UNIVERSITY OF SALERNO

Department of Computer Science

Master of Science in Computer Science

MASTER'S DEGREE THESIS

Better Safe than Sorry: Investigating the Evolution of Vulnerable Code Snippets Copied from Stack Overflow

SUPERVISORS

Prof. **Fabio Palomba**

Dr. **Emanuele Iannone**

Dr. **Giulia Sellitto**

CANDIDATE

Grazia Varone

0522501064

Academic Year 2021-2022

Nothing in life is to be feared, it is only to be understood.

Now is the time to understand more, so that we may fear less.

Marie Curie

Abstract

Online programming discussion platforms such as Stack Overflow serve as a rich source of information for software developers, providing benefits. However, when it comes to code security there are some caveats to bear in mind: Due to the complex nature of code security, it is very difficult to provide ready-to-use and secure solutions for every problem. Developers often reuse Stack Overflow code in their GitHub projects. So, locating vulnerable statements in source code is crucial to assure software security because vulnerable code can flow easily across software repositories. High false positive rate is a big obstacle for traditional static analysis. In this work, we empirically study the presence of the Common Weakness Enumeration – CWE, in code snippets of C/C++ Stack Overflow posts through two approaches: static analysis and machine learning. We want to detect the line code where there is a vulnerability because it should be easier for developers to find possible bugs, static analysis already allows it, for machine learning a novel approach based on ensemble learning is used. Moreover, we explore the evolution of reused vulnerable code snippets in GitHub repositories and assess if a Stack Overflow code snippet is still flawed from one commit to another. We find that the static analysis-based approach is better in detecting vulnerabilities (with a precision of 100% on both classes, a recall of 65% for the negative class, and a recall of 57% for the positive class) compared to the machine learning approach especially when the codes under consideration come from real-world projects, rather than codes in synthetic datasets containing intentional flaws. Indeed, although we obtain good performances with machine learning, the model cannot discriminate classes. To investigate the propagation of Stack Overflow code snippets in GitHub, we use two approaches: a query on the table POSTREFERENCEGH of dataset SOTorrent and a clone detection tool. For vulnerable code snippets, we build the history and show that in a total of 18 repositories analyzed, 11 of these repositories have never removed vulnerable code from their projects.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

1.1 Context and Motivation

Stack Overflow is the world's most popular Q&A website for programming questions. Since its launch in 2008, Stack Overflow has accumulated millions of questions and answers related to programming. As of February 2023, Stack Overflow has over 20 million registered users and has received over 24 million questions and 35 million answers [10]. The site and similar programming question-and-answer sites have replaced programming books globally for day-to-day programming reference in the 2000s, and today are an essential part of computer programming [11]. Based on the type of tags assigned to questions, the site's top eight most discussed topics are JavaScript, Java, C#, PHP, Android, Python, jQuery, and HTML [12]. When answering questions on Stack Overflow, it is common for developers to attach code snippets within their answers as part of the solutions. As an example, Figure 1.1 illustrates the different elements of a post in Stack Overflow. The asker posts a question that consists of three parts: (i) title, (ii) body and (iii) tags. The title of a question is a short description of the essence of the question, whereas the body explains the question in more details. A tag (which can be considered a metadata) provides a glimpse into the topic of the question. The example question in Figure 1.1 has two tags, JavaScript and URL, which indicates that the question is related to JavaScript and URLs. Users provide different answers according to their knowledge and expertise (20 answers for our example question). Moreover, the asker can accept one answer among all answers, which is typically referred to as the best answer. Furthermore, other users can also up/down vote questions and answers. Such feedback mechanisms are built upon the important concepts of trust and reputation [13], which allow Stack Overflow to associate reputation scores and badges to users.[1] The extensive collection of code snippets within these answers becomes a code repository for solving

[1] `https://stackoverflow.com/help/whats-reputation`

Figure 1.1: An example of Stack Overflow post

programming problems among developers. Code with weaknesses (this term can be explained as a flaw in the software that could enable a hacker to access to the system. These defects have many reasons, one of which is how the software has been designed) can be risky to share or reuse among developers. Code sharing leads to code snippets propagating quickly across software systems. Prior studies observe that code snippets in various programming languages on Stack Overflow can be insecure [14], [15], [16]. For example, Meng et al. identified security vulnerabilities, e.g., bypassing certificate validation and using vulnerable cryptographic hash functions, in the suggested code snippets of accepted answers on Stack Overflow [14]. More specifically the authors found that:

- 5 out of 12 cross-site request forgery-relevant posts, developers took the suggestion to irresponsibly disable the default cross-site request forgery protection.[2]
- 9 out of 10 SSL/TLS-relevant posts discussed insecure code to bypass security checks.[3]
- 3 out of 6 hashing-relevant posts accepted vulnerable solutions as correct answers, indicating that developers were unaware of best secure programming practices.[4]
- Highly viewed posts may inadvertently promote insecure coding practices. This problem may be further aggravated by misleading indicators such as accepted answers, answers' positive votes, and responders' high reputation.

Rahman et al. observed that 7.1% of the answers to Python language questions contain violations of secure coding practices, e.g., code injection [15]. Fischer et al. observed that 15.4% of the 1.3 million Android applications contain security-related code snippets from Stack Overflow, and 97.9% of such code snippets contain at least one insecure code snippet [16]. Multiple studies have investigated knowledge flow and knowledge sharing from Stack Overflow answers to repositories of open-source software hosted in GitHub [17], [18]. They report that these code snippets found on Stack Overflow can be "toxic", i.e., of poor quality, and can potentially lead to license violations [18], [19]. Verdi et

[2] https://cwe.mitre.org/data/definitions/352.html
[3] https://cwe.mitre.org/data/definitions/757.html
[4] https://cwe.mitre.org/data/definitions/328.html

al. investigated security vulnerabilities in the C++ code snippets shared on Stack Overflow over a period of 10 years [20]. From the 72,483 reviewed code snippets used in at least one project hosted on GitHub, they found a total of 99 vulnerable code snippets. An important aspect that has not been investigated in detail is the evolution of insecure code snippets copied from Stack Overflow. A code snippet in a GitHub file, over several commits, has its life-cycle. After being introduced, it could be modified or even deleted. We ask if the changes made to these snippets removed vulnerabilities or added other flaws. If these vulnerable code snippets are migrated without ever changing them from Stack Overflow to applications, these applications will be prone to attacks. Stack Overflow has been aware of the negative impacts of insecure code infiltrations; unfortunately, there has been no principled way of dealing with insecure code snippets included in the posted questions/answer other than labeling the moderator flag, downvoting those threads, or warning in the comments. Given the rich structure and information of Stack Overflow with ever-evolving programming languages, there is an apparent and urgent need to do a quantitative study about the reuse of these vulnerable code snippets and the changes made to them.

1.2 Goal and Results

For the reasons mentioned above, in this thesis we conducted a quantitative investigation deepening three issues: (1) the detection of Stack Overflow C/C++ vulnerable code snippets, (2) their propagation, (3) their evolution. For the first issue, we conduct a comparative study between static analysis and machine learning approaches to understand which is the best method for detecting vulnerable code snippets. For the second issue, we explore two ways for finding re-usage of vulnerable Stack Overflow code snippets: the SOTorrent dataset and a clone detection tool. For the third issue, we analyze the evolution of files containing re-usage of vulnerable code snippets, because not all the re-usages found at the previous step are vulnerable. In summary, this thesis makes the following contributions:

- We compared the vulnerability detection approaches on a new dataset of real-world projects and conducted an agreement analysis to understand how often the static analysis and the machine learning approach: (i) predict both the right value, (ii) predict both the wrong value, (iii) predict different values. We observed that static analysis is better to detect insecure codes, especially when codes are part of real-world projects. For this reason, we chose the static analysis approach for detecting vulnerable code snippets from collected Stack Overflow posts.
- We explored two different approaches for detecting re-usages of code snippets in GitHub. Each of the approaches conducted showed advantages and disadvantages. For example, for the SOTorrent approach finding references is easy but information about the commit that introduces the code of interest is not immediately available in the table PostReferenceGH. For the clone detection approach we detect easily any type of clone with a similarity degree but codes with syntax errors have to be discarded, and there could be too long execution for finding clones

with the increasing of files to analyze. We obtain re-usages in two repositories for the SOTorrent approach, while we obtain re-usages in 55 repositories for C files and 80 repositories for C++ files with the use of the NICAD tool.

- We study the evolution of vulnerable C/C++ code snippets in GitHub, analyzing the two repositories with vulnerabilities in C files, and 16 repositories with vulnerabilities in C++ files. This investigation leads us to conclude that for 11 out of 18 repositories, despite continual changes, there has never been the removal of the vulnerability. This could confirm two possible theories: (i) developers are aware of vulnerabilities but they don't care much about the security aspect or (ii) developers copy and paste code snippets from Stack Overflow without really understanding them.

We conclude by describing possible insights on how our findings can be made actionable, such as suggesting the development of a browser plugin that reports a "security score" for each code snippet in Stack Overflow posts. The plugin can support all developers, with any kind of experience related to security, to be well-informed about the code that they are going to copy.

1.3 Thesis structure

This thesis is structured as follows:

- **Chapter 2 - Background**: provides some background information on vulnerabilities, their detection, and a brief overview on Stack Overflow and GitHub.
- **Chapter 3 - Related work**: provides relevant details on the research carried out on mining security-posts from Stack Overflow and the possible ways to detect vulnerabilities in them, the propagation of code snippets from Stack Overflow to GitHub, state-of-the-art studies to prevent the copy of insecure code snippets, and finally an overview of what we intend to do in this thesis.
- **Chapter 4 - Design**: provides details about the research questions and the methodology to respond to them.
- **Chapter 5 - Results**: provides a response to the research questions set out in the previous chapter.
- **Chapter 6 - Discussion**: provides a discussion on implication of this work.
- **Chapter 7 - Threats to Validity**: describes and illustrates the threats to the validity of the study.
- **Chapter 8 - Conclusions**: concludes the thesis by presenting a summary of the research done and the future directions.

CHAPTER 2

Background

In this chapter, we present the domain of our work, starting from the concept of vulnerabilities, and their detection. After we describe techniques to represent code. Lastly, we overview Stack Overflow and GitHub platforms, focusing on the security-related post and mining approaches.

2.1 Security Vulnerabilities

A vulnerability is formally defined as "a specific flaw or oversight in a piece of software that allows attackers to do something malicious: expose or alter sensitive information, disrupt or destroy a system, or take control of a computer system or program" [21]. How does one recognize code that contains a flaw relevant to security? One of the key issues in discovering vulnerabilities is the challenge of identifying features that correctly describe vulnerable code and distinguish it from clean code. Furthermore, methods can be divided into static analysis, which processes code without executing it, and dynamic analysis, in which code behavior is analyzed at run-time.

Typical vulnerabilities in source code are sometimes shared between languages and application domains, while others only occur in some. For example, buffer overflows are typical in the C family of languages, and cross-site request forgery is only of interest in the context of web applications. The following subsection will introduce and describe some typical vulnerabilities.

2.1.1 Injection attacks

An injection attack is based on user input that causes unintended or dangerous behavior when interpreted or executed. Exploiting an injection vulnerability usually allows the user to make the interpreter execute arbitrary commands, and sometimes to access or alter data without authorization.

Injection attacks can be prevented by checking all user input and applying so-called **sanitization** techniques that convert harmful into harmless inputs, for instance, by filtering out special characters.

Buffer overflow

Common Weakness Enumeration (CWE)[1] defines buffer overflow as follows: "The software writes data past the end, or before the beginning, of the intended buffer".[2] To understand how a buffer overflow occurs, the following code performs a simple password check (Listing 2.1).

```
#include <stdio.h>
#include <string.h>

int main(void){
    char buff[15];
    int pass = 0;
    printf("\n Enter the password : \n");
    gets(buff);
    if(strcmp(buff, "realpassword")){
    printf ("\n Wrong Password \n");
    } else {
    printf ("\n Correct Password \n");
    pass = 1;
    }
    if(pass){
    /* Now Give root or admin rights to user*/
    printf("\n Root privileges given to the user \n");
    }
    return 0;
}
```

Listing 2.1: Buffer overflow example

The attacker can supply input of length greater than what `buff` holds. This might lead to the memory being overwritten, and the portion of code containing the `pass` variable can be overwritten as well. If this happens, the `pass` variable will be different from 0, the if statement will be accessed and therefore the user will be given root privileges. A buffer overflow can be easily fixed by checking that the string given as input, respects the constraints.

SQL injection

According to **OWASP** (Open Web Application Security Project)[3] foundation, SQL injections are among the top security vulnerabilities, belonging to the most common and serious vulnerabilities that are affecting web applications. The Common Weakness Enumeration defines an SQL injection as follows: "The software constructs all or part of an SQL command using externally influenced input

[1] https://cwe.mitre.org
[2] https://cwe.mitre.org/data/definitions/787.html
[3] https://owasp.org/

from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component".[4] When user-controllable input contains SQL syntax that is not removed, it can be interpreted as an SQL statement instead of user data and executed. This can be exploited to alter queries, for instance, to access files that should not be accessible, or to add additional statements changing or destroying databases. Any kind of database-driven website is potentially in danger of becoming the target of such an exploit if its sanitization is not thorough. The following snippet illustrates such a vulnerability (Listing 2.2).

```
[...]
cur = db.cursor()
name = raw_input('Enter name: ')
cur.execute("SELECT * FROM users WHERE username = " + name + ";")
[...]
```

Listing 2.2: SQL injection example

If the user enters a legitimate request, like `Tom`, then all goes well, and the query has no negative effects. However, if the user inserts a string containing SQL code, like `Tom'; DROP TABLE users;` the semicolon is interpreted as the end of the query and everything afterward is executed as a new command, deleting the entire database table. If the code is changed like this (Listing 2.3):

```
[...]
cur = db.cursor()
name = raw_input('Enter name: ')
cur.execute("SELECT * FROM users WHERE username = %s;", (name,))
[...]
```

Listing 2.3: SQL injection solution

The parameter passed after the comma is escaped, not directly substituted, in the place of the placeholder `%s`. Any SQL code tokens are removed, and the request can be executed safely. This is of course only one option to fix the problem, but it illustrates the general need for filtering and sanitizing the user-provided input.

Command injection

Quoting again the Common Weakness Enumeration: "The software constructs all or part of a command using externally influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended command when it is sent to a downstream component".[5] This is another case where untrusted data is executed, but this time, it is not targeted at an SQL database, but instead, a command that is executed by the machine that is attacked, for example, the server shell. This can give an attacker the capability to read, change or

[4] https://cwe.mitre.org/data/definitions/89.html
[5] https://cwe.mitre.org/data/definitions/77.html

destroy files they should not be able to access. The following minimal example for command injection in Python demonstrates the vulnerability (Listing 2.4):

```
import subprocess
filename = input('Please provide the path for the file: ')
command = 'cat {path}'.format(path=filename)
subprocess.call(command, shell=True)
```

Listing 2.4: Command injection example

If a user enters `file.txt`, the file is displayed with the cat command, but by adding a semicolon, the user can append other commands which are then executed as well. For example, entering `file.txt; ls` would cause the shell to print the current directory, and `file.txt; rm -rf /` would cause a considerable amount of damage. This vulnerability could have been prevented by sanitizing or filtering the input.

2.1.2 Cross-site scripting

Cross-site scripting, often abbreviated as XSS, is one of the most important vulnerabilities in web applications. In cross-site scripting, unsanitized data is also the root of the problem. Custom code inserted by a user is added to a website or a URL that is then delivered to other users, who will receive the code as part of the website and execute it in their browser. CWE defines Cross-site Scripting as follows: "The software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users".[6] An example is an email with a link to another website that contains malicious Javascript within the URL, which will then be executed upon following the link. To prevent XSS attacks, user-generated content has to be sanitized, using functions such as `html_escape` and others. The example below shows a vulnerable snippet of Python code using the web framework Flask (Listing 2.5). The input `param` is taken from the user and inserted into the result page via the `html.replace` function.

```
@app.route('/ XSS_param',methods =['GET'])
def XSS1():
param = request.args.get('param','not set')
html = open('templates / XSS_param.html').read()
resp = make_response(html.replace('{{param}}',param ))
return resp
```

Listing 2.5: Cross-site scripting example

2.1.3 Cross-Site request forgery

CWE defines Cross-site request forgery as follows: "The web application does not, or cannot, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by

[6] https://cwe.mitre.org/data/definitions/79.html

the user who submitted the request" and further explains:[7] "When a web server is designed to receive a request from a client without any mechanism for verifying that it was intentionally sent, then it might be possible for an attacker to trick a client into making an unintentional request to the web server which will be treated as an authentic request. This can be done via a URL, image load, XMLHttpRequest, etc., and can result in the exposure of data or unintended code execution". To illustrate this with an example: A badly protected website might receive a POST request from a user which contains parameters to change the password. This request could have been generated by the user intentionally clicking on an HTML form that generates the request - or because a malicious person tricked the user into clicking a link with the same parameters that trigger the same POST request. The password is changed either way, and the user might even be locked out of his account while the attacker has access. This kind of attack can be prevented by using tokens that are exchanged between client and server that are secret and unique for each request. The following example shows how a token could be used to prevent XSRF attacks (Listing 2.6).

```
from oauth2client import xsrfutil
[...]
def CheckToken(self, *args, **kwargs):
user = users.get_current_user()
token = str(self.request.get('xsrf_token'))
if not user or not xsrfutil.validate_token(_GetSecretKey(), token, user.user_id()):
self.abort(403)
```

Listing 2.6: Cross-Site request forgery example

2.1.4 Directory traversal / path disclosure

A path traversal or directory traversal vulnerability occurs when the user can manipulate input in a way that paths of a file system are exposed that were not meant to be accessed. As defined by CWE: "The software uses external input to construct a path name that is intended to identify a file or directory that is located underneath a restricted parent directory, but the software does not properly neutralize special elements within the path name that can cause the path name to resolve to a location that is outside of the restricted directory".[8] A typical case of this vulnerability is a website that displays a file for which the path is specified in a URL parameter. By changing this parameter to contain some '../../..', the attacker can navigate the file system and possibly display files that were not meant to be accessed. Assuming that `filepath` is a variable provided by the user, the following example shows a vulnerable code segment (Listing 2.7).

```
import os.path
[...]
prefix = '/home/user/files/'
full_path = os.path.join(prefix, filepath)
```

[7] https://cwe.mitre.org/data/definitions/352.html
[8] https://cwe.mitre.org/data/definitions/22.html

```
read(full_path, 'rb')
```

Listing 2.7: Directory traversal/path disclosure example

A possible way to fix this would be to use methods to check whether the prefixes are still the same.

