynamic analysis to detect memory safety issues. Recent research [CKDR22] shows significant limitations in current deep learning-based vulnerability detection systems. It shows that models often don't generalize well to real-world scenarios. Performance drops up to 73% when it is used outside their training datasets. Key challenges include data duplication, unrealistic dataset distributions, and models learning irrelevant features instead of actual vulnerability patterns. To fix these issues, they propose ReVeal. It achieves up to a 33.57% boost in precision and a 128.38% boost in recall compared to existing methods.

Analysis showed that code clones are often not harmful, and aggressive refactoring isn't necessary [KSNM05]. Many exist only for a short time or get changed to provide updates. Often, it would not be possible to refactor the code easily to remove the code clone. It was found that while clones can increase maintenance effort in some cases, only about half of the cloned methods showed a significant increase in effort and the impact varied depending on the system and context [LW08]. Only a small percentage of clones are refactored between releases [KMB+22]. Many refactorings are done to improve the code quality, and not to remove the clones themselves. It was argued that cloning can be a reasonable design decision [KG08]. It offers benefits such as code reuse and independent evolution. It was also found that many clones do not require consistent maintenance and are not bug-prone [HWP+21], supporting the idea that clones are not always bad. A more recent study [MRRS19] showed a different view. It showed that up to 21% of bug-fixing changes in clones may be due to bugs propagated through cloning and were not created independently. The study found that clones created in the same commit and file were more likely to contain bugs compared to being in different files. These findings highlight that while cloning can be beneficial, it may also introduce subtle errors that propagate silently. Similarly, it was found that found that 17% of type-3 (inconsistent) clones in industrial systems contained documented faults [WAKP16]. The fault rate varied significantly depending on developer awareness and practices. Their findings show that while many clones are handled correctly, a lack of clone awareness can lead to incomplete fixes and bugs. While it is possible to have responsible code cloning, a significant portion of clones can contain bugs. Detecting clones is important to reduce code cloning bugs.

Multiple studies have been done to detect code clones. One of these studies proposed VUDDY [KWLO17]. It is a scalable and accurate method for detecting code clones. An important property of VUDDY is its high scalability, which allows it to process a billion lines of code in 14 hours and 17 minutes. The code clone detection is at the function level. To optimise VUDDY, the functions are filtered depending on their length. This reduces unnecessary comparisons. It also applies abstraction to detect code clones that may have been slightly altered. It does this without significantly increasing

false positives. VUDDY is an improvement on previous research. One previous method that VUDDY compares itself against is CCFinder. CCFinder does code clone detection at token level [KKI02]. A token is the minimum unit the compiler can understand, like a keyword or number. Compared to detection using line-by-line, this handles changes to formatting like extra line breaks. The token level is also less computationally expensive compared to abstract syntax tree matching. It is very flexible for different languages. Newer methods such as transformer-based approaches like CloneXformer [NAEAK25], use abstract syntax trees and control-flow graphs to detect semantic clones which token-based tools like CCFinder may miss. These methods often trade scalability for precision. Systematic reviews [ZNPR+23] highlight that many clone detection tools struggle with multi-language support and computational overhead, areas where VUDDY's design excels. Deep learning techniques, like those surveyed in [LLL+22], show promise for cross-language detection but require heavy resources, underscoring the value of VUDDY's efficiency for large-scale analysis.

These studies address the code clones but do not focus on vulnerabilities specifically. Since Stack Overflow is often a source of code cloning, research has been done to determine the state of vulnerabilities within the posts of the platform [LN21]. The authors extracted 404,779 code snippets from the platform and 8,010 Java code snippets were analysed using FindBugs [HP04] to find vulnerabilities. Of these snippets, 59 had a total of 75 security issues, with 21 of these being from accepted answers. This, however, means that most of the code snippets on Stack Overflow were found to be secure. A different paper proposed Dicos [HWL21]. This method differs in that it uses the change history of posts to detect potential vulnerabilities. To flag a post as insecure, it determines if a change was a security fix. There are three key features that it uses to detect this, changes in security-sensitive APIs, security-related keywords and changes in the control flow. With this method, it was found that 151 out of 2,000 popular open-source projects had insecure code that was copied from Stack Overflow.

Recent work has also focused on collecting real-world vulnerability data from open-source repositories. The MoreFixes dataset [ANRG24] is the largest known collection of CVE fix commits across 6,945 GitHub projects and 26,617 CVEs. The dataset includes not only code changes but also metadata such as CWEs, commit messages, and repository information. MoreFixes includes potential fix commits identified via heuristic-based matching using the Prospector tool, for coverage beyond direct CVE references. These real-world vulnerabilities will be used to understand if there are indications of code cloning from vulnerable code.

There has been clear progress in vulnerability detection and code clone analysis. However, few studies focus on tracing the sources of vulnerable clones. Tools like VUDDY and deep learning approaches improve clone detection but they do not specifically address the security implications of copied code. Research on Stack Overflow shows that vulnerable snippets exist, but how those are copied into real projects is not often studied. This research bridges the gap between clone detection and vulnerability sourcing. The goal is to understand whether there are indications of code cloning from vulnerable code captured in the MoreFixes dataset.

