# Search for duplication and cloned codes

Lucas Collemare

08/07/2024

LRE
LABORATOIRE DE RECHERCHE DE L'EPITA

## Sommaire

## 1 Abstract

Code deduplication is a major problem that faces several areas of discussion. First of all, we can think of cheating for students, or even the theft of resources. But the subject that will interest us here is Common Vulnerabilities and Exposures (CVEs). This report describes work that was done to create a tool capable of propagating security vulnerability fixes. The goal here is to find, using a code duplication search, the actors affected by these security flaws to warn them of a vulnerability correction in their program. This security vulnerability correction will be based on a CVE. This report mainly concerns developers, security analysts and software architects

## 2 Introduction

Code duplication, commonly seen in software development, has both advantages and disadvantages. While it can speed up development by reusing proven blocks of code, it can also propagate security vulnerabilities, known as Common Vulnerabilities and Exposures (CVEs)[1], across various software projects. This research aims to explore cloned code detection with the specific goal of propagating security fixes efficiently and reliably, using advanced tools like Chroma for embeddings and Streamlit for visualization.

### 2.1 Context

Code reuse is common, especially in open-source environments, developers can unintentionally introduce security vulnerabilities by copying and pasting vulnerable code. This is risky in sectors where safety is paramount. For example, a computer science student may copy a piece of code for a project without realizing that the source code contains vulnerabilities. Similarly, a developer can reuse code found in a public repository without knowing that this code is broken or obsolete.

### 2.2 Goals

This study focuses on implementing a means to detect cloned codes and perform CVE propagation. The goal is to develop an application capable of scanning a database to identify duplications, based on known vulnerabilities. By associating the results of this detection with CVE databases, it is possible to alert developers to security risks and offer them recommended fixes.

### 2.3 Methodology

Chroma, an artificial intelligence-based tool for analyzing and comparing code embeddings, will be the primary tool to detect semantic similarities that could indicate duplication. Streamlit is used to create an interactive user interface where users can load codes, perform comparisons and view analysis results.

In practical terms, this application could for example be used to prevent students from cheating by checking the uniqueness of their work, or to help companies ensure that their code does not contain vulnerable portions already identified in other projects. Through this approach, this study contributes to securing software development and reducing the risk of propagating security vulnerabilities through code reuse.

# 3 Tools

## 3.1 Chroma

Chroma is an open-source, AI-native vector database that facilitates the development of applications using LLMs[2]

Chroma allows you to store not only embeddings (dense vector representations from language models) but also their associated metadata. This feature is practical for users who work with a lot of text or documents and who need to find precise information efficiently. Using Chroma, users can insert documents and perform queries by transforming them into embeddings, simplifying complex searches on large datasets.

For example, finding duplicate code can benefit from using Chroma to index and search similar code fragments across multiple code repositories. By transforming code snippets into embeddings and storing them in Chroma, researchers can quickly identify similarities that could indicate instances of code duplication.

In summary, Chroma is a powerful tool, enabling efficient vector data management with advanced search capabilities.

## 3.2 Streamlit

Streamlit is an open-source framework that simplifies the development of web applications for data science and artificial intelligence projects. In our context, Streamlit offers an ideal platform to easily visualize the results of cloned code analysis[3]

The Streamlit user interface allows you to create interfaces where you can display information such as in our case the code snippets suspected of being duplicated as well as their associated metadata. This visualization capability helps communicate results effectively to users.

Additionally Streamlit allows users to change search parameters and filter results. For example, one could use a slider to adjust the detection threshold, instantly observing how this affects the number of duplications detected.

Streamlit acts as the front-end where results can be analyzed in detail, providing a unified platform for examining and managing vulnerabilities related to cloned code. Its ease of use combined with its powerful interaction capacity makes it a very good tool in our context and to respond to our problem.

# 4 Code Cloned Types

The question here is how to define cloned code. Because indeed, if two codes do the same thing but are not similar, can we say that they are cloned codes ? There are several types of cloned code, each with their own characteristics, ranging from type 1 (the most similar) to type 4 (the least similar).