2.1.5 Open redirect vulnerabilities

As the name suggests, open redirect vulnerabilities occur when a website that is supposed to redirect a user gets the target of the redirect from a parameter value. When this URL parameter is manipulated, the redirect sends the user to the wrong page, for instance, a malicious page crafted by an attacker. The Python web framework Django contains a check called `is_safe_url` which can be used to perform a check to prevent open redirect vulnerabilities as shown in the following examples (Listings 2.8 and 2.9).

```
next_url = request.GET["next"]
return next_url
```

Listing 2.8: Open redirect vulnerabilities example

```
from django.utils.http import is_safe_url
[...]
if is_safe_url(request.GET["next"]):
next_url = request.GET["next"]
return next_url
```

Listing 2.9: Open redirect vulnerabilities solution

2.2 Detecting Security Vulnerabilities

2.2.1 Static analysis

In static code analysis, the form, structure, content, or documentation of a program is analyzed without executing it, using generalization and abstract rules [22]. Therefore, static approaches can name the precise cause of a vulnerability that was detected. Static code analysis tools are frequently used to identify potential issues, and a wide range of commercial and open-source tools are available. Without any doubt, those tools have a strong positive effect on code quality, and they allow developers to fix mistakes even before the program is run for the first time. The success of static analysis tools depends fundamentally on the quality of the underlying patterns, abstractions, or rules that are used to identify problems, and the more complex software becomes, and the more creative potential attackers are, the harder it is to define patterns of secure software. Although new rules can always be added to a static tool, if one hasn't written about a particular vulnerability yet, the tool will be blind to this problem and never recognize it. To create those patterns automatically appears to be nearly impossible, and creating them manually is tedious and time-consuming, especially since

technology and computer systems advance quickly, and therefore keeping track of all possible kinds of vulnerabilities is hard. The accuracy of static analysis tools is measured by calculating the rate of false positives (reported problems that are not an issue) and false negatives (missed problems). Especially the high false positive rate in many tools is a big obstacle, as those misclassifications cause developers to spend a large amount of time checking each one, which makes it harder to spot the actual problems.

2.2.2 Dynamic analysis

Another option is dynamic program analysis, in which programs are executed and monitored at run-time, and execution traces are systematically analyzed. Of course, not all possible inputs and execution paths can always be covered, so dynamic analysis can very well miss a lot of problems. A big practical problem is that dynamic analysis requires much more computational time and resources and is simply not possible in many circumstances, especially when analyzing larger software. Dynamic approaches are limited in their applicability by the large number of test cases that need to be run.

2.2.3 Machine Learning

As it has been pointed out, traditional approaches rely on features defined by human experts to detect vulnerable code, however, it is desirable to reduce the human workload and find more general, scalable, and objective methods. This is where Machine Learning comes into play. Machine Learning describes computational algorithms that allow computer systems to solve problems without explicit programming. It allows them to learn from experience and form concepts from examples by extracting patterns from the raw data. Approaches are distinguished as supervised learning techniques, which use labeled data to allow a learning system to gain insight and create a model, unsupervised learning, in which the system finds patterns and structures in the dataset without guidance, and reinforcement learning, in which the system is provided with rewards and penalties after each step to train it dynamically. Those approaches can analyze large repositories without requiring the source to be compiled, similarly to static analysis. However, they do not suffer from the false positive problem as much, as they allow for a fine-tuning of precision and recall, giving them an at least promising advantage over both static and dynamic approaches.

Neural Networks

A neural network is a system that consists of many simple, interconnected parts, called neurons, in analogy to the neurons in the human brain. The neurons are organized in layers. Each neuron takes in several inputs and puts out a single number as a function of these inputs, which is referred to as the neuron's activation. The first layer of neurons is called the input layer, the following layers are hidden layers, and the last is called the output layer. A neural network can also consist of just one single layer. In Figure 2.1, the network has six layers, including four hidden ones. The neurons of each layer

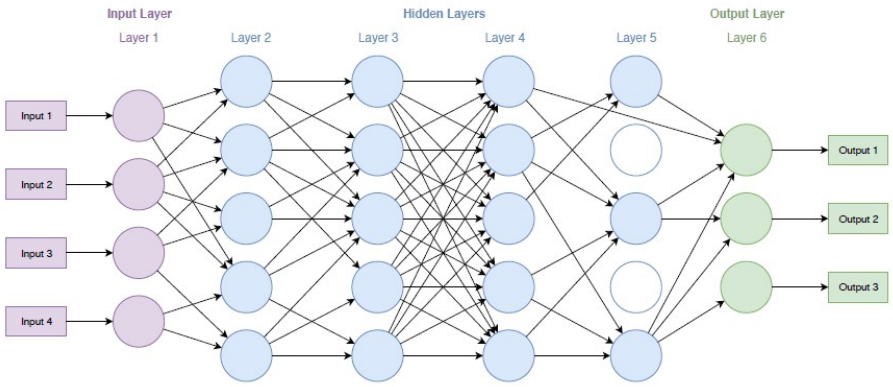


Figure 2.1: Example illustration of a neural network

are connected to neurons from the precedent and subsequent layer, and in the case of input neurons, also with input variables from the neural network's environment, or if they are output neurons, with output variables. Following their own set of rules, the neurons react to the other neuron's activation patterns by activating or de-activating themselves, and consequently cause changes in the next layer of neurons, until the activation pattern of the final layer of neurons provides the output of the neural network. The training of a neural network consists to find the right internal rules that make the network as a whole exhibit a desired behavior. Since the internal rules are described in terms of mathematical functions, this comes down to discovering the correct values for the parameters of those functions (weights). The number of neurons can vary between layers. If the output is only a single value, as is the case in some classification problems, the last layer might only have one neuron. A fully connected or 'dense' layer is a layer in which every neuron of the layer before is connected to every neuron of the dense layer. In the illustration, layer 4 is dense, all others are not dense, also called 'sparse'. A dropout layer is a layer in which some randomly chosen neurons are simply ignored by setting their output to zero so that they can't affect the next layer. In Figure 2.1, this is shown in layer 5. The percentage of neurons to ignore is set by a dropout parameter. This concept helps to prevent overfitting, which might occur if the network relies too heavily on some nodes. By randomly 'turning off' a share of nodes, the cost function has to take the neighboring nodes more into account.

Ensemble Learning

In machine learning, the bias is the deviation between the model's prediction and the actual value, whereas the variance is the error caused by the model's sensitivity to minute variations in the training set. An algorithm with high bias might underfit the target outputs and features (fail to recognize these important relationships), whereas an algorithm with high variance might do so by modeling the random noise in the training data (overfitting). The property of a model known as the bias-variance trade-off states that it is possible to reduce the bias in the estimated parameters at the expense of increasing the variance of the parameter estimated across samples. The use of mixture models and **ensemble learning** is one strategy for overcoming the bias-variance trade-off. Voting,

stacking, bagging, and boosting are typically the four main ensemble methods.

- **Voting**: voting ensemble method combines the output of various models' predictions to produce a final prediction. Since this ensemble method trains the models using all of the training data, the models should be distinct. The output for regression tasks is the average of the models' forecasts. Instead, there are two methods (hard voting and soft voting) for estimating the final output for classification tasks. The former involves taking the mode of all the model's predictions (Figure 2.2). The latter employs the highest probability, after averaging the probabilities for all the models (Figure 2.3). The main idea behind voting is to be able to generalize better by compensating for the errors of each model separately.

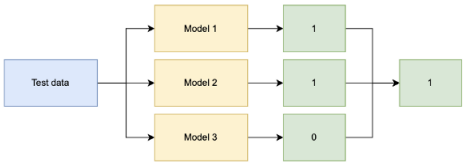


Figure 2.2: Hard voting

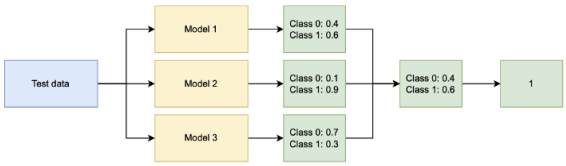
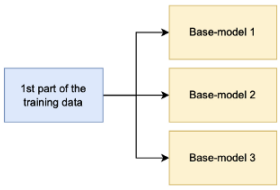
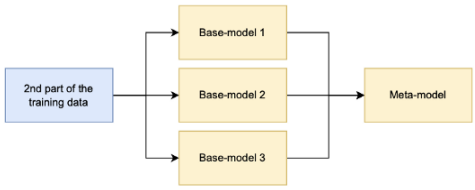


Figure 2.3: Soft voting

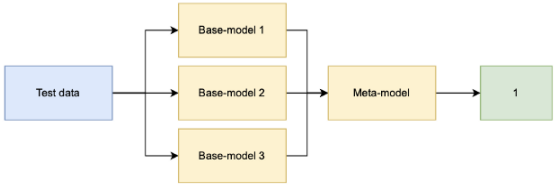
- **Stacking**: the weighted voting of the contributing models prevents all models from contributing equally to the prediction in the stacking ensemble models, which are an extension of the voting ensemble models. Base models (models fitted to the training data) and a meta-model make up the stacking models' architecture (a model that learns how to combine the predictions of the base models). It is customary to use a logistic regression model for classification tasks and a linear regression model for regression tasks. The base models' extrapolations from non-sample data are used to train the meta-model. The task determines the input of the meta-model. The predicted value is the input for regression tasks. The predicted value for the positive class is typically the input for binary classification tasks (Figures 2.4a, 2.4b, 2.4c).



(a) Training base-models of the stacking ensemble method



(b) Training meta-models of the stacking ensemble method



(c) New data on stacking ensemble method

- **Bagging**: is the process of sub-sampling training data to enhance a specific kind of classifier's generalization performance. On models that have a propensity to overfit the dataset, this

technique works well. Bootstrapping is the process used to sub-sample the data (the term bagging is derived from bootstrap + aggregating). This method involves performing random sampling with replacement over the data, which means that the training data subsets will overlap because the data is not being split but rather re-sampled. Once the results of all the models have been obtained, the final prediction for each piece of data can be made by either voting in a classification or regression model, where predictions are determined by the majority vote of contributing models.

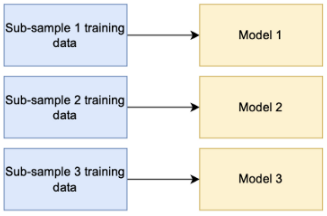


Figure 2.5: Training models for bagging ensemble method

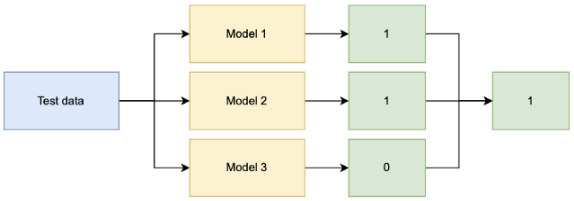


Figure 2.6: New data on bagging ensemble method

- **Boosting**: a technique that employs weak learners sequentially to build the model. The main principle is that each succeeding model corrects the flaws of the one before it. Although there are many different kinds of boosting algorithms, including Gradient Boosting and XGBoosting, AdaBoost was the first one created for binary classification. The process of AdaBoosting is shown in Figure 2.7 where the task is to categorize the circles and squares based on the x and y features. The first model creates a vertical separator line to categorize the data points. However, as seen, some of the data points in the circles are incorrectly classified. Therefore, the second model increases the weight of the incorrectly classified data points to classify these misclassified data points. This procedure is carried out repeatedly for the specified number of estimators.

2.3 Representing code

When working with machine learning techniques on source code, many different paths can be taken to preprocess the code and represent it in a suitable format.

- **Abstract syntax trees**: Abstract syntax trees (ASTs) provide a hierarchical representation of the code and can be used in different styles, omitting more or fewer tokens to simplify the tree.
- **Code as natural text**: Code can also be viewed just as a sequence of words or tokens, with the added characteristic of resembling natural language. Source code shares some characteristics with natural language text: repetitiveness of certain structures and common patterns, localness (repetitions occur in a local context), and long-term dependencies (`try` requiring a `catch` later in Java). Following this line of reasoning, machine learning models that are successfully used in natural language modeling can be applied to code as well, since the core strength of machine

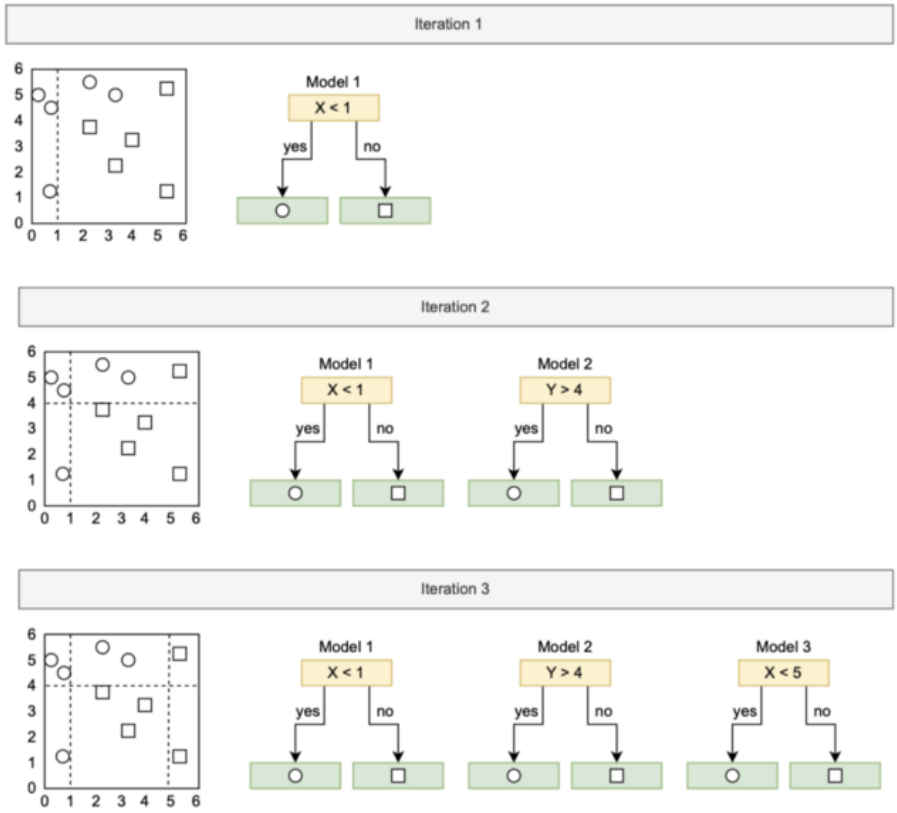


Figure 2.7: The training methodology for the boosting ensemble method

learning lies precisely in the ability to uncover and generalize patterns and handle noise. Code, however, is different from natural language in so far that it is 'semantically brittle' because small code changes can already alter the functionality completely. It also contains highly complex structural information, for example, nested loops.

- **Bag of words**: A very simple representation that just takes code as text is the bag of words model. A data sample is represented by a collection of the singular tokens and the number of times they appeared. The order of tokens is not taken into account. To give an example for code: The code snippet `if (a > b or b > c):` would be represented with `{'a': 1, 'b':2, 'c': 1, 'if': 1, 'or': 1, '>': 2, ':':1,'(': 1, ')': 1}`.

- **N-gram models**: The last n tokens are taken into account and collected as an element. For example, for n = 2, the code snippet from above would be transformed into the representation `{'(a', 'a >', '> b', 'b or', 'or b', ...}`, always looking at the last two tokens. For bigger n, this approach is able to capture some of the semantic context that lies in the order of tokens. However, these methods suffer from performance problems when dealing with high dimensionality.

If a neural network is to be trained on the data, the samples also have to be transformed into a numerical representation. For ASTs, there exist mainly handcrafted methods to transform the tree

structure into a numerical vector.

- **One-hot embedding**: A vector of dimensionality n is used to represent each token. The value n is at the same time the total number of unique tokens that can be encoded. Each token is encoded by a vector consisting only of zeros and one singular '1' at the position that refers to this token. As a result, those vectors are extremely high in dimensionality and sparse. For example, in the code snippet used as an example before, a vector with 9 dimensions would be necessary, and 'a' could be encoded as {1,0,0,0,0,0,0,0,0}, 'b' as {0,1,0,0,0,0,0,0,0}, and so on.

- **Word2Vec embedding**: it also yields a unique vector representation for each unique token, however, it takes semantics into account by giving similar tokens similar representations. Word2vec uses the ways tokens appear in combination and relation to each other to assign vectors with high cosine similarity to semantically similar tokens. For code tokens, it would be expected that, for instance, the vectors for true and false have higher similarity than the vectors for return and while. The encoding uses cells to represent certain semantic features. Figure 2.8 shows a simplified illustration of a word2vec model. Python code tokens like true or count are transformed in a vector representation of dimensionality eleven, represented by the eleven colored cells for each token.

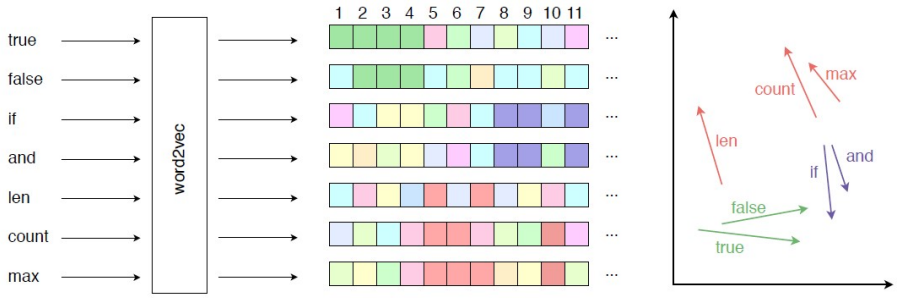


Figure 2.8: word2vec embedding

The different numerical values that make up a vector are represented with colors to highlight the differences and similarities between the vectors visually. In this example, the vector representations of `true` and `false` share similarities in the first to the fourth cell. Furthermore, one might, for example, guess that the 8th, 9th, and 11th cells might encode some of the semantic of logical operations, as they have similar values in the token representation of `if` and `and`. However, it is not possible to define a specific, unequivocal meaning for each cell. On the right of the figure, a simplified two-dimensional vector space is shown, containing the encoding for the words which are just symbolically represented as two-dimensional vectors. Tokens with related semantics have high cosine similarity or a very small angle between them.

2.4 Stack Overflow

Stack Overflow (SO) is an online collaborative platform for developers to post their programming questions, provide answers to existing questions, and find solutions to their difficulties faced during programming. A developer needs to add tags while posting a question to help other users to find out what the question is about. If the answer provided by any user gives a solution to the problem faced by the questioner, that answer can be selected by the questioner which is called the accepted answer to that question. Different members of the site can vote on questions and answers. The positive votes are called the *upvote* and the negative votes are called the *downvote*, which shows how helpful that question/answer was for other users. The score of a question/answer is determined by the difference between the number of up/down votes. Based on the different activities of each user on SO such as posting questions or answers, voting on them, posting comments, their reputation score increases which help them build their reputation on the SO website. The greater the reputation values, the more the capabilities for a member on SO like deletion of questions/answers, closing questions, etc. Stack Overflow encourages the community to revise the content of answers, including both textual description and code snippets, to maintain the quality of such answers. Any code snippet in answers can be revised, and new code snippets can be introduced to the answer at the revision phase. Stack Overflow contains millions of posts that cover a wide range of topics, such as general programming languages, web development, mobile development, and security-related topics. Although it seems logical and natural that the security-related posts are generally tagged with "security", there are exceptions to this general rule (Figure 2.9 and Figure 2.10). Therefore we cannot determine security-related posts by simply checking whether the posts contain the tag of "security", since the extracted posts will not be sufficient and satisfactory.