3 Methodology

To research the sources of vulnerable code snippets, a dataset of vulnerabilities will be used. The dataset will be used to extract the comments from the vulnerable code. URLs will be extracted from these comments, and these URLs will be filtered with an allowlist. URLs will be extracted because authors may reference the source of code that they copied in comments. If there are multiple references to the same source, then that can indicate that the source contains vulnerable code.

Figure 2 shows a step-by-step plan that explains how the sources of the vulnerable code are discovered. The rectangles explain what type of data each step has. The arrows explain how the data is used to get to the next step.

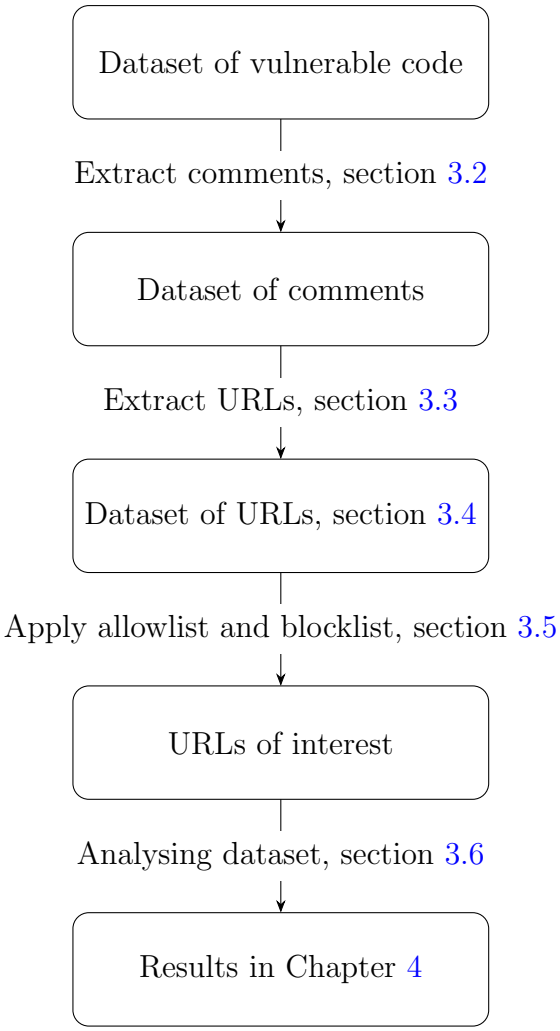

Figure 2: Overview of the steps that have to be taken to extract the URLs of interest.

The dataset of vulnerable code to be used for this research will be MoreFixes. This is a dataset of commits that fix vulnerabilities in open-source projects. The file changes of each commit are also present in the dataset. These file changes will be used to extract comments and URLs. To be able to effectively extract the comments, it is necessary to know in what programming language the file

is written. Different languages have different syntaxes for comments. The `file_change` table in MoreFixes also has this information. Together with information to uniquely identify the files, it works perfectly to test this methodology. Other datasets that contain the same information could also be used. Table 1 shows the `file_change` table of MoreFixes. The columns that are not used for this research are grey.

file_change
file_change_id
hash
filename
old_path
new_path
change_type
diff
diff_parsed
num_lines_added
num_lines_deleted
code_after
code_before
nloc
complexity
token_count
programming_language

Table 1: `file_change` table of MoreFixes with the used columns highlighted.

The old path and code before the commit are used since the new code might have any URLs linking to the source removed. The dataset also contains a `method_change` table. This table shows the isolated code of the vulnerable method that was fixed. However, this cannot be used because the isolated method starts at the method signature, and it's not uncommon for comments about a method to appear above the signature. Comments as such would not be included in the code stored in the table. For this reason, the `file_change` table is used.

The `cwe` table was also used for the new database. This was done to conserve a link between the files and what vulnerabilities they contain. The common weakness enumeration (CWE) is a list of common weaknesses. They reference a general weakness, not specific to an implementation. Since they are added along with the file changes, analysis can be done between URLs and CWEs.

3.1 Choosing Supported Languages

The `file_change` table contains 103703 file changes. There are 55 different programming languages in the table. The programming languages with the top ten most changes are shown in Table 2. It shows how often a change occurred in the table and how much percentage it is of the total number of file changes.
The top ten most common languages cover 78273 file changes. However, Markdown is a file type where text is described. A program cannot be created with Markdown, it is mostly used for

Language	Number of file changes	% of total
C	17898	17.3%
PHP	14811	14.3%
Java	9634	9.3%
Python	7233	7.0%
Go	6138	5.9%
C++	5941	5.7%
JavaScript	5691	5.5%
Markdown	3921	3.8%
Ruby	3516	3.4%
TypeScript	3490	3.4%

Table 2: Top ten most common languages for the file changes.

changelogs or read-me files. For this reason, it is not considered for this methodology. With the top nine programming languages there are still 74352 file changes. This is 71.7% of all the file changes. The remaining 28.3% of the file changes can be split up into 10.0% being of programming languages and 18.3% being changes of data files similar to Markdown.

For this methodology, the focus is on supporting the top nine programming languages (C, PHP, Java, Python, Go, C++, JavaScript, Ruby, and TypeScript), as they provide a dataset of 74,352 file changes (71.7% of the total). The remaining programming languages account for 10.0% of the changes. Future work could integrate more languages to increase the size of the analysis.