## 4.1 Type 1: Identical Similarity

Type 1, or exact clone, involves directly copying a block of code to another location without any modification. This type of clone is the easiest to detect because the code segments are identical character for character.

## 4.2 Type 2: Lexical Similarity

Type 2 cloned code includes minor variations in the code, such as changes in variable names, data types, or slight modifications to logical expressions. Minor changes can hide duplication detection.

## 4.3 Type 3: Syntaxical Similarity

Type 3 is cloned code that introduces significant additions or deletions of code. This type may include additional loops, conditions, or separate branches of code. This type of cloned code can be more difficult to identify because it requires analysis that goes beyond simple textual comparison and engages in understanding the logic of the code. It combines elements of original and new code, requiring semantic analysis to correctly identify the nature of the duplication.

## 4.4 Type 4: Semantic Similarity

Type 4, or semantic cloning, occurs when two or more code segments perform the same function but are implemented by different means. This type is the most complex to detect because it requires a deep understanding of what the code is supposed to do, rather than how it is written. Semantic cloning can often go unnoticed without careful analysis and functional testing.

## 5 Similarity Calculation Methods

### 5.1 Chroma

We use Chroma to exploit its advanced capabilities in processing programming languages via embeddings. Chroma leverages the all-MiniLM-L6-v2 language model to generate dense vector representations of code segments. This method begins by instantiating Chroma's default embedding feature, designed to transform text into embeddings that capture deep semantic features of the source code.[4] Here is how the process work in the final application to search the most similar chunks to the users codes

#### 5.1.1 Fill The Database

The goal here is to fill the database correclty. The First step is to consider that we have in our hands a lot of C files (We treats only this langage in this work). We'll fill this databse by chunking all the file into chunks. These chunks represents all the functions (search with a simple regex for the moment) of all the files that you got and that we put into the Chroma database.

#### 5.1.2 Get the Right Chunk in the Database

When you want to search similars chunks to your code in the database, the code you give is chunked by his turn. When it is chunked, all the chunks of your file will be compare to all the chunks of the database by queriering Chroma that will return the most similar chunk (In function of the number of responses you want, if you want more than one only return chunks, you can change it in the final application see part '9 Final Application')

#### 5.1.3 Analyse the Results Chunks

For each pair of code fragments analyzed, code1 and code2, we ask all-MiniLM-L6-v2 language model to obtain their corresponding vectors. These vectors vary in length and scale, which requires a normalization step. We calculate the Euclidean norm of each vector, which measures its length in vector space. By dividing each vector by its norm, we obtain normalized embeddings that all have the same unit length.

#### 5.1.4 Compare Results

The similarity is then obtained by calculating the Gram matrix of these vectors. The result is a value between 0 and 1, where 1 means perfect similarity and 0 means total dissimilarity. To make this score more intuitive, we multiply it by 100 and round it to one decimal place, thus producing a percentage. We then compare it to a threshold (80% in general but you can change it in the final application, see '9 Final Application') and this will say if yes or no this chunk will be a reponse of the application