Figure 2.9: A security-related post on Stack Overflow tagged with 'security'

As the popularity of Stack Overflow grows, the incentive of launching a large-scale security attack by exploiting the vulnerability of posted code snippets increases as well. Those innocent-looking yet insecure code snippets - if not properly handled and directly transplanted to production software - could cause severe damage or even a disaster (e.g., disrupting system operations, or leaking sensitive information). For example, as shown in Figure 2.11, since cryptocurrency has grown popular, attackers have injected malicious mining code such as *Coinhive* - a cryptocurrency mining service - into Stack

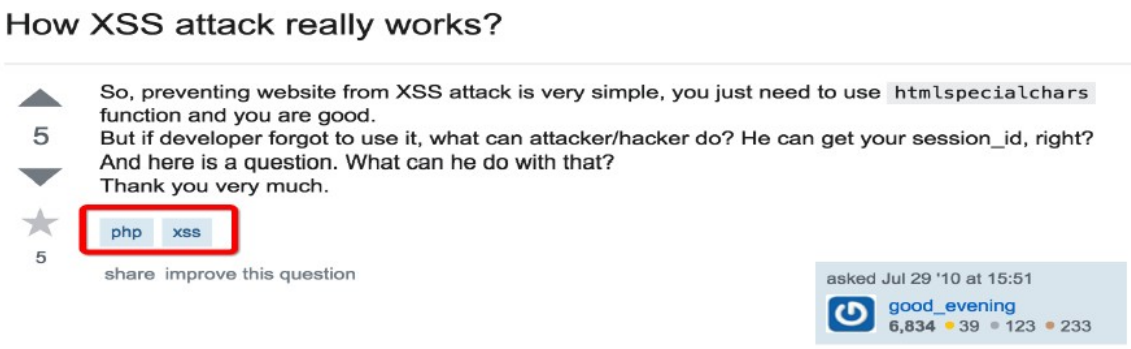


Figure 2.10: A security related post without a 'security' tag

Overflow; once innocent developers reuse or copy-paste such code snippets to generate the production software, the software users' devices could be compromised. Given the rich structure and information of Stack Overflow with ever-evolving programming languages, there is an apparent and imminent need to develop novel and sound solutions to address the issue of code snippet security in Stack Overflow.

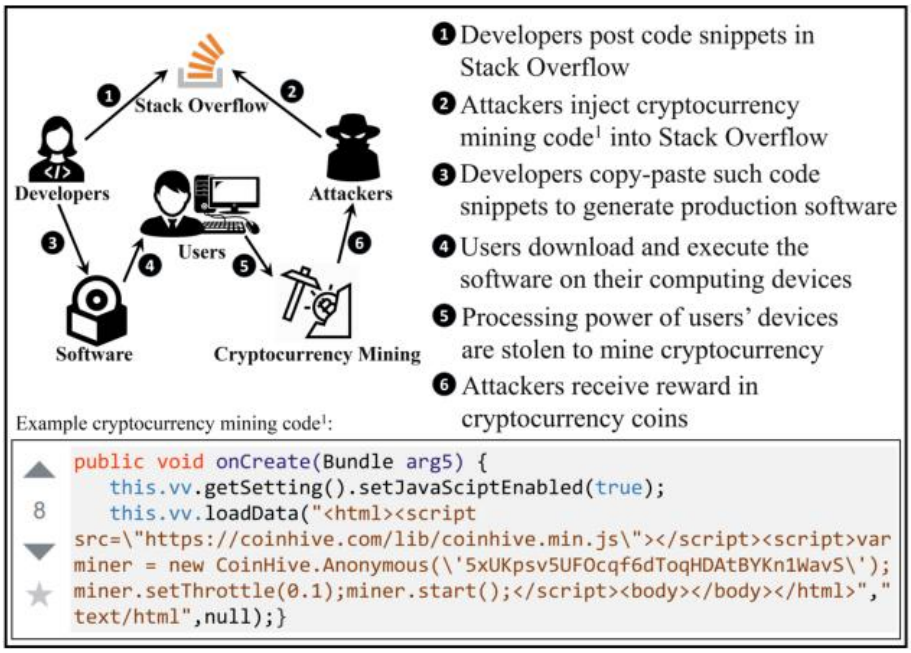


Figure 2.11: Example of code security attacks in Stack Overflow

Stack Overflow provides data dumps of all user-generated data, including questions asked with the list of answers, the accepted answer per question, up/down votes, favorite counts, post score, comments, and anonymized user reputation [23]. The data dump consists of eight database tables:

1. **Badges**: Contains a list of badges earned by the Stack Overflow users, for being especially helpful.[9]

2. **Comments**: Contains the list of comments made to each post.

[9]Besides gaining reputation with questions and answers, users receive badges for being especially helpful.

3. **PostHistory**: Contains all revisions made to each post.
4. **PostLink**: Contains all linked and duplicate links related to a post.
5. **Posts**: Contains all questions and answers posted on Stack Overflow. Each question or answer is stored as a separate post with a unique id, and other attributes such as user id, time, and text associated with it.
6. **Tags**: Contains all existent tags on Stack Overflow. Stack Overflow allows users to tag each question, with up to a maximum of 5 tags. Users can select an existing tag provided in the autocomplete text box or create a new one. To create a new tag, users need to have a minimum level of reputation on Stack Overflow.[10] This makes sure that new tags are only created by expert users, maintaining consistency among tags found on Stack Overflow. Expert users can also change the question tags, if the questions are incorrectly tagged.
7. **Users**: Contains an anonymized list of Stack Overflow users.
8. **Votes**: Contains the count for all the up-votes, down-votes, favorites, etc for each post.

2.5 GitHub

Before introducing GitHub, a quick overview of Git is done.

Git is a Version Control System (VCS) program that helps the tracking and merging changes in files of a project (repository in Git terms) and is especially popular for use in software projects. What changed, who changed it, and for what reason it was changed is tracked across versions, formally called revisions, essentially creating a "timeline" of the project development. When coordinating software development, particularly when work is made on the same file, Git can aid developers in solving conflicts about what should be added and removed. There are a few important concepts that encapsulate the general usage of Git as a VCS well.

- **Commit**: A snapshot of the code-base at a certain point in time, and is the most simple way to create a revision in Git. Each commit has an associated message written by the developer describing the revision. After a developer has written some code that should be part of the final software they would create a commit of that code. Making commits is therefore the major device for code contribution in a project. If a developer introduces a breaking change to the code base, one can revert to a previous commit where the code was functioning.
- **Branch**: Branching is utilized when one wants to work on multiple parts of software in parallel without them affecting each other. To accomplish this a branch can be created, which can be seen as a parallel "timeline" to the other branches. In this new branch, multiple commits can be made

[10] Approximate measurement of how much the community trusts a user; it is earned when the peers appreciate what a user is contributing.

which may at some point be merged into the original branch. The original branch is commonly referred to as the master or main branch, and typically contains the most stable version of the software.

- **Remote and local repositories**: The remote repository is the original project which may be hosted online. The local repository is a local snapshot of the remote repository which is being worked on and is usually continuously synchronized with the remote repository. The remote repository is used as a "source of truth", where changes that should be incorporated into the main project are uploaded (pushed in Git terms). A remote repository can be downloaded locally by issuing the clone command along with the remote URL. In doing this process all files in the repository are fetched along with the entire Git history of the project.

GitHub is the largest platform for hosting and collaborating on software projects with over 53 million developers [24]. Remote repositories of projects that exclusively are using the VCS Git can host their projects on GitHub. There are a few important GitHub-specific features:

- **Fork**: By forking a project on GitHub, users can create a complete copy of the project, which can be worked on alongside the main project. Forks are one of many ways for collaborators to separately work on changes in a project. When the changes are ready, users can ask the original project maintainers to review and possibly merge in the changes from the fork. The process of opening a request for a review and merge is called a pull request.
- **Issue**: By opening an issue on a GitHub project, one can describe bugs, ask for desired features or discuss the architecture or future of a project. Generally, as the name implies, issues are used for describing some issue with the software that the project pertains to, in order to help the developers identify the cause and fix it.
- **Star**: By starring a project on GitHub, users can indicate that they like a project and save the project to their account under the starred projects section. How many stars a project has is public and is generally an indication of a project's popularity.

2.5.1 SOTorrent

SOTorrent is an open dataset based on data from the official Stack Overflow data dump and the Google BigQuery GitHub (GH) dataset [1]. SOTorrent provides access to the version history of SO content at the level of individual post blocks. A post block can either be a text or a code block, depending on how the author formatted the content. Beside providing access to the version history, the dataset links SO posts to external resources in two ways:

1. By extracting linked URLs from text blocks of SO posts.
2. By providing a table with links to SO posts found in the source code of GitHub projects. This table can be used to connect SOTorrent and GH dataset.

SOTorrent contains all tables from the official Stack Overflow data dump (database schema in Figure 2.12). However, that dump does only provide the version history at the level of whole posts as Markdown-formatted text.

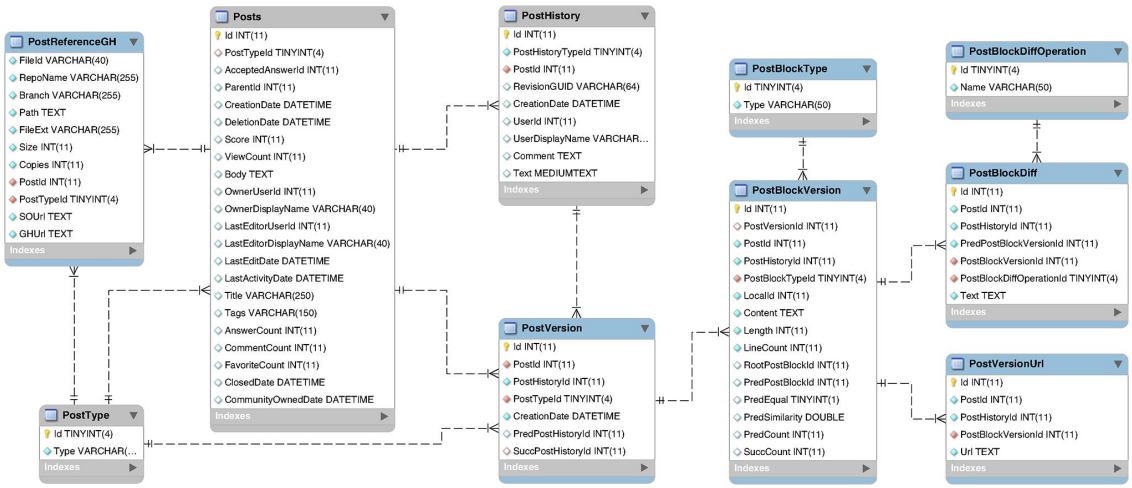


Figure 2.12: Database schema of SOTorrent: The tables from the official SO dump are marked gray, the additional tables are marked blue. Not all tables from the official SO dump and not all foreign key constraints are shown. Figure is taken from [1]

In the SO dump, one version of a post corresponds to one row in the table `PostHistory`. However, that table does not only document changes to the content of a post, but also changes to metadata such as tags or title. To analyze how individual text or code blocks evolve, there is the need to extract individual blocks from that content. First, Baltes et al. selected edits that changed the content of a SO post and linked each version to its predecessor and successor and stored it in table `PostVersion` [1]. The content of a post version is available as Markdown-formatted text. They split the content of each version into text and code blocks and extracted the URLs from all text blocks using a regular expression (table `PostVersionUrl`). To reconstruct the version history of individual post blocks (table `PostBlockVersion`), they established a linear predecessor relationship between the post block versions using a string similarity metric. For each post block version, they computed the line-based difference to its predecessor, which is available in table `PostBlockDiff`. One row in table `PostReferenceGH` represents one link from a file in a public GH repository to a post on SO. To extract those references, they utilized Google BigQuery, which allows to execute SQL queries on various public datasets, including a dataset with all files in the default branch of GitHub projects [25]. To find references to SO, they applied the following regular expression to each line of each non-binary file in the dataset: `(?i: https ?:// stackoverflow\.com /[^ \s) \.\ "]*)`. They stored that link together with information about the file and the repository in which the link was found in table `PostReferenceGH`.

2.6 Code Clones

Code clone can be defined as reusing a code fragment by copying it from one section of the software and then pasting it with or without minor modifications or adaptations into another section of the software.

Four types of clones can be identified based on the notion of similarity considered between code fragments. These clones based on their similarities are categorized as textual (Type I, II, III) and functional (Type IV). In detail, these types are:

- **Type I**: Exact clones in which the same fragment of code is copied without modification in its semantic or syntactic structure (except for spacing and comments).
- **Type II**: Clones that are syntactically identical fragments except for slight variations, such as different identifiers names, literals, types, or spacing.
- **Type III**: Clones that have been slightly modified by adding, removing, or reordering statements, in addition to Type I and Type II modifications.
- **Type IV**: Functional clones refer to code fragments that perform similar operations, but their syntactic and semantic structures are different.

Detecting different types of clones requires different levels of sophistication. While Type I and Type II can be relatively easy to detect using lexical-based analysis, other types (Type III and Type IV) require a higher level of complexity to match operationally identical code fragments.

2.6.1 Code Clone Detection

In general, the methods for detecting different types of clones in software systems can be classified into several categories as listed below:

- **Textual Analysis**: Text-based code clone detection techniques are the traditional and easiest way of detecting clones. Such methods analyze the lexical structure of source code, looking for similar textual patterns that might indicate cloning. In particular, the string-based detection techniques are often language-independent and are used for detecting either identical sections of code (Type I) or clones with slight modifications, such as renamed variables (Type II). In this technique, strings are compared line by line without any code transformation. However, lately, some text-based techniques involve transforming code by removing comments and whitespaces from the code. Since these techniques do not require an examination of code in terms of semantical or syntactical, the performance is fastest in the case of these techniques compared to other techniques. Dup is an example of a tool that uses textual analysis for clone detection [26].
- **Token-Based Technique**: In token-based clone detection techniques, at first, the code is transformed into different tokens and then compared against matching tokens in the series. In general,

a lexical analyzer is used to transform the code into tokens. As each code is tokenized, this technique performs slower as compared to the text-based technique, as much amount of time is consumed in tokenization. CCFinder is an example of a tool that uses token analysis for clone detection [27].

- **Code Metrics Analysis**: Such tools identify duplicates and semi-duplicates in source code by comparing pieces of code based on various metrics extracted from source code. The underlying assumption is that if two or more code fragments are clones, then they share several characteristics that these metrics can effectively capture. As compared to other clone detection techniques, code metrics methods use less time to detect code, are less complex, and are easy to use. Covet is a tool that follows the metrics-based clone detection technique [28].

- **Parsing Techniques**: Such techniques work by matching parts of the abstract syntax trees (AST) of source code. In particular, similar subtrees in the system's AST indicate cloning. In addition to detecting exact clones, such techniques can also identify Type II clones, given that variables' names and literal values are ignored when constructing the system's AST. When this technique is applied to a huge source code, it takes a lot of time. CloneDR is a tool that uses the AST-based approach for clone detection [29].

- **Graph Analysis**: Under this approach, similar code fragments in a program are identified by matching similar subgraphs in a program dependency graph (PDG). A PDG represents the structure of a program and its data flow. Such technique enhances the AST parsing technique by considering not only the syntactic structure of programs but also the data flow. This particular approach can capture clones in which matching code statements have been reordered. Duplix is a tool that applies PDG-based clone detection [30].

- **Hybrid Technique**: This technique can be diverse in its operations. Here, programmers typically bring two or more of the aforementioned techniques together to detect clone codes. The processes can be sophisticated, such that they are done in stages, an entire technique would be the first stage, and then another would be the following step. Koschke et al. compared the tokens of Abstract Syntax Tree (AST) nodes by making a direct comparison between each AST nodes [31]. Tairas et al. developed a method that could detect existing, functional clones in software; it combines suffix trees and AST-based techniques [32].

CHAPTER 3

Related Work

In this chapter, we first analyze papers that search for security-related posts on Stack Overflow using a keyword-based approach. Then, we analyze not only reports that studied the propagation of code snippets from Stack Overflow to GitHub but also reports that provided security recommendations to prevent the code from being copied. At the end of the chapter, we summarize the approach taken in this work.

3.1 Mining security-posts on Stack Overflow

3.1.1 Keyword-matching

Through various **APIs**, Fischer et al. analyzed Stack Overflow posts about Android security [16]. If a piece of code inside a Stack Overflow post calls one of the following APIs, it is deemed security-related:

- Java Cryptography Extension (JCE) and Java Cryptography Architecture (JCA).
- Java Secure Socket Extension (JSSE), Java Generic Security Service (JGSS), and Simple Authentication and Security Layer (SASL) for secure network communications.
- Java certification path API, PKCS#11, OCSP, X.509, and Certificate Revocation Lists (CRL) in java.security.cert.
- Java Authentication and Authorization Service for authentication and access control (JAAS).

Fully Qualified Names (FQN) are required of the code elements in a code snippet to determine which API is utilized by the snippet. Different APIs may contain classes (such as `android.util`

`.Base64`, `java.util.Base64`) and methods (such as `java.security.Cipher.getInstance` and `java. security.Signature.getInstance`) with the same name because Partially Qualified Names (PQN) are not unique. The distinction between non-unique class and method names can be done using FQNs. Only PQNs are available because code snippets posted on Stack Overflow are frequently incomplete. An oracle called JavaBaker is used to determine which API a code element belongs to.[1] If a piece of code is provided, JavaBaker will return the FQN for each element if it comes from one of the libraries that were initially specified. Therefore, if the oracle's returned result is not null, the code fragment is deemed security-related. Anyway, this approach based on searching snippets that used these APIs might have a high rate of false positives, because the **oracle might label a snippet as security-related, even though it does not belong to a security context**. After security-related code snippets are extracted, there is the need to classify them as such. Supervised learning is applied that needs the **manual labeling** of a small fraction of extracted code snippets to train the model. Therefore, a pair of two reviewers inspected the set of 1,360 security-related snippets extracted from answer posts from Stack Overflow. However, with this human approach, incorrect labeling is possible.

Chen et al. labeled insecure code based on the **Java security rules** summarized by prior work described above [6]. In detail, the authors defined five categories of security issues relevant to library improper use. Table 3.1 shows their criteria to decide whether a code snippet is insecure or not.

Although they tried their best to accurately label code, their analysis may be still subject to **human bias** and cannot scale to handle all crawled data or more security categories. They conservatively assumed a snippet to be secure if it does not match any given rule. However, it is possible that some labeled secure snippets match the insecurity criteria not covered by this study, or will turn out to be insecure when future attack technologies are created. An automatic approach that detects all the fragments that have vulnerabilities could be useful, rather than appealing to the choice of APIs and the consequent manual labeling to understand if that fragment contains vulnerabilities or not.