3.2 Extracting Comments

To get the URLs from code comments, we first have to extract the comments. Different languages have different syntaxes for comments, however, a comment always has a recognisable beginning pattern and an end. The ending of a comment may be a newline character or a more specific pattern.

Many programming languages share the same syntax for comments. Six of the nine supported programming languages use the same syntax for comments. Four different functions were implemented to cover the nine languages. The separate functions were made to ensure that only a single pass through the string has to be made to extract all comments. This improves performance while keeping modularity by having different functions for different syntaxes.

To minimize overhead and simplify support for additional languages, full parsing or tokenization of the code is avoided when extracting comments. The context of the code itself is not relevant so parsing is not necessary. Instead, comments are identified by scanning the code character by character, with a limited lookahead to detect comment syntax. This approach does not guarantee perfect accuracy. Some comments may be missed, and non-comment text may occasionally be misclassified. However, it provides a good, fast and lightweight approximation that is easy to adapt to new languages without complex parsing logic.

An important issue to tackle is strings in code. They may contain the syntax for a comment, however, it is in a string so it should not be read and extracted as a comment. To make sure this is taken into account, the algorithm keeps track of whether it is currently reading the characters in a string. If that is the case, it will skip through the characters and not recognise the beginning of a comment until it reaches the end of a string. The algorithm created is abstractly described in Algorithm 1.

Algorithm 1 Extract Comments from Code

```
Initialise an empty list comments
Initialise a boolean in_string to False
Initialise an empty string string_terminator
for every character in code do
    if in_string then
        if there is a backslash, indicating an escaped character then
            Skip ahead two characters in code
        else if the character is equal to string_terminator then
            Set in_string to False
        end if
    else if character is equal to the beginning of a string then
        Set in_string to True
        Store the character that opened the string in string_terminator
    else if character and look ahead matches the beginning of a single-line comment then
        Move to the end of the line while storing the comment characters
        Store the comment in the comments list
    else if character and look ahead matches the beginning of a multi-line comment then
        Move until you find the end of the multi-line comment
        Store the comment in the comments list
    end if
end for
return the list comments
```

The syntax used to match the character and look ahead to the starting and ending of a comment is dependent on the syntax for comments of the programming language. The exact syntax used here is listed below per language. Some specific languages may require slight modification of the algorithm, but Algorithm 1 is the base that all comment extraction is based on.

- Syntax for C, Java, C++, JavaScript, Go and TypeScript

  Single-line comment: A single-line comment starts with `//` and automatically ends at the end of the line.

  Multi-line comment: A multi-line comments starts with `/*` and continues through all characters and newlines until `*/` is seen.

- Syntax for PHP

Single-line comment: PHP has two versions of a single-line comment. A single-line comment can either start with //, like the C syntax, or it can start with #. It ends automatically at the end of the line.

Multi-line comment: A multi-line comments starts with /* and continues through all characters and newlines until */ is seen. The same as the C syntax.

- Syntax for Python

  Single-line comment: For Python, a single-line comment starts with # and ends automatically at the end of the line.

  Multi-line comment: Python does not natively support any syntax for multi-line comments, however, Python does support multi-line strings. A multi-line string that is not used in code and not stored in variables is often used as a multi-line comment. This gives us the following definition for a multi-line comment in Python.
  A multi-line comment starts with """. There are no text characters in front of the opening on the same line, at most only white space. A comment ends with """.

- Syntax for Ruby

  Single-line comment: For Ruby, a single-line comment starts with # and ends automatically at the end of the line.

  Multi-line comment: A multi-line comment starts with `=begin` and it ends when `=end` is seen.

3.3 Extracting URLs

After the comments have been extracted from a file. They are processed so the URLs can be extracted from them. To extract a URL, a regular expression was used. A regular expression defines a pattern of text. After defining the expression, it is possible to match and check it against text. This is often used for validation and in this case, used to find URLs in the comments. Figure 3 shows the regular expression that was created to extract the URLs. The regular expression is split up into five parts with each part on a new line. These new lines have only been added for readability. The actual regular expression has the parts directly concatenated.

```
1. https?:\/\/
2. (?:\w+:\w+@)?
3. (?:[\d.]{7,15}|\[[:\da-f]{3,39}\]|[a-z\d][a-z\d\-.]{0,252})
4. (?::\d{1,5})?
5. (?:\/[\w\-.~!$&'()*+,;%=:@?#]*)+
```

Figure 3: The regular expression used to match URLs.

Each part of the regular expression was created to match a specific part of a URL. This regular expression is quite strictly created to ensure matched strings are valid URLs. The list below explains the parts of the regular expression.

- `https?:\/\/`

  Matches `http://` or `https://`.

- `(?:\w+:\w+@)?`

  Matches optional URL userinfo of the format `username:password@`.

- `(?:[\d.]{7,15}|\[[:\da-f]{3,39}\]|[a-z\d][a-z\d\-.]{0,252})`

  Matches the hostname of the URL. Consists of three potential formats.

  `[\d.]{7,15}` Loose match for an IPv4 address.

  `\[[:\da-f]{3,39}\]` Loose match for an IPv6 address in square brackets.

  `[a-z\d][a-z\d\-.]{0,252}` Loose match for a domain name.

- `(?::\d{1,5})?`

  Matches optional port specification of the format `:12345`.