Meng et al. obtained 22,195 security-related posts considering keywords `Java` and `security` inside the content of questions and answers [14]. After extracting the question, answers, and relevant metadata for each post, they refined the data in three ways.

1. *Filtering less useful posts*: they refined posts by removing duplicated posts, posts without accepted answers, and posts whose questions received negative votes (usually because the questions were ill-formed or confusing).

2. *Removing posts without code snippets*: To better understand the questions within the program context, they only focused on posts containing code snippets. Since the crawled data did not include any metadata describing the existence of code snippets, they developed an intuitive filter to search for keywords `public` and `class` in each post. Based on their observation, a post usually contains these two keywords when it includes a code snippet. This approach is inaccurate because the filter may **incorrectly remove some relevant posts** that contain code.

[1] `https://github.com/siddhukrs/java-baker`

Category	Parameter	Insecure
SSL/TLS	HostnameVerifier	allow all hosts
	Trust Manager	trust all
	Version	<TLSv1.1
	Cipher Suite	RC4, 3DES, AES-CBC MD5, MD2
	OnReceivedSSLError	proceed
Symmetric	Cipher/Mode	RC2, RC4, DES, 3DES, AES/ECB, Blowfish
	Key	static, bad derivation
	Initialization Vector (IV)	zeroed, static, bad derivation
	Password Based Encryption (PBE)	<1k iterations, <64-bit salt, static salt
Asymmetric	Key	RSA <2,048 bit, ECC <224 bit
Hash	PBKDF	<SHA224, MD2, MD5
	Digital Signature	SHA1, MD2, MD5
	Credentials	SHA1, MD2, MD5
Random	Type	Random
	Seeding	setSeed→nextBytes, setSeed with static values

Table 3.1: Criteria used to decide code's security property, table is taken from [6]

For example, the answer in Figure 3.1 is one of many security-related posts on Stack Overflow without the keywords `public` and `class`. We can conclude that the choice of these keywords is erroneous because leads to an incomplete collection of posts, so the list of keywords used might be wider.

3. *Discarding irrelevant posts*: After applying the above two filters, they manually examined the remaining posts, and decided whether they were relevant to Java secure coding, or simply contained the checked keywords accidentally.

Zhang et al. empirically studied the prevalence of the CWE in code snippets of C/C++ related answers [33]. They explored the characteristics of $Code_w$, i.e., code snippets that have CWE instances, in terms of the types of weaknesses and the evolution of $Code_w$. They found that:

1. CWE-119, i.e., improper restriction of operations within the bounds of a memory buffer, is common in answer code snippets.

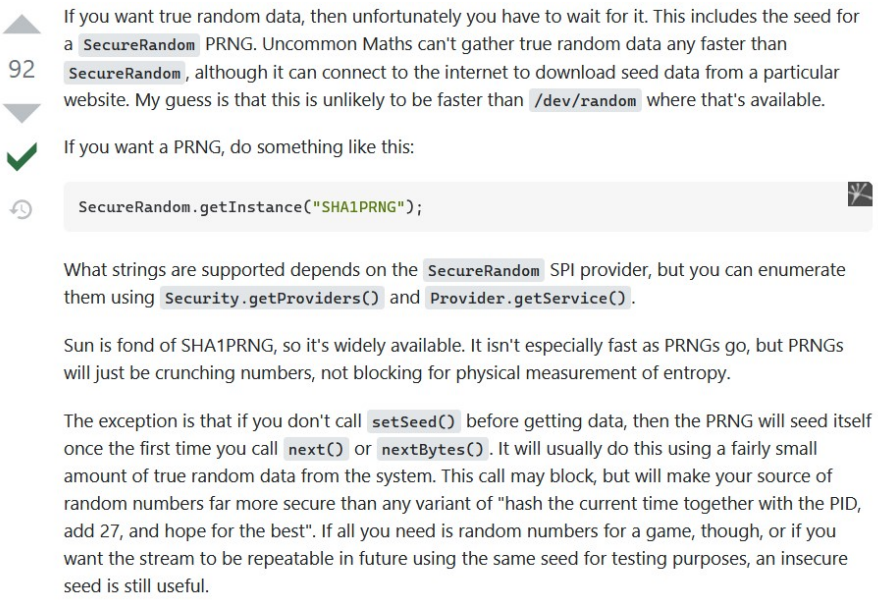


Figure 3.1: Example security-related post containing code without the public and class keyword

2. In general, code revisions are associated with a reduction in the number of code weaknesses.

They detected code snippets with weaknesses using Cppcheck.[2] Cppcheck can identify 59 out of the 89 types of C/C++ code weaknesses. When they scanned code snippets with Cppcheck, they skipped any code snippet that returns **syntax errors**. This approach may not capture all types of security weaknesses in C/C++ code snippets on Stack Overflow, because can happen that there could be a snippet potentially vulnerable in which there are trivial errors like missing a semicolon. An approach that detects vulnerability code snippets by bypassing trivial syntax errors could be useful because allows the inclusion of many more code snippets that could be helpful for analysis. Even if this study has focused on CWE in code snippets C/C++ answers, it can not be excluded that code snippets from **questions can also have CWE instances**. To mitigate the bias from pseudo-code or command line functions, they **removed code snippets with less than five lines** of code.

Rahman et al. focused on Python code snippets [15]. To identify insecure coding practices in code blocks, they used a catalog of insecure coding practices reported by Openstack.[3] The list reported by Openstack included 77 coding patterns of Python that have security weaknesses. However Openstack is not comprehensive, and many code snippets might not be analyzed. The use of **string matching** is wrong to detect the presence of insecure codes because this approach is strongly susceptible to **false positives**. They mitigated this threat by manually inspecting 100 posts. This practice is not scalable, so there is a strong need not to rely on the matching of strings to find suitable posts.

[2] https://github.com/danmar/cppcheck
[3] https://www.openstack.org/

3.1.2 Machine learning

Studies described before have used human-defined rules to mine security discussions, but these works still miss many posts, which may lead to an incomplete analysis of the security practices reported on Q&A websites. Traditional supervised machine learning methods can automate the mining process; however, the required negative (non-security) class is too expensive to obtain.

Le et al. proposed a novel learning framework, PUMiner, to automatically mine security posts from Q&A websites [34]. PUMiner builds a context-aware embedding model to extract features of the posts, and then develops a two-stage Positive-Unlabeled model to identify security content using the labelled Positive and Unlabeled posts. PUMiner is effective with the validation performance of at least 0.85 across all model configurations. Moreover, Matthews Correlation Coefficient (MCC) of PUMiner is 0.906, 0.534 and 0.084 points higher than one-class SVM, positive-similarity filtering, and one-stage PU models on unseen testing posts, respectively. PUMiner also performs well with an MCC of 0.745 for scenarios where string matching totally fails. Even when the ratio of the labelled positive posts to the unlabelled ones is only 1:100, PUMiner still achieves a strong MCC of 0.65.

Ye et al. brought an important new insight to exploit social coding properties in addition to code content for automatic detection of insecure code snippets in Stack Overflow through ICSD system [2]. To determine if the given code snippets are insecure, they not only analyzed the code content, but also utilize various kinds of relations among users, badges, questions, answers, code snippets and keywords in Stack Overflow. The system architecture is shown in Figure 3.2.

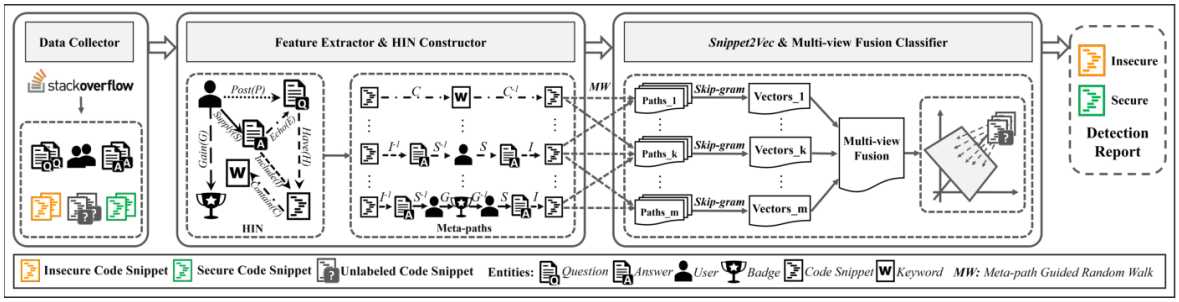


Figure 3.2: System architecture, figure is taken from [2]

It consists of the following major components:

- **Data collector**: A set of crawling tools are developed to collect the data from Stack Overflow. The collected data includes users' profiles, their posted questions and answers, and the code snippets embedded in the questions/answers.
- **Feature extractor**: Resting on the data collected from the previous module, to depict the code snippets, it first extracts the content-based features from the collected code snippets (i.e., keywords such as function names, methods and APIs), and then analyzes various relationships among different types of entities (i.e., user, badge, question, answer, code snippet, keyword), including i) question-have-code, ii) answer-include-code, iii) code-contain-keyword, iv) user-post-question, v) user-supply-answer, vi) answer-echo-question, and vii) user-gain-badge relations.

- **HIN constructor**: In this module, based on the features extracted from the previous component, a structured HIN (Heterogeneous Information Network)[4] is first presented to model the relationships among different types of entities; and then different meta-paths are built from the HIN to capture the relatedness over code snippets from different views (i.e., with different semantic meanings).

- **snippet2vec**: Based on the built meta-path schemes, to reduce the high computation and space cost, a new network embedding model snippet2vec is proposed to learn the low-dimensional representations for the nodes in HIN, which are capable to preserve both the semantics and structural correlations between different types of nodes. In snippet2vec, given a set of different meta-path schemes, a meta-path guided random walk strategy is first proposed to map the word-context concept in a text corpus into a HIN; then skip-gram is leveraged to learn effective node representation for a HIN.[5]

- **Multi-view fusion classifier**: Given different sets of meta-path schemes, different kinds of node (i.e., code snippet) representations will be learned by using snippet2vec. To aggregate these different representations, a multi-view fusion classifier is constructed to learn importance of them and thus to make predictions (i.e., the unlabeled code snippets will be predicted if they are insecure or not).

Different from ICSD, Chen et al. utilized HIN to depict relatedness over code snippets to generate code-to-code sequences, based on which sequence to sequence (seq2seq) concept in machine translation is further leveraged to learn representations of code snippets [3]. More specifically, they introduced HIN as an abstract representation, and then use meta-path to incorporate higher-level semantic relations to build up relatedness over the code snippets. Afterwards, considering both code content and social coding properties, they proposed a novel seq2seq learning model named CodeHin2Vec for representation learning of code snippets. Different from the traditional seq2seq model, CodeHin2Vec extends the basic encoder-decoder architecture by elaborately devising hierarchical attention mechanism to first learn the context between node embeddings in the input sequence, and then learn the alignments and relevances between hidden layer vectors for the output sequence generation. This allows a refined architecture to cope better with sequence modeling and thus fully exploit code content and HIN structure to learn better representations of code snippets. After that, a classifier is built for insecure code snippet detection. The system developed, called iTrustSO (shown in Figure 3.3), has the following merits:

- It introduces HIN as an abstract representation of Stack Overflow data, and exploits a meta-path based approach to characterize the relatedness over code snippets. The proposed solution provides a natural way of expressing complex relationships in social coding platforms.

[4] kind of graphical model for integrating and modeling real-world information

[5] unsupervised learning techniques used to find the most related words for a given word

- It integrates HIN with seq2seq concept for representation learning. In iTrustSO, a new model CodeHin2Vec is proposed to seamlessly combine code content and HIN-based relations to learn representations of code snippets.

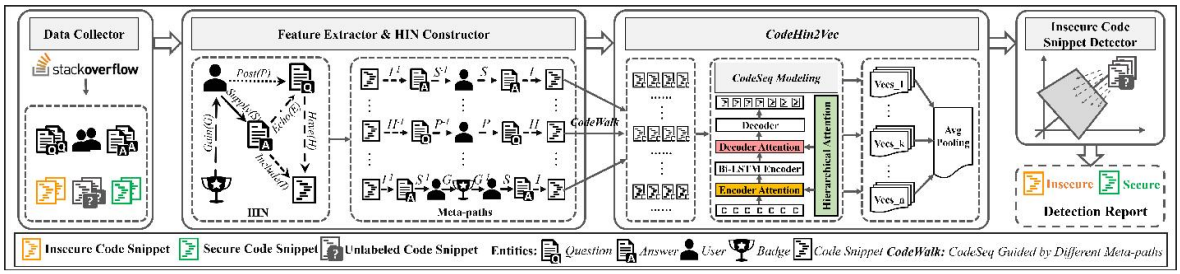


Figure 3.3: System architecture, figure is taken from [3]

It seems that representation learning is particularly useful for the detection of secure/insecure code snippets, it would be interesting to know if the fusion of a representation learning model and a deep learning model can improve performances in the detection task.

3.2 Propagation of code snippets from Stack Overflow to GitHub

Verdi et al. focused on the vulnerability of C++ code snippets shared in Stack Overflow and whether and how such vulnerable code snippets may have migrated to open source software repositories in GitHub [20]. This understanding is critical because, given the popularity of GitHub, such weak software repositories can be reused by other software repositories. To study Stack Overflow posts' evolution and their relation with GitHub, the **SOTorrent** dataset was used. In SOTorrent, each Stack Overflow post (question or answer) is identified by a Stack Overflow ID. All modifications to the posts during these years are stored in the dataset. Figure 3.4 shows the connections between Stack Overflow posts and their histories in the SOTorrent dataset. Each post in the SOTorrent dataset may contain a URL to a GitHub repository if the URL of the post is found in the GitHub repository. 99 vulnerable code snippets found in Stack Overflow were reused in a total of 2859 GitHub projects. Pieces of information about the detected vulnerabilities were presented to developers of the studied GitHub projects.

The same problem is addressed by Bai et al. [7]. The authors found a set of Stack Overflow posts that included some vulnerabilities. Next, they built a crawler to search for projects on GitHub which reused code snippets from these Stack Overflow posts entirely or in part. Finally, they interviewed the developers of these projects to further understand how and why the insecure code was propagated. For each selected Stack Overflow post, they identified lines of code in Stack Overflow posts, which were used to search projects in GitHub (Table 3.2). They built a crawler on Amazon Web Services (AWS) to use the GitHub search API to search for projects containing these lines of code. For each repository, they extracted the matched file and used the **MOSS** tool (Measure Of Software Similarity) to calculate the similarity of this code to the code snippet from the relevant Stack Overflow post. They

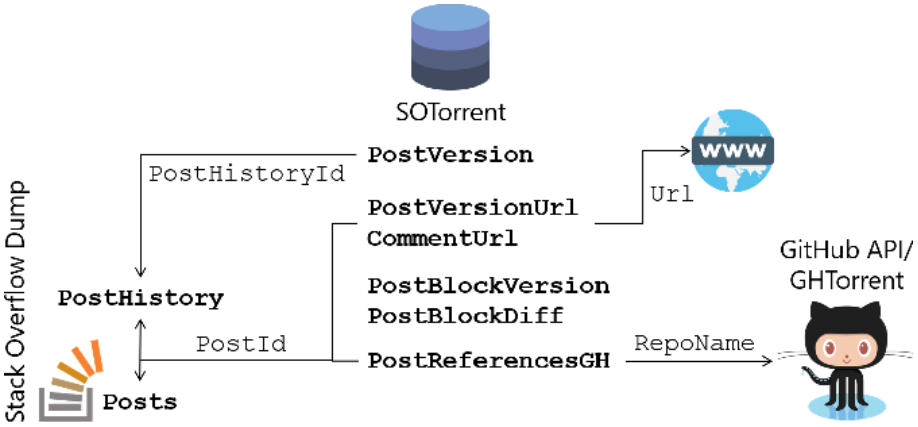


Figure 3.4: Connection of SOTorrent table to other resources

then prioritized findings based on the MOSS similarity score, as well as repository attributes like the number of stars, the number of watchers, the number of forks, the last modification date, the number of commits, and the number of contributors. In general, they favored repositories with more popular, and more recent, activities. As the work described before, also in this case there was the clone detection of insecure snippets from Stack Overflow to GitHub, but no one has thought about how that code copied on that repositories can evolve.

ID	Security Vulnerability	Description	Code Snippet Example
1	Using Crypto-insecure Pseudo-Random Number Generators (RNG)	A Crypto-insecure PRNG increases the attacker's ability to predict the random number. E.g., Using the java.util.Random function instead of java.security.SecureRandom	k += r * random.nextInt(3);
2	Using ECB mode for encryption	ECB block cipher mode for AES is not semantically secure, i.e., observing the ciphertext can reveal information about the plaintext	var decipher = crypto.createDecipheriv('aes-128-ecb', new Buffer(key, 'hex'), '');
3	Using Non-random Initialization Vector (IV) for CBC encryption	Using a non-random IV increases the attacker's ability to guess the plaintext	private String iv = "fedcba9876543210";
4	Using constant (hardcoded) encryption keys	Increases the attacker's ability to recover the encrypted text	String myKey = "StrongPassword";
5	Using constant salts for password-based encryption	Makes the ciphertext vulnerable towards dictionary-based offline attacks	byte[] salt = "DYKSalt".getBytes()
6	Using fewer than 1000 iterations for password based encryption	Decreases the average time needed for an attacker to crack the ciphertext	PBEParameterSpec pbeParamSpec = new PBEParameterSpec(salt, 20);
7	Using static seeds to seed RNGs	Non-random seeding increases the attacker's ability to predict/guess the next number in the sequence	Random random = new Random(37461831);
8	Improperly seeding RNGs	Seeding a RNG with a predictable value (e.g., time, process ID) can increase the attacker's ability to guess a future generated number	init("123456");

Table 3.2: Security vulnerabilities considered for the analysis, table is taken from [7]

3.3 Preventing copy&paste of insecure code snippets

In 2019, Fischer et al. focused on supporting software developers in making better decisions on Stack Overflow by applying **nudges**, a concept borrowed from behavioral economics and psychology [35]. The system can be represented as a learn-to-nudge loop that represents the interaction and interference of the community behavior, classification models, and proposed security nudges on Stack Overflow.

1. The community behavior on Stack Overflow triggers the loop by continuously providing and reusing code examples that introduce new use cases and patterns of cryptographic application programming interfaces (APIs).

2. In the initial step, a representative subset of these code examples is extracted and annotated by human experts. The annotations provide ground truth about the use cases and security of cryptographic patterns in the given code.

3. Then, a representation for these patterns is learned by an unsupervised neural network based on open-source projects provided by GitHub.

4. In combination with the given annotations, the pattern embeddings are used to train an additional model to predict their use cases and security.

5. Based on these predictions, they can apply security nudges on Stack Overflow by providing security warnings, reminders, recommendations, and defaults for encryption code examples.

6. Further, they allow assigned security moderators within the community to annotate unknown patterns and provide feedback on predictions of the models.

Therefore, the system creates a learn-to-nudge loop that is supposed to iteratively improve the classification models, which in turn help improve the security decisions made by the community and the security of code provided on Stack Overflow. Whenever an insecure code example is detected, a security warning, which surrounds the code, is displayed to the user. It could be an idea to add something useful to recommendations such as formal documentation that would additionally allow suggesting this solution. One possible way to achieve that is to create a link between code examples from Stack Overflow and natural language text in the official documentation.

The work just described, **tested only for Java programming language** and not on other languages, seems an intuitive and useful solution compared to the work presented by Verdi et al. [20]. In this work, they designed a browser plug-in that can warn users of the potential vulnerability in a code snippet during their visit to Stack Overflow. Whenever a developer visits a Stack Overflow post, the extension is activated. To determine whether the solution offered in the post is vulnerable, the extension consults a database of weak C++ code samples in Stack Overflow. The extension then displays a warning message to the developer explaining why the code snippet is vulnerable if the suggested solution is found to be vulnerable and suggests non-vulnerable equivalent code snippets

from other Stack Overflow posts. This method's drawback is that the vulnerabilities **database must be updated** frequently because new varieties of vulnerabilities can emerge over time.

3.4 Our work

In this work, we decide to analyze C/C++ code snippets because C/C++ are the languages that have the most security vulnerabilities [36]. As it emerged in the first section, the keyword-matching approach to finding security-related posts has limitations. Firstly, it requires considerable domain expertise and a tedious trial-and-error process to select appropriate keywords. Another important challenge of keyword-matching is to balance the precision and recall of information retrieval. If we make the assumption of labeling a post as security-related if it contains at least one of the keywords chosen, then with fewer keywords it is likely to miss many posts of interest. On the contrary, even with more relevant keywords, it does not guarantee better matching accuracy. More keywords may result in a higher false positive rate due to the multiple meanings of a word. For example, the word "exploit" also means "take advantage of" some techniques/tools in general, which is not necessarily related to a security attack. For this reason, we will do a comparison between this approach and a novel approach that uses machine learning to detect vulnerable code snippets. For the keyword-matching approach, we could focus on state-of-the-art keywords (shown in Tables 3.3 and 3.4) proposed by Hong et al. in their work where they examined the change history of Stack Overflow posts for discovering insecure code snippets [8].

C/C++ security-sensitive APIs
strcpy, strncpy, strcat, strncat, system, memcpy, memset, malloc, gets, vfork, realloc, pthread_mutex_lock, free, chroot, strlen, vsprintf, sprintf, scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf, snprintf, atoi, strtok, strcmp, strncmp, strcasecmp, strncasecmp, memcmp, signal, va_arg.

Table 3.3: Security-sensitive APIs for C/C++ posts, table is taken from [8]

For the machine learning approach, it could be interesting using Ensemble Learning [4]. As already mentioned in Section 3.2, the works have analyzed only the propagation of code weaknesses from Stack Overflow to GitHub, while we also want to study the changes over time of that piece of code in the GitHub repository to see if other vulnerabilities are introduced. As seen in the third section, security recommendations could be useful, a similar approach to security-nudges (tested only for

Category	Security-related Keywords
Nouns	vulnerab, fault, defect, sanit, mistake, flaw, bug, infinite, loop, secur, overflow, error, remote, mitigat, realloc, heap, privilege, underflow, attack, DoS, denial-of-service, initiali, xss, leak, patch, authori, corruption, crash, memory, null, injection, out-of-bounds, use-after-free, dereferenc, buffer, hack, segment, authentication, exploit.
Modifiers	incorrect, vulnerab, harm, undefine, unpredict, unsafe, secur, malicious, dangerous, critical, bad, unprivileged, negative, stable, invalid.
Verbs	flaw, hack, fix, change, modify, exploit, mitigat, leak, realloc, invoke, inject, ensure, reject, initiali, fail, authori, update, attack, trigger, lock, corrupt, crash, prevent, avoid, access, cause, overflow, terminat.

Table 3.4: Selected security-related keywords categorized as nouns, modifiers, and verbs. Table is taken from [8]

Java) can be followed for C/C++ code snippets, maybe with the adding formal documentation to suggestions. Existent extension that suggests security recommendations for C/C++ programming language uses a database that has to be continuously updated, clearly uncomfortable. We make the replication package publicly available for further investigations.[6]

[6] https://github.com/graziavarone/StackOverflow-Vulnerabilities-Evolution-In-GitHub

CHAPTER 4

Research Method

In this chapter, we present our empirical study. First, we introduce the goal and the research questions that we keep in mind during our investigation. Then, we discuss the strategies used to achieve the aim of our study.

4.1 Goal

The goal of the study was to investigate the impact of copy and pasting vulnerable code snippets from Q&A websites, to assess the life cycle of insecure codes, and after how much time they are removed from the open-source projects. The discovery of unsafe code as early as possible might be helpful not only for developers but also end-users: the former are interested in understanding why they should not copy that code, and learning more about security for those who have no experience in this field; the latter can be informed that the software used can have a vulnerability and so they could feel less attracted to use it.

4.2 Research questions

Our study was structured around three main research questions (RQs). We started analyzing how to obtain code snippets with vulnerabilities from Stack Overflow. From the analysis of related work, we described two approaches to detect vulnerable code snippets. The first consists of finding posts with code which contain at least one keyword belonging to a set chosen before. Once we have obtained code snippets, a static analysis tool is launched to determine if there are vulnerabilities. The second consists of an automatic approach through the use of machine learning. We want to compare

these two approaches to understand if an automatic approach for vulnerable code can perform well to avoid the long and tedious choice of keywords. This reasoning led to our RQ_1:

RQ_1 - The best approach to detect vulnerable code snippets

Which is the best approach for detecting Stack Overflow insecure code snippets?

Once detected code snippets useful for the analysis, we studied the propagation of the code to open-source projects. Knowledge sharing through code reuse routinely occurs between Stack Overflow and GitHub. This leads us to ask:

RQ_2 - Propagation of vulnerable code from Stack Overflow to GitHub

What is the most suitable technique for the detection of code snippets copied and pasted from Stack Overflow?

The vulnerable code migration to GitHub projects has been investigated in related work but without focusing on the change history of files in GitHub repositories containing migrated vulnerabilities. It is important to know if, sooner or later, the file containing that insecure piece of code will be modified or not, and if its modification can lead to other vulnerabilities. Hence, we asked:

RQ_3 - Changes on vulnerable code propagated on GitHub

How do C/C++ code snippets copied from Stack Overflow evolve in GitHub projects?

The following section describes the context of the investigation and how we addressed those research questions.

4.3 Research method

4.3.1 Context

The context of the study was composed of propagation and change history for vulnerable code in GitHub repositories. In our study, we focus only on repositories that respect the following criteria:

- Only C/C++ projects.
- Minimum number of releases equal to 10.
- Minimum star number equal to 10,000.
- Exclude fork.

To filter the repositories that must be analyzed in RQ_2, we use the GITHUB SEARCH ENGINE: a web application used to retrieve, store and present projects from GitHub, as well as any statistics related to them [37]. With regards to the collection of code snippets, we use SOTorrent, an open dataset

based on the official Stack Overflow data dump. The dataset chosen simplifies the study on the evolution of code snippets because it provides access to the version history of Stack Overflow content at the level of whole posts and individual text or code blocks. It connects Stack Overflow posts to other platforms by aggregating URLs from text blocks and comments, and by collecting references from GitHub files to Stack Overflow posts. In this study, we have used the last dataset version,[1] released on 2020/12/31, and based on the Stack Overflow data dump 2020/12/08. Specifically, from the dataset we filter only post containing code snippets (`<code>...</code>` tags) in the period from 2015 to 2020. We chose this time interval in order to reduce the number of code snippets to analyze later. Once obtained these posts we can remove all posts not containing C/C++ code, by making use of GUESSLANG.[2] GUESSLANG uses a deep learning Tensorflow model built with 1,900,000 unique source code files, randomly picked from 170,000 public GitHub projects. GUESSLANG accuracy is very high (93.45%) but it is not perfect because some challenging source codes that are at the border between two languages can fool GUESSLANG. In fact, a valid C source code is almost always a valid C++ code, and a valid JavaScript source code is always a valid TypeScript code. In addition to that, GUESSLANG may not guess the correct programming languages of very small code snippets. Small snippets don't always provide enough insights to accurately guess the programming language. For example, `print("Hello world")` is a valid code snippet in several programming languages including Python, Scala, Ruby, Lua, Perl, etc. In this way we have collected C/C++ codes that are used for the first step of this study that we describe in the following subsection.

4.3.2 RQ_1 - The best approach to detect vulnerable code snippets

Static analysis-based approach

This section describes how we detect code weaknesses in the C/C++ code snippets collected by Stack Overflow questions/answers by performing static code analysis. We elaborate on the details of each step below.

1. **Data collection**: All posts containing in the body at least one of the keywords, reported in Table 3.3 and Table 3.4, are obtained by filtering on posts with C/C++ code collected before. So a code snippet is taken into account for static analysis if it contains at least one of the keywords of interest. At the end of this collection step, we obtained 481,800 posts to perform static analysis on. However, a code might contain more keywords of interest, so we drop duplicates, with the aim of making unique every tuple {`PostId`, `Code`}, reducing the number of posts for the analysis to 219,574.

2. **Detecting vulnerabilities**: To detect C/C++ code snippets with weaknesses, we use a static C/C++ code analysis tool called CPPCHECK[3] to scan all resulting C/C++ code snippets. CPPCHECK is a static analysis tool for C/C++ that supports various source code level checks,

[1] SOTorrent Dataset

[2] `https://github.com/yoeo/guesslang`

[3] `https://cppcheck.sourceforge.io/`

e.g., memory/resource leaks, automatic variable checking, and bounds checking. CPPCHECK is designed so that the preprocessed code is tokenized. Some of the defects that might be detected are included in the Table 4.1.

Defect	Description
Automatic variable checking	All variables declared within a block of code are automatic by default. An uninitialized automatic variable has an undefined value until it is assigned a valid value of its type.
Bounds checking for array overruns	Method for detecting whether a variable is within some bounds before it is used. C programming language never performs automatic bounds checking to raise speed. However, this leaves many buffer overflows uncaught.
Classes checking	Unused function, variable initialization and memory duplication.
Usage of deprecated functions	Discouragement of the use of a function because it has been superseded or is no longer considered efficient or safe, without completely removing it or prohibiting it to use.
Exception safety checking	Usage of memory allocation and destructor checks.
Memory leaks	A program incorrectly manages memory allocations, in a way that memory which is no longer needed is not released.
Resource leaks	Type of resource consumption by a program that does not release resources it has acquired, for example forgetting to close a file handle.
Style errors	Examples of style errors: the scope of variables that can be reduced, values assigned but never used etc.

Table 4.1: Defects detected by CPPCHECK

Machine Learning-based approach

We reuse VELVET (noVel Ensemble Learning approach to automatically locate VulnErable sTatements) framework, which aims to predict the vulnerable statements in source code [4]. We will give first a brief introduction about how VELVET is built, and then we will describe how we want to use it in our work. In particular, VELVET analyzes the code as graphs and tries to identify vulnerable graph nodes. To this end, VELVET is trained on vulnerable/non-vulnerable samples in a supervised learning setting, where graph nodes are annotated with a vulnerability probability: vulnerable nodes are annotated with 1 and non-vulnerable nodes are marked with 0 probability. Vulnerability localization is designed as a classification task where each node is assigned a vulnerability probability and the node with the maximum vulnerability probability will be predicted as the vulnerable location. The authors conducted experiments with the two datasets, which include statement-level vulnerability annotations [38][9]:

- **JULIET**: synthetic dataset containing intentional flaws to test static analysis tools. The test cases in the dataset have vulnerabilities covering 118 different Common Weakness Enumeration (CWE) classes and well-annotated location information. We extracted 55,681 vulnerable functions with corresponding faulty locations, and also 55,681 non-vulnerable functions without any vulnerable statements.

- **D2A**: real-world dataset containing snippets collected from multiple C/C++ projects: FFmpeg[4], Httpd[5], Nginx[6], OpenSSL[7], and LibTIFF[8]. Zheng et al. collected vulnerabilities using static program analysis and differential analysis [9]. Compared to existing datasets, D2A preserves more details such as function-call traces and locations that trigger the vulnerabilities. The authors also spent much manual effort to ensure the accuracy of D2A's labels and location information. In this work, we derive our real-world dataset from D2A, and we end up with 3,506 unique functions with annotated vulnerable locations and 3,222 non-vulnerable counterparts in total.

Every sample in the datasets has one single vulnerable statement and is written in the C language.

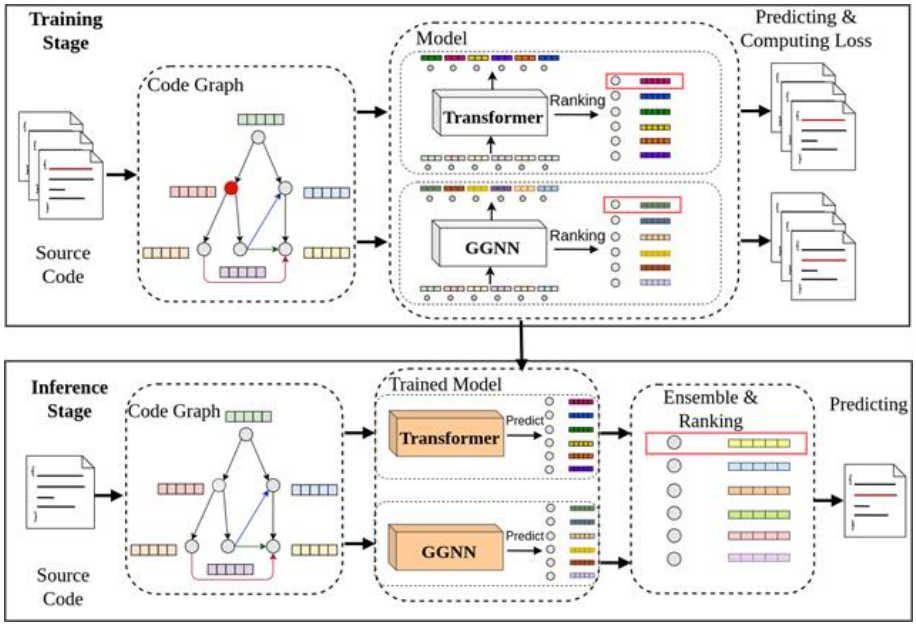


Figure 4.1: VELVET Overview, figure is taken from [4]

Figure 4.1 illustrates the workflow of the proposed approach. The Code Property Graph (CPG) is used to represent the graph semantics of programs. CPG is a code representation designed specifically for vulnerability detection. CPG combines properties of abstract syntax trees, control flow graphs and program dependence graphs in a joint data structure. This comprehensive view on code enables us to elegantly model templates for common vulnerabilities using graph traversals. Similar to the

[4] https://github.com/FFmpeg/FFmpeg
[5] https://github.com/apache/httpd
[6] https://github.com/nginx/nginx
[7] https://github.com/openssl/openssl
[8] https://gitlab.com/libtiff/libtiff

query in a database, a graph traversal passes over the code property graph and inspects the code structure, the control flow, and the data dependencies associated with each node. This joint access to different code properties enables crafting concise templates for several types of flaws and thereby helps to audit large amounts of code for vulnerabilities. JOERN is used to generate CPG [39]. The red line in the training sample represents the vulnerable statement, and the corresponding (red) node is annotated in the graph as the ground truth. In the Training Stage two distinct neural architectures are trained, a GGNN (Gated Graph Neural Network) and a Transformer, to force them to learn the diverse aspects of vulnerable patterns. In the Inference Stage, the knowledge of both models is combined to make comprehensive predictions on previously unseen data. To vectorize the node information, a word2vec model is trained over all possible code tokens in the datasets, and for each node, the token embeddings corresponding to that node are concatenated, as the initial node representation. The vectorized code graph was the input of both well-trained models, and they will output the transformed node representations H^g, H^{tr} and then the vulnerability scores, S^g and S^{tr} are computed. To aggregate the predictions, ensemble vulnerability scores are calculated, S^{en}, for all nodes by averaging: $S^{en} = 0.5*S^g + 0.5*S^{tr}$, and then the node with the highest ensemble score will be the predicted vulnerable node.

In our work, to detect vulnerable code snippets from Stack Overflow posts, we start from the material provided by VELVET authors and followed two steps [40]: the replication stage and the vulnerability detection from Stack Overflow code snippets. In the first step, Juliet and D2A datasets are preprocessed for training and save GNN and Transformers models. Once models are trained we assess how much results obtained are close to the metrics presented by VELVET authors. The Table 4.2 shows the results obtained on the two datasets used. The models trained on the Juliet dataset have the best performances, for this reason, in the next step we decide to use these models and discard what we obtained with the D2A dataset.

In the second step, we convert code snippets in CPG with JOERN. For each code snippet, we will have two CSV files: `nodes.csv` and `edges.csv`. Then each code is converted into tokens reusing the function `tokenizer` from a related work of VELVET [41], a word-based tokenization algorithm that breaks the sentence into words by splitting on operators with the support of regular expressions. A Word2Vec model is trained on all tokens obtained. Then, we create input files for VELVET models through CPG and convert each code into a vector with the Word2Vec model trained before. In the end, test each code snippet with GGNN and Transformer best models saved at the previous stage, and aggregate predictions to detect which code snippet contains vulnerable statements. With the ensemble learning approach 21,706 out of 421,024 code snippets are predicted as vulnerable.

Problems encountered: at the beginning, during the replication stage, the results obtained for training/testing/validation are far from the metrics reported in the original paper. The material code provided for VELVET does not contain how to pre-process the code of the datasets. To make sure that input files will be generated well, we involved the authors of VELVET with several questions that

Approach	JULIET					D2A				
	Classification					Classification				
	Classif. Acc.	Precision	Recall	F1	Localiz. Acc.	Classif. Acc.	Precision	Recall	F1	Localiz. Acc.
VELVET-ENSEMBLE	91%	81%	75%	77%	90%	47%	17%	5%	8%	46%
VELVET-GGNN	80%	72%	64%	66%	80%	49%	34%	18%	21%	44%
VELVET-TRANSFORMER	91%	83%	75%	78%	85%	49%	24%	8%	11%	47%

Table 4.2: VELVET results. The term *Classif. Acc.* refers to the accuracy in binary classification, while the term *Localiz. Acc.* refers to the accuracy in predicting the line where there is a vulnerability

go from the content of these files to the parameters used to train Word2Vec models. Questions about pre-processing of code snippets are clarified and it seems that the Word2Vec model has to be trained with parameter `min_count` set to 1, not reported in their paper [4]. After setting this parameter, the metrics obtained are obviously not equal but are closer to the original for the Juliet dataset than before the involvment of authors.