- `(?:\/[\w\-.~!$&'()*+,;%=:@?#]*)+`

  Matches `/` followed by any of the allowed characters in the path, query and fragment string.

Using this regular expression, the URLs are extracted from the comments. To parse the extracted URLs, the Python library `urllib.parse` was used. This allowed for a more strict and correct method to destructure the URL into its parts.

3.4 Database Design

A PostgreSQL database is used to store the CWEs that are connected to the source file and extracted URLs. Using PostgreSQL was done to support easy analysis. Using SQL queries, the structured data can be transformed and conditionally returned to gain insight into the extracted URLs.

The database consists of three main tables and helper tables. These helper tables were necessary to form a many-to-many relationship. The three main tables are the `file_change`, `url` and `cwe` tables. The many-to-many relationship helper tables are between `file_change` and `url`, and `file_change` and `cwe`. This is because a single file in `file_change` can have multiple URLs, but a URL can also be referenced by multiple files. The same goes for a CWE. A file can be associated with multiple CWEs, and a CWE can be associated with multiple files. The helper tables that allow these many-to-many relationships are called `change_cwe` and `file_url`.

Figure 4 shows a graphical overview of the tables, their columns and the relationship between the tables. A column name in bold means it is part of a key for that column. The column name will also be suffixed with a key symbol.

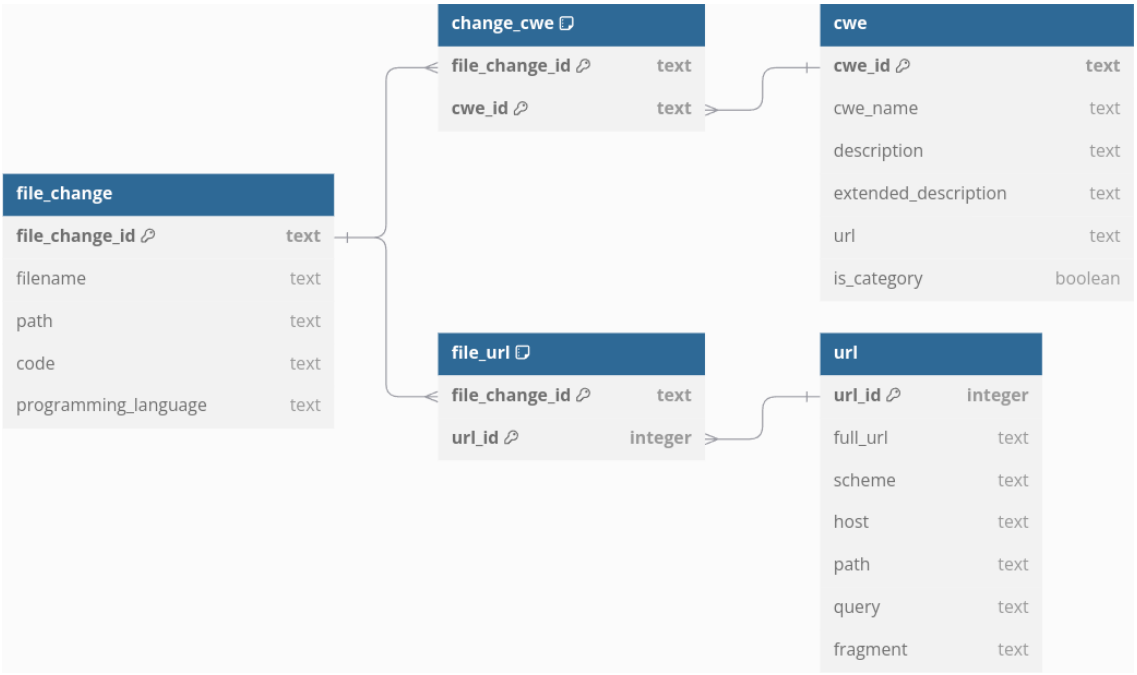

Figure 4: A diagram showing the tables of the database and their relationship to other tables.

3.4.1 Table `file_change`

The `file_change` table contains any changes to files that were part of a commit in the original dataset. The content of the columns is directly copied from the original dataset. It is not altered. Not every column from the `file_change` table in the MoreFixes dataset is present in this database. This is because the dataset has been designed to be minimal and information about the changes that the commit brought did not need to be included. The column `code` in this table is the same as `code_before` in the MoreFixes dataset. The column of `old_path` in the original dataset has been renamed to `path` to reflect this as well.

3.4.2 Table `cwe`

This table has the exact same structure as the original dataset. No columns were renamed or removed. However, this table does not contain every entry that MoreFixes has by default. Only CWEs are added for which files were found that have URLs in comments. This table was added to allow for analysis based on the type of vulnerability. URLs, or part of them, can be associated with specific vulnerabilities because of this table.

3.4.3 Table `url`

This is the table that contains the URLs that were found in the files. Each URL gets its own identification number and the exact URL that was found is placed in the `full_url` column. However, the URL is also parsed and destructured into its separate parts. This allows for analysis specific to the different parts of the URL. For example, only the host of URLs can be used to find suspicious

domains, if they have an unexpected appearance rate. Further explanation on the different parts of the URLs can be found in section 3.3.