Comparison between the approaches

To compare the two approaches described and determine which is better, we use the dataset REVEAL as ground truth [42]. To build REVEAL, vulnerabilities are collected from two open-source software issue trackers: the first is Chromium,[9] and the other is Debian.[10] The dataset consists of two JSON files: `vulnerables.json` and `non-vulnerables.json`, with vulnerable and non-vulnerable code snippets respectively. The file `vulnerables.json` contains 2,240 code snippets, while `non-vulnerables.json` contains 20,494 code snippets. On these files we analyze which between CPPCHECK and VELVET behave better in terms of classification. Four key concepts are usually the basis for evaluation: true positives, true negatives, false positives, and false negatives. Positive and negative refer to the prediction, meaning that in this work a prediction of 'vulnerable' would be a positive, and a prediction of 'not vulnerable' would be a negative. The terms true and false refer to whether the prediction corresponds to the actual value. Hence:

- **False Positive (FP)**: Clean code incorrectly labeled as vulnerable.
- **True positive (TP)**: A vulnerability that was correctly spotted.
- **False Negative (FN)**: An actual vulnerability that was not classified as such.
- **True Negative (TN)**: A piece of code that was classified as 'not vulnerable' and is indeed harmless.

Four metrics are directly derived from those four values: precision, recall, accuracy and F1 score.

- **Precision**: Rate of true positives within all positives. It measures how precise the model is in terms of how many of the predicted positives are actual positives, or phrased differently, how

[9] `https://bugs.chromium.org/p/chromium/issues/list`
[10] `https://security-tracker.debian.org/tracker/`

much trust can be placed in the classification of a positive, and how many false alarms are produced.

$$Precision = \frac{TP}{TP + FP}$$

- **Recall**: Also called sensitivity, is a measurement for the rate of positives that were correctly identified in comparison to the total number of actual positives. One could take it as a measurement for how vigilantly the classifier spots all positives - or how much gets overlooked.

$$Recall = \frac{TP}{TP + FN}$$

- **Accuracy**: fraction of correct predictions compared to all predictions. For binary classification, it is defined as following:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

  However, accuracy does not provide much insight when there is a class imbalanced data set, meaning that there are many more positives than negatives or vice versa. In a case where true positives are rare and true negatives are very common, a classifier can achieve high accuracy scores even though it misses most of the positives, as the many true negatives make it seem like the overall outcome was quite accurate.

- **F1 score**: Balanced score that takes precision and recall into account. The F1 score is not as easily influenced by a large number of true negatives and is better suited for class-imbalanced data sets. The F1 score is defined as following:

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

  In an ideal, perfect case, the model would achieve a near 0% rate for false positives and false negatives, meaning that precision and recall both are close to 1, as well as accuracy and F1 score.

After computing and comparing the metrics reported above for the approaches, could be interesting an agreement analysis to understand how often CPPCHECK and the machine learning model:

- Predict both the right value.
- Predict both the wrong value.
- CPPCHECK predicts the right value and the machine learning model does not, and vice versa.

4.3.3 RQ_2 - Propagation of vulnerable code from Stack Overflow to GitHub

Starting from the best approach emerged in the first part of the study, we study techniques to detect re-usage of code snippets in GitHub, in this subsection we describe the approach adopted.

All C/C++ files in GitHub repositories that copy posts with vulnerable code snippets can be obtained in two ways:

- With a SQL query on the SOTorrent dataset (Listing 4.1). The table `PostReferenceGH` allows us to link Stack Overflow vulnerable question/answer and GitHub repository file filtering by the repositories name obtained with the criteria described before.

```
SELECT reference.Repo, reference.branch, reference.Path, reference.PostId,
reference.SOUrl, reference.GHUrl
FROM sotorrent-org.2020_12_31.PostReferenceGH reference
WHERE reference.PostId IN ({ids_post_vulnerable}) and reference.Repo IN ({
repositories})
```

Listing 4.1: Select repositories containing vulnerable code from Stack Overflow

- With a clone detection tool. NICAD (Automated Detection of Near-Miss Intentional Clones) allows us to find which vulnerable code snippets are reused in the repositories of interest. NICAD is a flexible TXL-based hybrid language-sensitive/text comparison software clone detection system [43]. It is a scalable, flexible clone detection tool designed to implement the hybrid clone detection method in a convenient, easy-to-use command-line tool that can easily be embedded in IDEs and other environments. It takes as input a source directory or directories to be checked for clones and a configuration file specifying several parameters like a threshold, a minimum and a maximum number of lines for clones. It provides output results in both XML form for easy analysis and HTML form for convenient browsing. NICAD handles a range of languages, including C, Java, Python, and C#, and it is designed to be easily extensible to other programming languages using a component-based plugin architecture. We first extend the NICAD tool with TXL files suitable for C++ language. With GUESSLANG we split C and C++ Stack Overflow code snippets into two different folders. For each GitHub repository, we obtain the last 30 tags, and for each of them this command is executed: `nicad6cross functions {language} {folder_code_snippets} {repository}`. Where `language` is the file extension that the tool takes into account for the clone detection, `folder_code_snippets` can be the folder of C++ codes if the language is C++ (C otherwise), and the parameter `repository` is the folder where the GitHub project is locally cloned.

4.3.4 RQ_3 - Changes on vulnerable code propagated on GitHub

Starting from the codes obtained in the previous stage, we study how code snippets of Stack Overflow in GitHub evolve, in this subsection we describe the approach adopted (summarized in Figure 4.2).

To understand if the SO code snippets used in GitHub evolve we decided to use the same approach adopted by Manes and Baysal [44], creating a table called GHCODESNIPPETHISTORY. Before explaining this approach we introduce before the concept of *Code Context*. Since we focus on the change history of code, the unit of the analysis is the commit. When we examine the code that contains

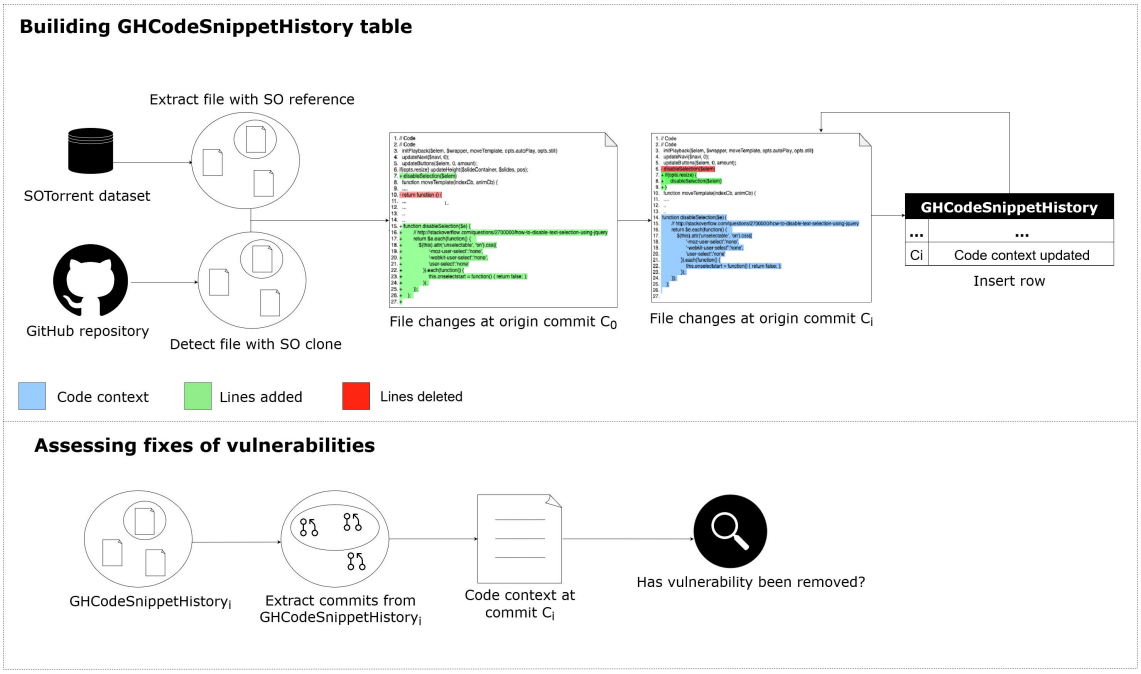


Figure 4.2: Pipeline of approach adopted

a reference to a Stack Overflow post, we first need to identify the original commit that introduced that reference. We call it the *origin commit* C_0. The changes introduced in the origin commit become the code snippet of interest, and we would like to understand how this code evolves. Suppose that a changeset δ_0 that is a part of the origin commit C_0, touches file f, that has a reference to SO post, and transforms its revision r_1 to r_2. First, we detect lines $L \in \delta_0 = \delta_{a0}$ that have been added by δ_0 in r_2. These lines attribute to the first version of the reused code snippet. To locate them, we use `git show origin-commit-hash` command. This subset δ_{a0}, defines a code context (Δ), i.e., a set of lines added to the origin commit $\Delta = \delta_{a0}$. From this point on, we want to identify all commits that change the code lines of this code context. Now consider a revision r_j to f. Let C_j be the commit that upgrades file from r_{j-1} to r_j. Let δ_j be the changeset of C_j, and $\delta_{dj} \subseteq \delta_j$ is set of deleted lines, and δ_{aj} be added lines in C_j . If, a code line, $L \in \delta_{dj}$, i.e., the line is a part of the deleted lines, and $L \in \Delta$, i.e., the line is a part of the code context as well, then C_j is modifying the current context and is added to the table, otherwise, revision r_j is not of interest to us as it is not modifying the reused snippet. If C modifies the context, we update the context as $\Delta = (\Delta \backslash \delta_d) \cup \delta'_a$, where $\delta'_a \subseteq \delta_a$, and is collection of all the lines in δ_a, that are in close proximity to the already present lines in Δ, i.e., the code context. In summary, starting from GitHub projects that reused Stack Overflow vulnerable code snippets, a GitHub's REST API request is sent to gather general information on the project such as clone URL. If the request is successful, this means that the project is public and can be cloned without infringement of copyright. The URL of such a request has the following signature: `https://api.github.com/repos/{user}/{repoName}`. After cloning the repository, the file with the reused code snippet in the codebase is located. This is the file that has a URL to Stack Overflow post. `git blame` is executed on the identified file and creates a `blame` file and producing the entire commit-graph of a given source file. This commit-graph

is then sorted in chronological order. Using the concept of code context, a subset of commits from the commit-graph is selected since these commits contribute to the change history of the reused code snippet, so the entire metadata of the commit are extracted along with the changes in the file of interest (i.e., changes to the reused snippet).

For each file, we take all commits in GHCODESNIPPETHISTORY that modifies the code snippet of interest. For each commit, we have to verify, with the best approach that emerged at RQ_1, if there are any vulnerabilities on the current version of code snippet taken in account.

CHAPTER 5

Results

This chapter describes the results obtained for each research question and the conclusions related to the main goal.

5.1 RQ_1 - The best approach to detect vulnerable code snippets

This section will provide the answer to the first research question.

RQ_1 - The best approach to detect vulnerable code snippets

Which is the best approach for detecting Stack Overflow insecure code snippets?

As mentioned in the section 4.3.2, the goal of **RQ_1** was to get the best approach for vulnerability detection. We compared a static analysis approach to a machine learning on the dataset REVEAL. The Table 5.1 shows the results obtained from this comparison.

The Table shows that static analysis can distinguish quite well, in terms of precision and recall, between the positive and the negative class for classification, unlike machine learning. We first tested the dataset REVEAL samples with ensemble learning, combining the predictions of the GGNN and Transformer. However, ensemble learning does not discriminate vulnerable codes from the not-vulnerable. Given these low performances, we wanted to check if ensemble learning could worsen the metrics, so the dataset REVEAL samples are tested using only the Transformer model. The latter is the model that achieved the best results on the Juliet dataset. From the Table, we can safely say that not only the use of ensemble learning worsened the results, but also that although the Transformer can better discriminate the positive class from the negative class, the recall metric for the positive class is lower (15%) respect to the static analysis approach (57%). Perhaps these low results for machine

Approach	Precision	Recall	F1 Score	Label
Static analysis	100%	65%	79%	Not vulnerable
	100%	57%	73%	Vulnerable
Machine learning (ensemble learning)	100%	100%	100%	Not vulnerable
	0%	0%	0%	Vulnerable
Machine learning (Transformer)	100%	68%	81%	Not vulnerable
	100%	15%	26%	Vulnerable

Table 5.1: Comparison between approaches

	Both right	Both wrong	CppCheck is right	ML is right	# samples
Not vulnerable	7,425 (36%)	1,403 (7%)	4,393 (21%)	5,008 (24%)	20,494
Vulnerable	113 (5%)	721 (32%)	898 (40%)	166 (7%)	2,240

Table 5.2: Agreement analysis

learning are caused by the fact that the models were trained and tested on a synthetic dataset, while for this comparison the models were tested on a real-world dataset. For this reason, we decided to proceed with this study using static analysis to detect vulnerabilities in code snippets.

Moreover, an agreement analysis is done between the machine learning approach (we regard the results for the Transformer model and not ensemble learning because of slightly better metrics) and static analysis approach, to understand how often CPPCHECK and the machine learning model:

- Predict both the right value.
- Predict both the wrong value.
- CPPCHECK predicts the right value and the machine learning model does not, and vice versa.

The Table 5.2 shows a summary of the agreement analysis.

The Table highlights even more the gap between static analysis and the machine learning predictions for the snippets vulnerable.

RQ_1 - Summary results

With a precision of 100% on both classes, a recall of 57% on the positive class, and a recall of 65% on the negative class, the best approach for detecting vulnerabilities in code snippets is static analysis. This approach correctly predicts 40% vulnerable code snippets of the REVEAL dataset where the machine learning approach fails.

5.2 RQ_2 - Propagation of vulnerable code from Stack Overflow to GitHub

This section will provide the answer to the second research question.

RQ_2 - Propagation of vulnerable code from Stack Overflow to GitHub

What is the more suitable technique for the detection of code snippets copied and pasted from Stack Overflow?

Before executing the query mentioned in section 4.3.3, from the code snippets analyzed by static analysis, we filtered only posts with vulnerabilities, for an amount of 116,578 vulnerable posts. The execution of the query on the table POSTREFERENCEGH returns two repositories with a reference to insecure posts: `bitcoin/bitcoin` and `tensorflow/tensorflow`. With this approach, the data obtained are few, but we have to remember that the code snippets collected at the beginning are on a time frame of five years, so it is possible that with a higher time frame, more data could be obtained from POSTREFERENCEGH.

As regards the clone detection approach, starting from 133 repositories, we found 55 repositories with clones in C files, and 80 repositories with clones in C++ files (some repositories overlapping).

For the repositories given by the query on SOTorrent, we find the commit where the reference to the SO post is introduced, this is our *origin commit*. For the repositories given by clone detection, we take into account the code clones found by NICAD at the oldest tag, this choice is made because if a code clone is introduced for more time, then it could be more possible that there are changes on it. Done the checkout at the oldest tag, we have to obtain in which commit that code clone is introduced, this is our *origin commit*.

RQ_2 - Summary results

The usage of the SOTorrent dataset turned out to be very limiting. We got only two repositories for the analysis, probably because of initial choices about data collection. Instead, NICAD has been useful for detecting clones of Type-1, Type-2, and Type-3. We obtained 55 repositories with clones for C files and 80 repositories with clones for C++ files.

5.3 RQ_3 - Changes on vulnerable code propagated on GitHub

This section will provide the answer to the third research question.

RQ_3 - Changes on vulnerable code propagated on GitHub

How do C/C++ code snippets copied from Stack Overflow evolve in GitHub projects?

As mentioned in section 4.3.4, the goal of **RQ_3** was to know if files in GitHub repositories, containing an insecure piece of code taken from Stack Overflow, will be modified or not over time and if these modifications can lead to other vulnerabilities.

Before building the GHCodeSnippetHistory table for each repository we selected from clone codes resulting from NICAD only the vulnerable snippets, because it is useless to analyze changes for a code that is never been vulnerable from the origin commit to the last commit. This leads us to exclude the majority of the clones found because they do not show vulnerabilities, resulting in only 2 repositories with vulnerabilities in C files, and 16 repositories with vulnerabilities in C++ files.

To build the GHCodeSnippetHistory table, starting from the origin commit, we execute the command `git diff` between the previous and next commit, until the last commit on that file. For each command executed, we verify if the lines deleted and added respect the requirements described in 4.3.3, if it happens we add a new row to GHCodeSnippetHistory table with:

- Repository name.
- Commit.
- File name.
- Code context updated.
- Commit date.

It is important to clarify that a file name may be renamed in a certain commit, it is necessary to have the new file name to continue the analysis until the most recent commit. This aspect is managed through git commands, thanks to the option `-M`.

Once built the history for each GitHub repository, we assessed if, in every commit reported in the history, a file shows a vulnerability in the actual code context saved. So we read every row of a GHCodeSnippetHistory table, checkout the repository considered at a specific commit, and execute the CPPCHECK tool that returns the code lines with vulnerabilities. Between these lines, we verify if a vulnerable code line is in the corresponding code context, if it happens then at that commit the file shows still a vulnerability even though there have been changes to the code context. The Table 5.3 shows the results obtained for each of the 18 repositories analyzed.

Only in 7 repositories vulnerable codes were removed from files, in the other repositories there was: (1) the editing to the corresponding code context in the file without dispose of the flawed line,

Repository	Removed	Not removed
lvgl/lvgl	✓	
xbmc/xbmc (C files)		✓
arendst/tasmota	✓	
aria2/aria2		✓
aseprite/aseprite		✓
bitcoin/bitcoin	✓	
chenshuo/muduo		✓
davisking/dlib		✓
google/filament		✓
google/leveldb		✓
microsoft/airsim		✓
opencv/opencv		✓
ossrs/srs		✓
spacehuhntech/esp8266_deauther	✓	
sqlitebrowser/sqlitebrowser	✓	
taosdata/tdengine	✓	
tensorflow/tensorflow		✓
xbmc/xbmc (C++ files)	✓	

Table 5.3: Results for code snippets propagation in GitHub repositories

or even worse (2) the file was never edited since the origin commit. This demonstrates the lack of attention about software security by developers.

RQ_3 - Summary results

For 11 out of 18 repositories, changing the code snippets does not lead to the removal of the vulnerability.

CHAPTER 6

Discussion

In this chapter, we provide the general findings and address some implications of this study.

6.1 Main Findings

The overarching theme of this work was to understand the evolution of vulnerable code snippets in Stack Overflow and how these code snippets are reused and changed on GitHub. The following subsections will provide further considerations on the results obtained.

6.1.1 RQ_1 - The best approach to detect vulnerable code snippets

The first part of this thesis deals with vulnerability detection comparison between machine learning and static analysis. We chose to analyze only C/C++ code snippets to restrict the domain of interest, but this comparison can be done also for other programming languages, for example, JavaScript, which is one of the eight most discussed topics in Stack Overflow [12]. Indeed, for creating Code Property Graphs before training VELVET is possible using JOERN also in this case because over C/C++ it supports also JavaScript as a possible programming language. VELVET has to be trained on a different dataset containing vulnerabilities in JavaScript [45], and another static analysis tool have to be used.[1] For doing the comparison it is possible taking advantage of VulinOSS, a dataset of security vulnerabilities in open-source systems, testing the two approaches on JavaScript code samples [46].