3.5 URL Allowlist and Blocklist

Many of the URLs in the dataset will be irrelevant to code cloning. To address this, an allowlist-based filtering approach is used. An allowlist is a manually created set of rules that define which URLs are relevant or not. Any URL that does not match these rules is filtered out of the list. Since there is no inherent method to automatically determine the relevance of a URL for code cloning, a human is required to create the allowlist, making sure that only relevant sources of code snippets are kept.

The allowlist is applied to two parts of the URL:

- **Hostname Patterns**: Domains known to host code snippets or technical discussions.
- **Path Patterns**: URL paths that typically contain source code files or executable examples.

A blocklist is used to exclude paths that do not have useful code. For example, license files or documentation, even if the files are on allowed domains. The hostname allowlist consists of domains that often contain code snippets, such as version control platforms, developer forums, and technical documentation sites. Table 3 provides a categorized list of included domains.

Table 3: Hostname Allowlist

Category	**Domains**
Code Hosting	`github.com`, `gitlab.com`, `bitbucket.org`, `gist.github.com`
Developer Forums	`stackoverflow.com`, `reddit.com`, `dev.to`
Pastebins	`pastebin.com`, `hastebin.com`

Paths are filtered using regular expressions to match common code-related structures. The following patterns are included:

- File extensions, for example, `.*\.(py|js|java|cpp)$`
- Code directories, for example., `.*/src/.*` or `.*/examples?/.*`
- Version control paths, for example, `.*/blob/.*` for GitHub

To avoid false positives, certain paths are explicitly excluded with a blocklist, even if they originate from allowed domains. These include:

- License files, `.*/LICENSE.*`
- Documentation, `.*/README.*` or `.*/docs?/.*`
- Non-technical content, `.*/issues/.*` or `.*/blog/.*`

The filtering process is implemented using regular expressions. A URL is retained only if:

1. Its path does not match any blocklist pattern.
2. Its hostname matches any pattern in the hostname allowlist.
3. Or its path matches at least one allowed pattern.

3.6 Analysing URLs

After the URLs are extracted from the comments, the final dataset needs to be queried and analysed. Since it will not be feasible to analyse every URL manually due to the dataset size, some targeted queries will be performed to identify patterns and sources of vulnerable code. The analysis will mainly be done on the filtered dataset obtained through the hostname allowlist and path pattern rules.

The following queries will be used to analyse the dataset:

- **Top ten hosts with the most unique URLs before filtering**: This query identifies the most referenced domains across all extracted URLs, providing a baseline for comparison against the filtered results.
- **Top ten hosts with the most unique URLs after filtering**: This highlights the most frequent code-sharing platforms or repositories that remain after applying the filtering criteria.
- **Top ten hosts referenced by unique files**: This query counts how many different files in the dataset reference URLs from each host, identifying domains that appear frequently across different code snippets.
- **Top ten URLs referenced by files**: This identifies full URLs that are referenced by multiple files, which may indicate commonly cloned or reused code examples.
- **Top hosts by CWE frequency**: Shows which domains are most frequently linked to specific types of vulnerabilities.
- **URLs linked to multiple CWEs**: Identifies individual URLs that appear in files associated with multiple vulnerability types.

4 Results

After extracting the comments and URLs from the top nine programming languages of the MoreFixes dataset, the unfiltered database contains a total of 17,743 unique URLs from 3,283 different hosts. This is an average of approximately 5.4 unique pages per host. The most frequently occurring host is `github.com`, which has 3,404 distinct URLs. The ten most commonly appearing hosts in the unfiltered dataset are shown in Table 4.

Host	Number of unique URLs
github.com	3404
docs.aws.amazon.com	1498
www.w3.org	702
tools.ietf.org	688
stackoverflow.com	398
core.telegram.org	320
en.wikipedia.org	266
developer.mozilla.org	258
bugzilla.remotesensing.org	221
php.net	167

Table 4: Top ten hosts with the most unique URLs before filtering.

After applying the filtering with an allowlist the dataset was reduced to 2,338 unique URLs with 85 hosts. This significantly narrows the scope of analysis and increases the average number of pages per host to 27.5. The new top ten most frequent hosts in the filtered dataset are presented in Table 5.

Host	Number of unique URLs
github.com	1760
stackoverflow.com	398
gist.github.com	39
golang.org	13
gitlab.com	9
csrc.nist.gov	8
huggingface.co	6
chromium.googlesource.com	6
bitbucket.org	5
aospxref.com	4

Table 5: Top ten hosts with the most unique URLs after filtering.

As expected, the filtering process removes many generic or documentation-focused domains, concentrating the dataset on platforms more likely to be associated with code reuse. All subsequent analyses are based on this filtered set of URLs.

In addition to counting unique pages per host, it is also interesting to look at how many unique files per host. Table 6 lists the top ten hosts by the number of unique files in which they are referenced.

While the top entries overlap with those in the previous table, differences toward the end suggest that some hosts are referenced more across many files, even if they don't have as many unique URLs.

Host	Number of times referenced by unique files
github.com	2086
stackoverflow.com	330
gist.github.com	40
golang.org	13
gitlab.com	9
crypto-js.googlecode.com	7
bitbucket.org	4
dev.to	3
huggingface.co	3
pastebin.com	2

Table 6: Top ten hosts referenced by unique files.