[1] https://eslint.org/

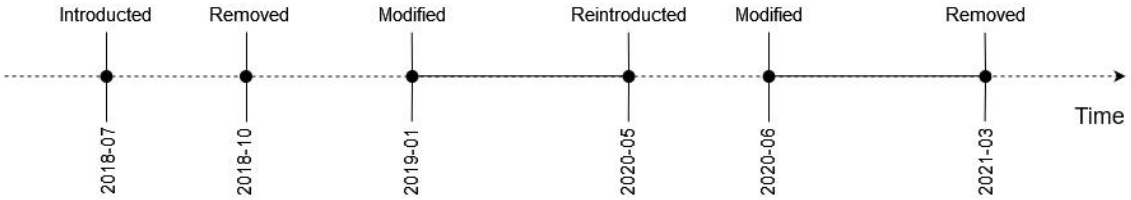


Figure 6.1: Time series plot for lvgl/lvgl

Figure 6.2: Time series plot for arendst/tasmota

6.1.2 RQ_2 - Propagation of vulnerable code from Stack Overflow to GitHub

The second part of this thesis deals with detecting code re-usages in GitHub code snippets from Stack Overflow providing two ways for finding re-usages. As mentioned in the previous chapter, the first way, is the use of SOTorrent which returns only two repositories. We hope that the choice of another most discussed programming language and a wider time frame will give more material to analyze. Instead, the second way is the NICAD clone detection tool that returns 5 repositories with clones in C files, and 80 repositories with clones in C++ files. NICAD tool offers two possible kinds of granularities: functions and blocks. In this work, we used the first kind of granularity for performance reasons, but code snippets in Stack Overflow posts are not always functions. So we suggest trying to use the second kind of granularity to obtain more data for the analysis.

6.1.3 RQ_3 - Changes on vulnerable code propagated on GitHub

The third part of this thesis deals with analyzing the change history of vulnerable code snippets copied from Stack Overflow as we reported in the previous chapter only seven out of 18 repositories removed vulnerability over time. To learn more about these evolutions, we did a quantitative analysis to understand how long after the vulnerabilities are removed and what happened in the middle. For clarity, we reported the results of the quantitative analysis in a separate time series plot for each of the seven repositories analyzed. Continuous lines in some figures mean a period of several changes to code context without removing or adding a vulnerability.

Looking at the plots, we notice that in four repositories the vulnerability is removed in the first commit after the *origin commit* (Figure 6.2, 6.3, 6.5, 6.6). But only the repository `taosdata/tdengine` removes vulnerability in "only" two months, the others three repositories remove vulnerability after two years and more. Another discussion can be done for Figure 6.4, where after the introduction of the vulnerability in 2017, the code context is edited three times but the vulnerability is removed

Figure 6.3: Time series plot for bitcoin/bitcoin

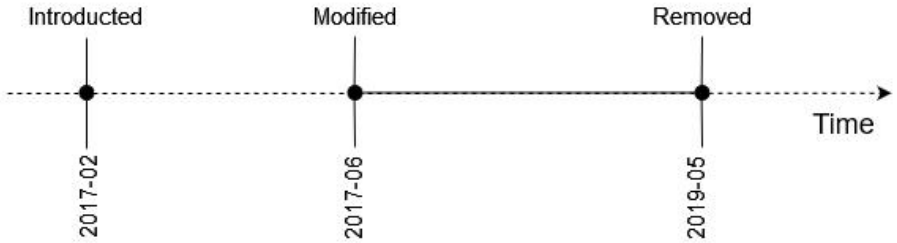


Figure 6.4: Time series plot for spacehuhntech/esp8266_deauther

only in 2019. It is interesting the case of Figure 6.1 and 6.7. In Figure 6.1, the vulnerability is removed two months later from the *origin commit*. After that, the code context is updated five times without introducing a flaw again, but in 2021 the vulnerability is reintroduced. Since 2020/06/08 the code context is modified continuously 15 times without success, only on 2021/03/10 the vulnerability is removed definitively. In Figure 6.7 there are two files with vulnerabilities. The vulnerability in the first file is removed after five years, while the vulnerability in the second file is removed after four years but introduced again in 2016 and removed definitively in 2017. More details about the quantitative analysis done is reported in Appendix A. We wanted to do further analysis to see if there was some particular event that led to the removal of vulnerabilities. To do this we looked for references to issues/pull requests in the commit messages that changed the code context. Some commits are linked to pull requests with no description provided, or are linked to pull requests for style fixes. More interesting is the case of `lvgl/lvgl` and `sqlitebrowser/sqlitebrowser` repositories. In both repositories, a commit that fixed an issue was committed when the vulnerability was removed. In detail, in the first repository, an issue about memory is solved, while in the second repository, the commit concerns outdated library issues.

6.2 Implications

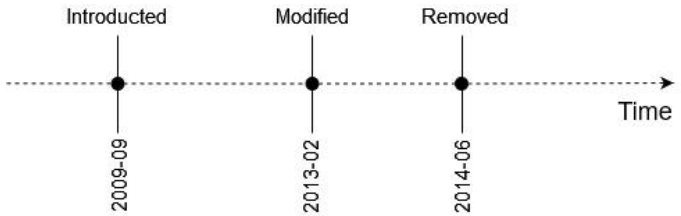
Our research may have a variety of implications and can further drive progress in other research areas. We now discuss some of the implications that are targeted to the greater scientific community working with Stack Overflow or GitHub data.

Figure 6.5: Time series plot for sqlitebrowser/sqlitebrowser

Figure 6.6: Time series plot for taosdata/tdengine

6.2.1 Software developers

We recommend software developers keep an eye on the Stack Overflow discussions as these posts and code snippets that are borrowed to GitHub projects do change over time. We suggest developers re-visit Stack Overflow every amount of time. The such improved practice would allow developers to keep their awareness of any critical changes to the code they adapt from Stack Overflow. Furthermore, while adapting a code snippet, we advise developers to study the edit history of the post, as higher activity on the answer post may be an indicator of better content quality. Given the popularity of Stack Overflow and its influence on the software developers' community, the code reuse activity on Stack Overflow is definitely beneficial. On the other hand, low-quality, inaccurate, and/or malicious code shared on Stack Overflow may be reused by thousands of developers and propagated into production software that may expose its users to security/data breaches [16]. One of the biggest problems with code reuse is that some developers reuse code snippets without thinking about what they are copying. The code may be outdated or may infringe a software license, including bugs, or expose security vulnerabilities. Therefore, to avoid such problems software developers must be extra cautious while relying on the ready-to-use code solutions from Stack Overflow. It is beneficial for developer communities to encourage expert developers to share their expertise and knowledge on online programming platforms and promote knowledge transfer. However, online programming discussion platforms such as Stack Overflow should share responsibility in curating the content by automatically detecting and removing insecure code snippets [16], providing security-related warnings to developers, or adding code snippet maturity/stability scores [44].



(a) Time series for Splash.cpp

(b) Time series for AudioDecoder.cpp

Figure 6.7: Time series plots for xbmc/xbmc

6.2.2 Researchers

Many tools for detecting clones have been developed, but many tools present limitations. Therefore, researchers should devote greater attention to this topic by developing better clone detection tools, detecting of security vulnerabilities and malicious code snippets in online platforms [16], improving Stack Overflow's answer scoring system, and more. Detecting code clones by leveraging an ML-based approach is promising since they can be applied to different programming languages, code lengths (including Stack Overflow code snippets that are mostly short), and incomplete code blocks [47]. Various online Q&A programming forums and developer communities exist nowadays, such as Stack Overflow, Stack Exchange,[2], and CodeProject.[3]. In this study, we only focus on Stack Overflow. In order to build the body of knowledge on how developers reuse available online help, it would be necessary to conduct studies beyond Stack Overflow and investigate other online platforms to better understand the best practices of code reuse, as well as the pitfalls and obstacles that developers face when reusing code from different platforms.

[2] `https://stackexchange.com/`

[3] `https://www.codeproject.com/script/Forums/List.aspx`

CHAPTER 7

Threats to Validity

In this chapter, we will analyze potential threats to the validity of this study.

7.1 Threats to Validity

According to Campbell and Stanley [48], threats to validity are either internal or external. Internal validity concerns ”controlling“ the aspects of experimental settings to ensure that the outcomes are caused only by the introduced techniques [49]. External validity refers to showing a real-world effect, but without knowing which factors caused the observed difference [49]. Cook and Campbell extended the list of validity categories to the conclusion, internal, construct, and external validity [50]. The latter classification has often been adopted in past empirical software engineering studies [51]. In order to determine the potential threats to validity that may affect our studies, we referenced Wohlin et al. guidelines [51]:

1. **Internal validity**: If a relationship is observed between the treatment and the outcome, we must make sure that it is a causal relationship, and that it is not a result of a factor of which we have no control or have not measured. In other words that the treatment causes the outcome.

2. **External validity**: The external validity is concerned with generalization. If there is a causal relationship between the construct of the cause, and the effect, can the result of the study be generalized outside the scope of our study? Is there a relation between the treatment and the outcome?

3. **Construct validity**: This validity is concerned with the relation between theory and observation. If the relationship between cause and effect is causal, we must ensure two things: 1) that the

treatment reflects the construct of the cause well and 2) that the outcome reflects the construct of the effect well.

4. **Conclusion validity**: This validity is concerned with the relationship between the treatment and the outcome. We want to make sure that there is a statistical relationship, i.e. with a given significance.

Completely avoiding/mitigating threats is often unfeasible, given the dependency between some of them: avoiding/mitigating a kind of threat (i.e., in internal validity) might intensify or even introduce another kind of threat [51]. As a result, there are inherent trade-offs between validities: with internal and external validities, for example, the more we control extraneous factors in our study, the less we can generalize our findings to a broader context. As for our studies, in the following subsections, we consider the different kinds of threats.

7.1.1 Threats to internal validity

One of the threats in this category is perhaps related to the tools used in this study: GUESSLANG and CPPCHECK. For this reason, a *deficiency of treatment setup* might have affected the study. We use the GUESSLANG tool to determine whether a code snippet is written in C/C++. GUESSLANG is based on a deep learning model trained with source code files. Although the accuracy of the tool is evaluated to be 91 percent from our 100 randomly sampled code snippets, around 10 percent of our collected C/C++ code snippets can be false positives. Code snippets that are not in C/C++ can be introduced in our study and may bias our understanding of code weaknesses in Stack Overflow posts. We detect code snippets with weaknesses using CPPCHECK. CPPCHECK can identify 59 out of the 89 types of C/C++ code weaknesses. The results that are generated by CPPCHECK can contain false positives, which may bias our results, although it aims to minimize false positives. Another kind of threat that might affect the study is the *incompleteness of data*, caused by the use of the NICAD tool. To run clone detection on GitHub repositories, we have to parse the Stack Overflow code snippets before. Some code snippets may contain syntax errors, we have to discard them. Unfortunately, for this reason, not all possible clones can be detected. Moreover, with the other approach to find codes propagated in the GitHub repositories, that is a query on the table POSTREFERENCEGH, we have obtained very few results. We guess that these few results are caused by the initial choices for collecting Stack Overflow code snippets from SOTorrent. As we said in subsection 4.3.1, only posts in the period from 2015 to 2020 are collected, probably with a wider time frame we could collect more code snippets. The more are code snippets collected, the more the probability to have GitHub repositories that have a reference to Stack Overflow code snippets.

7.1.2 Threats to external validity

The main threat in this category is perhaps related to the study of the propagation of Stack Overflow code snippets to GitHub repositories. For this reason, an *interaction of selection and treatment*

might have affected the study. We analyzed the history of 18 repositories, concluding that 11 of them never remove vulnerability from the file. However, the number of repositories studied is limited, it could be possible that with a greater sample of histories, the results change. Since we do not have enough data to generalize these results, we cannot conclude that in most cases the GitHub repositories never remove vulnerabilities from files.

7.1.3 Threats to construct validity

The main threat in this category is perhaps related to the selection of code snippets with a set of keywords. For this reason, the *appropriateness of data* might have affected the study. This study selects security-related codes based on the keywords summarized by prior work [8], so our studied code snippets are limited to these rules. We relied on state-of-the-art keywords because their choice requires considerable domain expertise. The most extensive list has 127 security keywords proposed by Pletea et al. [52]. However, Le et al. have found that such a list is still missing many important security keywords [34]. With further investigation, they found out that missing the keyword "ssh" (a cryptographic network protocol) alone would result in at least 32,216 potential posts being overlooked. In conclusion, with the keywords inherited by [8], we might not find all security-related posts suitable for our study. Another threat relates to the fine-tuning of the VELVET model. We tried to train GGNN and Transformer models on the D2A dataset, but the metrics were very low (see Table 4.2). We invite further investigation into using machine learning models trained on real-world projects but with good metrics for another comparison with a static analysis tool. One of the popular datasets is BIG-VUL [53]. It is one of the largest vulnerability datasets which includes line-level ground-truths. The dataset is collected from 348 open-source GitHub projects, which includes 91 different CWEs from 2002 to 2019, 188,636 C/C++ functions with a ratio of vulnerability functions of 5.7% (i.e., 10,900 vulnerable functions), and 5,060,449 LOC with a ratio of vulnerable lines of 0.88% (i.e., 44,603 vulnerable lines). BIG-VUL has been widely used in research with good results, achieving an F-measure of 91%, a Precision of 97%, and a Recall of 86% in predicting vulnerabilities at function-level [54].

7.1.4 Threats to conclusion validity

The main threat in this category is perhaps related to the replication of VELVET. In our study we have replicated the VELVET approach as best as possible, however, the results obtained by Ding et al. and reported in Table 7.1, are a little bit different from the results obtained in this work [4]. This is probably due to the *lack of data preprocessing*, though we have followed all the steps (from the conversions in CPGs with JOERN, passing for Word2Vec training, until the creation of files with required keys), we could be wrong by entering in files some values that negatively affect the training of our models. Despite this, we cannot say even with the results of Table 7.1 which approach is better between static analysis and machine learning in our case, because even if the three C/C++ static

Approach	JULIET						D2A					
	Pred	Vul-CLS				Vul-LOC	Pred	Vul-CLS				Vul-LOC
	Acc.	Acc.	Precision	Recall	F1	Acc.	Acc.	Acc.	Precision	Recall	F1	Acc.
VELVET-ENSEMBLE	**99.5%**	**99.6%**	**99.9%**	**99.3%**	**99.6%**	**99.6%**	**51.1%**	58.9%	**76.2%**	48.1%	**59.0%**	**30.1%**
VELVET-GGNN	93.6%	94.2%	**99.9%**	89.0%	94.2%	98.5%	45.5%	56.7%	73.2%	39.1%	51.0%	19.6%
VELVET-TRANSFORMER	99.3%	**99.6%**	**99.9%**	**99.3%**	**99.6%**	99.1%	47.2%	**59.3%**	70.5%	**50.4%**	58.8%	29.3%
Infer*	N/A	69.4%	41.5%	54.4%	47.1%	36.9%	N/A	N/A	N/A	N/A	N/A	N/A
FlawFinder	N/A	58.3%	36.6%	51.0%	42.6%	8.6%	N/A	48.7%	53.3%	6.0%	10.7%	6.7%
RATS	N/A	59.3%	36.5%	46.4%	40.9%	N/A	N/A	45.8%	47.9%	16.4%	24.2%	N/A

Table 7.1: VELVET results, table is taken from [4]. *They do not compare with Infer on D2A, since Zheng et al. [9] used Infer during data collection. Thus, it is unfair to compare with Infer on D2A.

code analyzers (Infer[1], FlawFinder [2], RATS[3]) reported in Table have performed worse, there was no comparison with CPPCHECK, that is the static code analysis tool used in our study.

[1]Infer
[2]FlawFinder
[3]RATS

CHAPTER 8

Conclusions

This chapter provides a summary of the research and future perspectives.

8.1 Conclusions

Stack Overflow like other crowd-sourced platforms is designed to stimulate knowledge exchanges between developers. However, this platform is not equipped with a robust mechanism to ensure the good quality of answers and code snippets exchanged by its users. The incentive system (e.g., reputation, badges, upvotes, downvotes) in Stack Overflow was designed partly to encourage users to share quality contents. However, this approach works only when each user is responsible and–or knowledgeable enough about every detail of the shared knowledge, which given the complex nature of software development is a difficult task.

In this thesis, we have analyzed vulnerabilities in C/C++ code snippets shared on Stack Overflow and their migration/evolution to GitHub projects. This is the first study that is not limited to the verification of the insecure code snippet still alive in the GitHub file, but we want to study the overall history of the code snippet, from its introduction to its removal. We have investigated security vulnerabilities in the C/ C++ code snippets shared on Stack Overflow over 5 years. We compared machine learning and static analysis on vulnerability detection achieving better performances for the static analysis, with a precision of 100% on both classes, a recall of 65% for the negative class, and a recall of 57% for the positive class. Starting from the 219,574 code snippets collected from Stack Overflow with keyword-matching, we found re-usages (through SOTorrent and clone detection) of vulnerable code in 18 GitHub repositories. For seven repositories, vulnerabilities were removed from files after some years. Furthermore, two of the repositories committed changes that remove vulnerabilities from files with the particularity that these two commits are the fixes of open issues.

The issues may be caused by the vulnerable code snippets copied from Stack Overflow. The other 11 repositories still have not corrected vulnerabilities. These results suggest that most of the developers are unaware of what they are copying from Stack Overflow.

8.2 Future Work

Based on the obtained results, a solution to avoid the use of unsafe code from online forums is needed. In this way, developers are aware of the risks of that code snippets and it should be lower propagation in GitHub projects. Starting from a vulnerable code in a post and its question associated, it is possible to prevent the reuse of code with weaknesses in GitHub repositories with the following two steps: the first is finding a certain number of duplicated questions of the initial where there is the code vulnerable; then, for each duplicated question all answers are analyzed, if one of these answers does not report vulnerabilities, the link to this answer can be given to help developers.

8.2.1 A browser extension to avoid the copy of vulnerable code in Stack Overflow

To inform users about the existence of vulnerabilities in a code snippet posted on Stack Overflow, a browser extension may be developed. The extension gets activated when a developer visits a Stack Overflow post. With an arbitrary approach for detecting vulnerabilities, the code read in the post is verified to understand if it is secure or not. If the provided solution is indeed found vulnerable, the extension then shows a warning message to the developer. The extension then recommends non-vulnerable similar code snippets from other Stack Overflow posts, starting from the question related, so that the developer can reuse those safe code snippets instead of the vulnerable code snippet. To recommend similar code snippets could be useful finding duplicate questions. Since there are thousands of questions submitted to Stack Overflow every day, manually identifying duplicate questions is difficult. Thus, there is a need for an automated approach that can help in detecting these duplicate questions. To address this need, it is possible to use an automated approach named DUPPREDICTOR, which takes a question as input and detects potential duplicates of this question by considering multiple factors [5]. DUPPREDICTOR extracts the title and description of a question and also tags that are attached to the question. DUPPREDICTOR then computes the latent topics of each question by using a topic model. Next, for each pair of questions, it computes four similarity scores by comparing their titles, descriptions, latent topics, and tags. These four similarity scores are finally combined to result in a new similarity score that comprehensively considers the multiple factors. To examine the benefit of DUPPREDICTOR, an experiment is performed on a Stack Overflow dataset which contains a total of more than two million questions. The result showed that DUPPREDICTOR can achieve a recall-rate@20 score of 63.8%. Fig. 8.1 presents the overall framework of DUPPREDICTOR.

The framework contains two phases: the model-building phase and the prediction phase. In the model-building phase, the goal is to train a model from historical duplicate questions which have been detected. In the prediction phase, this model would be used to detect new duplicate questions. For

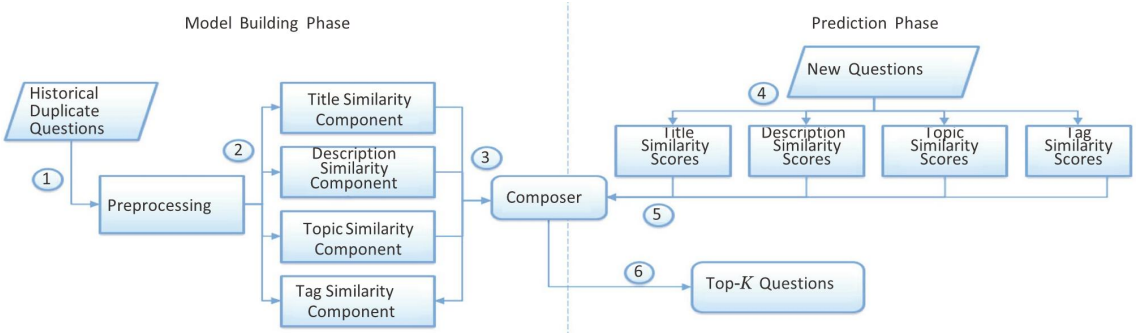


Figure 8.1: Overall framework of DUPPREDICTOR, figure is taken from [5]

each pair of questions, four scores capture the similarity of the questions. These scores are computed by the title similarity component, description similarity component, topic similarity component, and tag similarity component. Next, these four sets of similarity scores are input into the composer component, which would then automatically learn a good weight for each of the four components. After the composer component has learned the weights, in the prediction phase, it is used to return a list of K historical questions that are potential duplicates of a new question.

So thanks to this framework, after finding the most similar questions, for each of them, all answers are scanned with the aim to find a safe code snippet to help the user.

APPENDIX A

Quantitative Analysis

The tables obtained after the quantitative analysis are shown below.

Repository	File	Commit	Date	Vulnerability
arendst/tasmota	lib/Adafruit_SGP30-1.0.3/Adafruit_SGP30.cpp	67477b1	2018-03-31	Introducted
arendst/tasmota	lib/Adafruit_SGP30-1.0.3/Adafruit_SGP30.cpp	615645e	2020-05-23	Removed

Table A.1: Quantitative analysis for arendst/tasmota

Repository	File	Commit	Date	Vulnerability
bitcoin/bitcoin	src/qt/guiutil.h	5659e73	2020-11-25	Introducted
bitcoin/bitcoin	src/qt/guiutil.h	6958a26	2022-04-16	Removed

Table A.2: Quantitative analysis for bitcoin/bitcoin

Repository	File	Commit	Date	Vulnerability
sqlitebrowser/sqlitebrowser	libs/qhexedit/src/commands.cpp	0cf9ecd	2013-03-16	Introducted
sqlitebrowser/sqlitebrowser	libs/qhexedit/src/commands.cpp	ff302c4	2016-08-01	Removed

Table A.3: Quantitative analysis for sqlitebrowser/sqlitebrowser

Repository	File	Commit	Date	Vulnerability
spacehuhntech/esp8266_deauther	esp8266_deauther/Settings.cpp	1bccb3e	2017-02-25	Introducted
spacehuhntech/esp8266_deauther	esp8266_deauther/Settings.cpp	3ae0a10	2017-06-18	Modified
spacehuhntech/esp8266_deauther	esp8266_deauther/Settings.cpp	24d9043	2018-03-24	Modified
spacehuhntech/esp8266_deauther	esp8266_deauther/Settings.cpp	a235458	2019-05-14	Removed

Table A.4: Quantitative analysis for spacehuhntech/esp8266_deauther

Repository	File	Commit	Date	Vulnerability
taosdata/tdengine	source/libs/parser/test/parInsertTest.cpp	3954edb	2022-06-13	Introducted
taosdata/tdengine	source/libs/parser/test/parInsertTest.cpp	b420bf4	2022-08-15	Removed

Table A.5: Quantitative analysis for taosdata/tdengine

Repository	File	Commit	Date	Vulnerability
xbmc/xbmc	xbmc/utils/Splash.cpp	45285e8	2009-09-23	Introducted
xbmc/xbmc	xbmc/utils/Splash.cpp	b0fa5b3	2013-02-09	Modified
xbmc/xbmc	xbmc/utils/Splash.cpp	7222153	2014-06-10	Removed
xbmc/xbmc	xbmc/cores/paplayer/AudioDecoder.cpp	45285e8	2009-09-23	Introducted
xbmc/xbmc	xbmc/cores/paplayer/AudioDecoder.cpp	349ec40	2012-05-10	Modified
xbmc/xbmc	xbmc/cores/paplayer/AudioDecoder.cpp	92e8bc4	2013-04-05	Removed
xbmc/xbmc	xbmc/cores/paplayer/AudioDecoder.cpp	a09e223	2016-03-06	Reintroducted
xbmc/xbmc	xbmc/cores/paplayer/AudioDecoder.cpp	58d81a7	2017-11-20	Removed

Table A.6: Quantitative analysis for xbmc/xbmc

Repository	File	Commit	Date	Vulnerability
lvgl/lvgl	src/lv_misc/lv_mem.c	1b9845e	2018-07-25	Introducted
lvgl/lvgl	src/lv_misc/lv_mem.c	41695bf	2018-10-05	Removed
lvgl/lvgl	src/lv_misc/lv_mem.c	eca9245	2019-01-28	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	f5bd68f	2019-01-28	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	31d0a7f	2019-01-28	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	edb58cc	2019-03-15	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	ba21600	2019-04-04	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	97392f4	2020-05-13	Reintroducted
lvgl/lvgl	src/lv_misc/lv_mem.c	0ebcf7e	2020-06-08	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	585bc32	2020-06-15	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	25fbcea	2020-06-18	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	43f5e4d	2020-07-07	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	67d268b	2020-08-31	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	d6dd619	2020-11-25	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	ec12455	2020-12-11	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	81b13bf	2021-01-11	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	bc25998	2021-01-12	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	1d6d2eb	2021-01-21	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	ea6ee3d	2021-01-23	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	0726882	2021-01-26	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	8a1af86	2021-02-04	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	06917a6	2021-02-19	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	8004058	2021-02-24	Modified
lvgl/lvgl	src/lv_misc/lv_mem.c	7bf547a	2021-03-10	Removed

Table A.7: Quantitative analysis for lvgl/lvgl

Acknowledgements

First of all, I would like to thank my supervisor Prof. Fabio Palomba because the Software Dependability lessons that I attended two years ago were very inspirational for me. In his explanations I have always perceived a certain enthusiasm about what he deals with, the same enthusiasm I have put in these five years and that I will put in any challenge I face. These were the reasons that led me to choose him as my supervisor for my thesis, having the opportunity of deepening a topic that I have always found interesting: software vulnerabilities.

I can not mention Dr. Emanuele Iannone and Dr. Giulia Sellitto for their patience, suggestions, and flawless guidance during these months. I am grateful that you have always helped me to face all the obstacles encountered. Thanks for the daily motivation, especially when previously decided plans have not produced the expected results, but despite this the day later we had already decided together an alternative. Thank you for your availability.

Special thanks go to my family, without whom I would never have reached this milestone in my life. You have always supported me and allowed me to make the choices I wanted without any limitations. Thank you, really.

Thanks to Anna, Martina, and Valeria because they are an example for me due to their determination. They made me believe more in myself. But mostly they taught me not to give up and to love the challenges in every context. Challenges **ESPECIALLY** against Martina in board games where one day we will beat her so we will have the honor of saying that at least one time she did not win (Martina we love you XOXO).

To my six queens: Alessia, Giusy, Mary, Noemi, Rosaria and Teresa. I feel lucky to have met you and to have faced many difficulties together since our first year of university. Even though we can't see

each other as often as years ago, every time we succeed to see each other (surprisingly) and spend a day together I am happy because I am so pleased to spend time with you. Although it has been several years nothing has changed in our friendship and I hope it never changes, you have made me smile over the years and you still make me smile. I conclude with this beautiful sticker that summarizes all our precision and organization, but above all our being all in agreement:

I thank Armando for his sweetness and his comprehension. Thanks for listening to all my complaints, even the dumbest ones without ever minimizing them. Thanks for all the times when I thought I could not do something but you gave me the strength to bring out the best in me. When I am with you all my anxieties and worries disappear (how sweet I am) because you can bring only optimism into my life. **EVEN IF** sometimes you make me *un poco loco*, I care about you so much, and I will always be by your side. *"Your skin and bones, turn into something beautiful, and you know, you know I love you so"*.

Finally, I must also thank my team for making me feel included from day one. Each of you taught me so much and together we faced so much, being able with team spirit to face technical challenges that initially seemed difficult, *SpectAngular*. I could not wish to be on a better team than this because it is amazing, but maybe the last word is better to say in Italian with another tone of voice: *ECCEZIONALE ECCEZIONALE*.

Bibliography

[1] S. Baltes, L. Dumani, C. Treude, and S. Diehl, "Sotorrent: reconstructing and analyzing the evolution of stack overflow posts" in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018* (A. Zaidman, Y. Kamei, and E. Hill, eds.), pp. 319–330, ACM, 2018. (Citato alle pagine iv, 20 e 21)

[2] Y. Ye, S. Hou, L. Chen, X. Li, L. Zhao, S. Xu, J. Wang, and Q. Xiong, "Icsd: An automatic system for insecure code snippet detection in stack overflow over heterogeneous information network" in *Conference: the 34th Annual Computer Security Applications Conference*, pp. 542–552, 12 2018. (Citato alle pagine iv e 28)

[3] L. Chen, S. Hou, Y. Ye, T. Bourlai, S. Xu, and L. Zhao, "itrustso: an intelligent system for automatic detection of insecure code snippets in stack overflow" in *Conference: ASONAM '19: International Conference on Advances in Social Networks Analysis and Mining*, pp. 1097–1104, 08 2019. (Citato alle pagine iv, 29 e 30)

[4] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, and B. Ray, "Velvet: a novel ensemble learning approach to automatically locate vulnerable statements" 2021. (Citato alle pagine iv, vi, 33, 38, 39, 41, 58 e 59)

[5] Y. Zhang, D. Lo, X. Xia, and J.-L. Sun, "Multi-factor duplicate question detection in stack overflow" *Journal of Computer Science and Technology*, vol. 30, no. 5, p. 981, 2015. (Citato alle pagine v, 61 e 62)

[6] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags, "How reliable is the crowdsourced knowledge of security implementation?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 536–547, 2019. (Citato alle pagine vi, 25 e 26)

[7] W. Bai, O. Akgul, and M. L. Mazurek, "A qualitative investigation of insecure code propagation from online forums" in *2019 IEEE Cybersecurity Development (SecDev)*, pp. 34–48, 2019. (Citato alle pagine vi, 30 e 31)

[8] H. Hong, S. Woo, and H. Lee, "Dicos: Discovering insecure code snippets from stack overflow posts by leveraging user discussions" in *ACSAC '21: Annual Computer Security Applications Conference*, pp. 194–206, 12 2021. (Citato alle pagine vi, 33, 34 e 58)

[9] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su, "D2a: A dataset built for ai-based vulnerability detection methods using differential analysis" 2021. (Citato alle pagine vi, 38, 39 e 59)

[10] "All sites – stack exchange" 2023. `https://stackexchange.com/sites?view=list#users`. (Citato a pagina 1)

[11] "Stack overflow developer survey 2022" 2022. `https://survey.stackoverflow.co/2022/`. (Citato a pagina 1)

[12] "Tags." `https://stackoverflow.com/tags`. (Citato alle pagine 1 e 51)

[13] A. Anderson, D. Huttenlocher, J. Kleinberg, and J. Leskovec, "Discovering value from community activity on focused question answering sites: A case study of stack overflow" in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, (New York, NY, USA), p. 850–858, Association for Computing Machinery, 2012. (Citato a pagina 1)

[14] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. Arango-Argoty, "Secure coding practices in java: Challenges and vulnerabilities" in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 372–383, 2018. (Citato alle pagine 2 e 25)

[15] A. Rahman, E. Farhana, and N. Imtiaz, "Snakes in paradise?: Insecure python-related coding practices in stack overflow" in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 200–204, 2019. (Citato alle pagine 2 e 27)

[16] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? the impact of copy&paste on android application security" in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 121–136, 2017. (Citato alle pagine 2, 24, 54 e 55)

[17] D. Yang, P. Martins, V. Saini, and C. Lopes, "Stack overflow in github: Any snippets there?" 2017. (Citato a pagina 2)

[18] C. Ragkhitwetsagul, J. Krinke, M. Paixao, G. Bianco, and R. Oliveto, "Toxic code snippets on stack overflow" *IEEE Transactions on Software Engineering*, vol. 47, pp. 560–581, mar 2021. (Citato a pagina 2)

[19] L. An, O. Mlouki, F. Khomh, and G. Antoniol, "Stack overflow: A code laundering platform?" 2017. (Citato a pagina 2)

[20] M. Verdi, A. Sami, J. Akhondali, F. Khomh, G. Uddin, and A. K. Motlagh, "An empirical study of c++ vulnerabilities in crowd-sourced code examples" 2019. (Citato alle pagine 3, 30 e 32)

[21] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley, 2007. (Citato a pagina 5)

[22] IEEE, "Ieee standard glossary of software engineering terminology" *IEEE Std 610.12-1990*, 1990. (Citato a pagina 10)

[23] "Stack exchange data dump" 2018. `https://archive.org/details/stackexchange`. (Citato a pagina 18)

[24] "The state of the octoverse." `https://octoverse.github.com`. (Citato a pagina 20)

[25] "Google cloud platform. github data" 2018. `https://cloud.google.com/bigquery/public-data/github`. (Citato a pagina 21)

[26] B. S. Baker, "On finding duplication and near-duplication in large software systems" in *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, (USA), p. 86, IEEE Computer Society, 1995. (Citato a pagina 22)

[27] 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*, vol. 28, no. 7, pp. 654–670, 2002. (Citato a pagina 23)

[28] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics" *1996 Proceedings of International Conference on Software Maintenance*, pp. 244–253, 1996. (Citato a pagina 23)

[29] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees" *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377, 1998. (Citato a pagina 23)

[30] J. Krinke, "Identifying similar code with program dependence graphs" in *Proceedings Eighth Working Conference on Reverse Engineering*, pp. 301–309, 2001. (Citato a pagina 23)

[31] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees" *2006 13th Working Conference on Reverse Engineering*, pp. 253–262, 2006. (Citato a pagina 23)

[32] R. Tairas and J. Gray, "Phoenix-based clone detection using suffix trees" in *ACM-SE 44*, 2006. (Citato a pagina 23)

[33] H. Zhang, S. Wang, H. Li, T.-H. Chen, and A. E. Hassan, "A study of c/c++ code weaknesses on stack overflow" *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2359–2375, 2022. (Citato a pagina 26)

[34] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "PUMiner" in *Proceedings of the 17th International Conference on Mining Software Repositories*, ACM, jun 2020. (Citato alle pagine 28 e 58)

[35] F. Fischer, H. Xiao, C.-Y. Kao, Y. Stachelscheid, B. Johnson, D. Razar, P. Fawkesley, N. Buckley, K. Böttinger, P. Muntean, and J. Grossklags, "Stack overflow considered helpful! deep learning security nudges towards stronger cryptography" in *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, (USA), p. 339–356, USENIX Association, 2019. (Citato a pagina 32)

[36] "What are the most secure programming languages?." `https://www.mend.io/most-secure-programming-languages/`. (Citato a pagina 33)

[37] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies" in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pp. 560–564, IEEE, 2021. (Citato a pagina 36)

[38] NIST, "Juliet test suite v1.3" 2017. `https://samate.nist.gov/SRD/testsuite.php`. (Citato a pagina 38)

[39] "Joern." `https://github.com/octopus-platform/joern`. (Citato a pagina 40)

[40] "Velvet." `https://github.com/ARiSE-Lab/VELVET`. (Citato a pagina 40)

[41] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks" 2019. (Citato a pagina 40)

[42] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" 2020. (Citato a pagina 41)

[43] J. R. Cordy and C. K. Roy, "The nicad clone detector" in *2011 IEEE 19th International Conference on Program Comprehension*, pp. 219–220, 2011. (Citato a pagina 43)

[44] S. S. Manes and O. Baysal, "Studying the change histories of stack overflow and github snippets" in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 283–294, 2021. (Citato alle pagine 43 e 54)

[45] R. Ferenc, P. Hegedűs, P. Gyimesi, G. Antal, D. Bán, and T. Gyimóthy, "Challenging machine learning algorithms in predicting vulnerable javascript functions" in *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pp. 8–14, 2019. (Citato a pagina 51)

[46] A. Gkortzis, D. Mitropoulos, and D. Spinellis, "Vulinoss: A dataset of security vulnerabilities in open-source systems" in *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, (New York, NY, USA), p. 18–21, Association for Computing Machinery, 2018. (Citato a pagina 51)

[47] Y.-Y. Zhang and M. Li, "Find me if you can: Deep software clone detection by exploiting the contest between the plagiarist and the detector" in *AAAI Conference on Artificial Intelligence*, 2019. (Citato a pagina 55)

[48] D. T. Campbell, "Experimental and quasi-experimental designs for research in teaching" *Handbook of research on teaching*, 1963. (Citato a pagina 56)

[49] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 9–19, 2015. (Citato a pagina 56)

[50] T. Cook and D. Campbell, *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin, 1979. (Citato a pagina 56)

[51] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer US, 2000. (Citato alle pagine 56 e 57)

[52] D. Pletea, B. Vasilescu, and A. Serebrenik, "Security and emotion: Sentiment analysis of security discussions on github" in *International Working Conference on Mining Software Repositories (MSR)*, pp. 348–351, 05 2014. (Citato a pagina 58)

[53] J. Fan, Y. Li, S. Wang, and N. Tien, "A c/c++ code vulnerability dataset with code changes and cve summaries" in *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 508–512, 06 2020. (Citato a pagina 58)

[54] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction" in *Proceedings of the 19th International Conference on Mining Software Repositories*, 03 2022. (Citato a pagina 58)