To better understand which specific URLs are referenced most frequently, Table 7 lists the ten most commonly referenced URLs based on the number of different files in which each appears. Some of these URLs point to open-source projects or documentation pages that are referenced repeatedly, probably showing popular sources for reused or cloned code.

Full URL	Number of times referenced by files
https://github.com/FreeRTOS	656
https://github.com/GeyserMC/Geyser	118
https://github.com/semplon/GeniXCMS	106
https://github.com/leofeyer	51
https://github.com/janeczku/calibre-web)	41
https://github.com/firefly-iii).	31
https://github.com/junit-team/junit/wiki/Parameterized-tests	27
https://github.com/kantega/notsoserial	24
https://github.com/pluginsGLPI/order	23
https://github.com/Jasig/mod_auth_cas	20

Table 7: Top ten URLs referenced by files.

Table 8 explores how frequently certain hosts are associated with specific types of vulnerabilities. This is done by looking at the number of files linking a host to a given CWE. The results show that `github.com` is associated with a wide range of vulnerabilities. However, platforms like `stackoverflow.com` also appear, showing how developers could be influenced by code found in public forums.

Host	CWE ID	Name of CWE	File count
github.com	NVD-CWE-noinfo	Insufficient Information	1026
github.com	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	330
github.com	CWE-287	Improper Authentication	268
github.com	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	124
stackoverflow.com	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	110
github.com	CWE-20	Improper Input Validation	88
github.com	CWE-502	Deserialization of Untrusted Data	83
github.com	CWE-787	Out-of-bounds Write	82
github.com	CWE-918	Server-Side Request Forgery (SSRF)	75
github.com	CWE-352	Cross-Site Request Forgery (CSRF)	68

Table 8: Top hosts by CWE frequency.

Some unique URLs appear in files that are linked with multiple different CWEs. These URLs may represent code samples that are reused in diverse contexts or that have multiple weaknesses. Table 9 highlights the top ten URLs that are associated with the highest number of distinct CWEs.

Full URL	CWE count
https://github.com/WebAssembly/design/blob/master/JS.md#webassemblyinstance-constructor	11
https://github.com/python/cpython/pull/24297/files	11
https://www.virtualbox.org/svn/vbox/trunk/src/VBox/VMM/VMMR0/HMR0.cpp	10
https://github.com/DolbyLaboratories/dlb_mp4base/blob/70a2e1d4d99a8439b7b8087bf50dd503eeea2291/src/esparser/parser_hevc.c#L1233	10
https://github.com/electron/node-is-valid-window	9
https://github.com/apache/airflow/pull/26658	9
http://gist.github.com/urraka/685d9a6340b26b830d49	9
https://github.com/squid-cache/squid/pull/81#discussion_r153053278	9
https://github.com/piwik	8
https://github.com/matrix-org/synapse/pull/1744	8

Table 9: URLs linked to multiple CWEs.

To determine the effectiveness of this methodology. Manual review of a sample of URLs was performed. It is not a random sample. The URLs were filtered to only include URLs with the host `stackoverflow.com` or `gist.github.com`, excluding links to other content such as GitHub repositories. This puts the focus on Q&A services. Table 10 shows the top twenty most commonly referenced URLs in this subset, including the number of CWEs that are connected to that URL. These URLs were manually reviewed.

The review reveals that the first URL, https://gist.github.com/4325783, is a vulnerable code snippet. It was created in December 2012. Comments on the gist mention that it has been used in CPython, the reference implementation of the Python programming language. A comment made in July 2024 explains that the gist has a security issue and that the issue has been fixed in CPython. This directly shows that code cloning has been responsible for introducing a vulnerability.

Another URL that mentions a vulnerability is http://stackoverflow.com/questions/161738/what-is-the-best-regular-expression-to-check-if-a-string-is-a-valid-url. However, this URL links to a question and not a specific answer. The vulnerable answer has the third-highest score of the answers with 91 points. The second and first answers have 198 and 498 points. A

URL	File count	CWE count
https://gist.github.com/4325783	13	1
http://gist.github.com/urraka/685d9a6340b26b830d49	12	9
https://stackoverflow.com/a/24613980/6943581	9	7
https://stackoverflow.com/questions/33692969/assembler-mmx-errors	8	2
https://stackoverflow.com/a/30669632	8	2
https://stackoverflow.com/a/16119722/561309	8	2
http://stackoverflow.com/questions/9107240/1-evalthis-vs-evalthis-in-javascript	8	5
http://stackoverflow.com/q/1661863/1114320	6	2
http://stackoverflow.com/a/11117338/333340	6	5
http://stackoverflow.com/questions/3191664/	5	4
http://stackoverflow.com/questions/1361149/get-img-thumbnails-from-vimeo/4285098#4285098	5	1
https://stackoverflow.com/a/17822709	5	2
https://gist.github.com/ozh/7951236	4	3
http://stackoverflow.com/questions/235504/validating-crontab-entries-w-php	4	3
http://stackoverflow.com/questions/324486/how-do-you-read-css-rule-values-with-javascript	4	4
https://stackoverflow.com/questions/4166762/php-image-upload-security-check-list	4	3
https://gist.github.com/ndarville/3452907#file-secret-key-gen-py	4	3
http://stackoverflow.com/a/14420217	4	3
http://stackoverflow.com/questions/161738/what-is-the-best-regular-expression-to-check-if-a-string-is-a-valid-url	4	1
http://stackoverflow.com/questions/764360/a-list-of-string-replacements-in-python	4	2

Table 10: Top twenty Stack Overflow and GitHub Gist URLs referenced by vulnerable files

reply below the vulnerable answer mentioned that the snippet is vulnerable to a cross-site scripting attack. This snippet may not have been cloned as much as the two other answers because they contain much more preferred answers, as seen by the scores.

The other URLs in Table 10 do not directly mention a security issue. The second URL, http://gist.github.com/urraka/685d9a6340b26b830d49, includes a comment posted six months after the gist was created. They ask a question about a memory leak that is in the original version of the gist. The gist was updated with a fix after a discussion with the creator about the origin of the memory leak. This issue can be seen as a vulnerability as well.

In the Stack Overflow URLs outside of the twenty shown in Table 10, a clear example of a source of vulnerable code is https://stackoverflow.com/a/1912522/694469.. This URL points to an accepted answer that originally had a vulnerable JavaScript function and was later edited to acknowledge that vulnerability and slightly improve its security. The current version of the answer begins with the following warning:

> **EDIT:** You should use the DOMParser API as Wladimir suggests, I edited my previous answer since the function posted introduced a security vulnerability.

While the warning improves the answer's security, the original vulnerable version was posted in 2009 and subsequently accepted. The mention of a vulnerability was edited in at the end of the answer in 2017. The warning at the beginning of the answer was made in 2019. This shows how vulnerable code can spread with platforms like Stack Overflow before being corrected. Code cloning would introduce this vulnerable code, which could then be exploited in any project that uses it. It also shows that the community is responsible for identifying and fixing these vulnerabilities, and this may not happen quickly. Vulnerable code can be embedded into many projects over the years when it isn't clearly identified as vulnerable. While the majority of the URLs do not mention or point to vulnerable code, the review shows that the density of URLs pointing to sources of vulnerable code is high in the final dataset.

5 Discussion

The goal of the methodology is to identify the sources of vulnerable code introduced through code cloning by analysing URLs in comments from a dataset of vulnerable code. While the approach gave some insights, the results show that the method has some limitations and may not be as effective as initially hoped. Despite some limitations, the methodology shows that developers do sometimes include source references in their code comments, mainly from platforms like GitHub and Stack Overflow. This methodology provides a good indication and heuristic for potentially code matching to get a more definitive idea of the source of a vulnerability.

The analysis of the MoreFixes dataset gave 17,743 unique URLs across 3,283 hosts in the unfiltered dataset. After applying the allowlist and blocklist filters, the dataset was narrowed down to 2,338 unique URLs from 85 hosts. The most frequently referenced hosts, such as `github.com` and `stackoverflow.com`, are as expected. These platforms are well-known sources of reusable code. However, the volume of irrelevant URLs shows a challenge. Developers often include URLs in comments for reasons unrelated to code cloning, such as code licenses, referencing documentation, or citing related work.

The URLs from platforms like GitHub and Stack Overflow in the filtered dataset suggest that some developers acknowledge code cloning by linking to its source. For example, the top referenced URLs point to specific repositories or code snippets, that could indicate cloning behaviour. However, this methodology does not provide direct evidence of code cloning because code snippets are not compared to their potential.

The analysis shows some patterns and trends in the copy-pasting strategies of developers:

- URLs found in comments imply that some developers credit their sources, at least for code copied from public repositories or forums. This is a positive finding, as it indicates an awareness of the origins of reused code.
- GitHub appears more frequently than other sources in the filtered dataset, which shows its important role in open-source development. Developers often reference GitHub repositories, possibly for specific functions.
- Stack Overflow also appears often, confirming its role as a resource for code snippets. However, the lack of granularity in the data makes it difficult to know if the referenced snippets were the actual sources of vulnerabilities or unrelated.

The methodology also provides important insights into developer behaviour and documentation practices. The presence of URLs in comments, even when not directly related to code cloning, shows that developers are documenting code sources and influences. This global heuristic could be used by tools to create better code clone detection systems or to help with vulnerability detection by associating known vulnerable sources with their potential clones.

5.1 Limitations

The methodology suffers from a few limitations that may make it less effective:

- Many URLs in comments do not directly indicate code cloning. They may reference documentation, issue trackers, or unrelated resources. This reduces the precision of the analysis.
- The approach does not verify if the referenced URLs actually contain code that matches the vulnerable code in the dataset. Without this step, it is impossible to confirm that the URLs represent actual sources of cloned code.
- The allowlist and blocklist are necessary to filter out unnecessary data, however, it is probably not perfect. URLs that are relevant may get filtered out and irrelevant URLs may get included.
- The MoreFixes dataset has vulnerabilities from open-source projects on GitHub, this may introduce more of a bias of URLs that link to GitHub compared to code not originating from GitHub.
- Even if developers explain that their code was cloned from a different source, they may not explicitly provide URLs in their comments. This methodology would miss these cases.

6 Conclusions and Further Research

This study looked at the sources of vulnerable code patterns by analysing URLs in comments from the MoreFixes dataset, a large dataset of vulnerabilities in open-source projects. The methodology explained extracting and filtering URLs to identify potential sources of code cloning, with the goal of understanding how vulnerabilities spread with cloned code. The results show GitHub and Stack Overflow as the most frequently referenced platforms, suggesting that developers often acknowledge code reuse by linking to these sources. However, the analysis also revealed limitations, as many URLs were unrelated to code cloning, and the methodology could not definitively know if the referenced code matched the vulnerable snippets in the dataset. Despite these challenges, the study provides good insights into developer practices and offers a heuristic for identifying potential sources of vulnerable code.

The findings show the importance of good code attribution and checking when cloning code from external sources. While some developers document their sources, the amount of vulnerabilities linked to reused code points out that more strict cloning practices are needed. The methodology can serve as a starting point for further research into code cloning and vulnerability propagation, particularly when combined with more advanced techniques like code similarity analysis.

6.1 Future Research

Several directions for future research emerge from this work:

- Integrating code comparison tools could verify if the referenced URLs contain code that matches the vulnerable snippets. This would provide direct evidence of code cloning and improve the methodology.
- Applying the methodology to a broader dataset, including private code or projects from other platforms. This could reduce potential bias and provide a more comprehensive view of code cloning practices.
- Developing machine learning models to automatically classify URLs based on their relevance to code cloning could improve the precision in the filtering process and reduce reliance on manual allowlists and blocklists.

By improving in these subjects, future research could build on this methodology to create more robust tools and practices for detecting and preventing vulnerabilities introduced through code cloning. This will contribute to the broader goal of improving software security and reducing the risks associated with code cloning.

References

[ANRG24] Jafar Akhoundali, Sajad Rahim Nouri, Kristian Rietveld, and Olga Gadyatskaya. Morefixes: A large-scale dataset of cve fix commits mined through enhanced repository discovery. In *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2024, page 42–51, New York, NY, USA, 2024. Association for Computing Machinery.

[Ble11] Tyler Bletsch. *Code-reuse attacks: new frontiers and defenses*. PhD thesis, North Carolina State University, 2011. AAI3463747.

[CKDR22] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2022.

[Fed24] Federal Bureau of Investigation. 2023 internet crime report, 2024.

[FK05] W.B. Frakes and Kyo Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[HWL21] Hyunji Hong, Seunghoon Woo, and Heejo Lee. Dicos: Discovering insecure code snippets from stack overflow posts by leveraging user discussions. In *Proceedings of the 37th Annual Computer Security Applications Conference*, ACSAC '21, page 194–206, New York, NY, USA, 2021. Association for Computing Machinery.

[HWP+21] Bin Hu, Yijian Wu, Xin Peng, Jun Sun, Nanjie Zhan, and Jun Wu. Assessing code clone harmfulness: Indicators, factors, and counter measures. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 225–236, 2021.

[JRKM21] Hadhemi Jebnoun, Md Saidur Rahman, Foutse Khomh, and Biruk Asmare Muse. Clones in deep learning code: What, where, and why?, 2021.

[JRM25] Mahmoud Jahanshahi, David Reid, and Audris Mockus. Beyond dependencies: The role of copy-based reuse in open source software development. *ACM Transactions on Software Engineering and Methodology*, January 2025.

[KG08] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, Dec 2008.

[KKI02] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.

[KMB+22] Jaweria Kanwal, Onaiza Maqbool, Hamid Abdul Basit, Muddassar Azam Sindhu, and Katsuro Inoue. Historical perspective of code clone refactorings in evolving software. *Plos one*, 17(12):e0277216, 2022.

[KSNM05] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 187–196, New York, NY, USA, 2005. Association for Computing Machinery.

[KWLO17] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, 2017.

[LLL+22] Maggie Lei, Hao Li, Ji Li, Namrata Aundhkar, and Dae-Kyoo Kim. Deep learning application on code clone detection: A review of current knowledge. *Journal of Systems and Software*, 184:111141, 2022.

[LN21] Sherlock A. Licorish and Thushika Nishatharan. Contextual profiling of stack overflow java code security vulnerabilities initial insights from a pilot study. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 1060–1068, 2021.

[LW08] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. In *2008 IEEE International Conference on Software Maintenance*, pages 227–236, 2008.

[LZX+18] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proceedings 2018 Network and Distributed System Security Symposium*, NDSS 2018. Internet Society, 2018.

[MLOS20] Sarah Meldrum, Sherlock A. Licorish, Caitlin A. Owen, and Bastin Tony Roy Savarimuthu. Understanding stack overflow code quality: A recommendation of caution. *Science of Computer Programming*, 199:102516, 2020.

[MRRS19] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. An empirical study on bug propagation through code cloning. *Journal of Systems and Software*, 158:110407, 2019.

[NAEAK25] Mona Nashaat, Reem Amin, Ahmad Hosny Eid, and Rabab F Abdel-Kader. An enhanced transformer-based framework for interpretable code clone detection. *Journal of Systems and Software*, 222:112347, 2025.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, 2003. RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

[WAKP16] Stefan Wagner, Asim Abdulkhaleq, Kamer Kaya, and Alexander Paar. On the relationship of inconsistent software clones and faults: An empirical study. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 79–89. IEEE, 2016.

[ZLS+19] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[ZNPR+23] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software*, 204:111796, 2023.