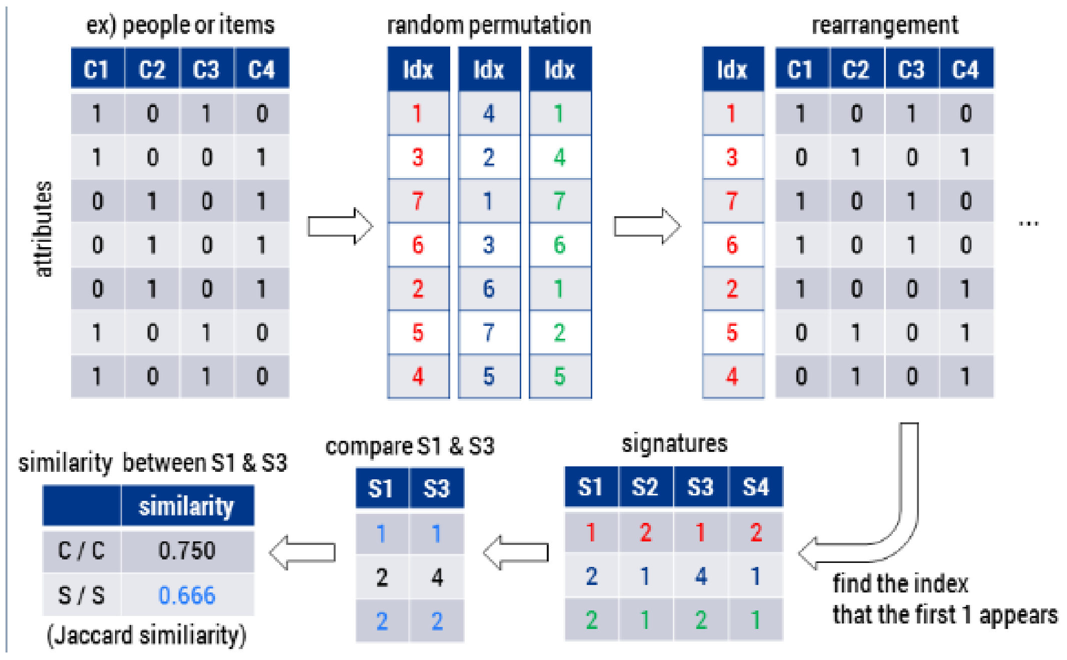
## 5.2 Other Algoritmh

In addition to the Chroma embedding techniques discussed previously, other algorithms like Minhash and Simhash are important. They offer different approaches to measuring similarity between segments of text or code, often used because of their effectiveness in handling large datasets.[5]

### 5.2.1 Minhash

Minhash is a probabilistic algorithm used to estimate the similarity of datasets. It is particularly effective at identifying similarity between data sets, such as text documents or code files, by comparing the "shingles" (subsets) they comprise. In the context of code duplication, Minhash involves breaking code into a collection of small chunks or tokens and creating a "signature" for each document using the minimum hash values produced from those chunks.

The main advantage of Minhash lies in its ability to quickly estimate Jaccard similarity between documents. Jaccard similarity is a statistical measure that calculates the size of the intersection divided by the size of the union of the sample sets. By comparing these Minhash signatures rather than the entire data set, we can effectively determine which code segments are likely to be duplicated without a direct and exhaustive comparison of each element in the data set. This feature makes Minhash particularly suitable for preliminary scans in large code bases where performance and speed are critical.



ex) people or items

| C1 | C2 | C3 | C4 |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |

random permutation

| Idx | Idx | Idx |
|---|---|---|
| 1 | 4 | 1 |
| 3 | 2 | 4 |
| 7 | 1 | 7 |
| 6 | 3 | 6 |
| 2 | 6 | 1 |
| 5 | 7 | 2 |
| 4 | 5 | 5 |

rearrangement

| Idx | C1 | C2 | C3 | C4 |
|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 0 | 1 |
| 7 | 1 | 0 | 1 | 0 |
| 6 | 1 | 0 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 |
| 5 | 0 | 1 | 0 | 1 |
| 4 | 0 | 1 | 0 | 1 |

signatures

| S1 | S2 | S3 | S4 |
|---|---|---|---|
| 1 | 2 | 1 | 2 |
| 2 | 1 | 4 | 1 |
| 2 | 1 | 2 | 1 |

compare S1 & S3

| S1 | S3 |
|---|---|
| 1 | 1 |
| 2 | 4 |
| 2 | 2 |

similarity between S1 & S3

| | similarity |
|---|---|
| C / C | 0.750 |
| S / S | 0.666 |

(Jaccard similiarity)

Figure 1: Representation of the MinHash algorithm

### 5.2.2 Simhash

Simhash is another commonly used algorithm for near-duplicate detection. It generates a fingerprint for each document by processing features (such as tokens or syntactic elements in the code) into a binary hash representation. Each feature influences the final hash by toggling certain bits, with the degree of influence weighted by the importance of the feature. The resulting hash is a fixed-length binary string where similar documents produce similar hash values.

The main utility of Simhash in detecting code duplication is its effectiveness in comparing large numbers of documents. Documents are considered similar if their Simhash fingerprints are within a certain Hamming distance from each other—the Hamming distance being the number of positions at which the corresponding bits differ. Simhash allows for quick comparisons because only hash values need to be compared, rather than entire documents.

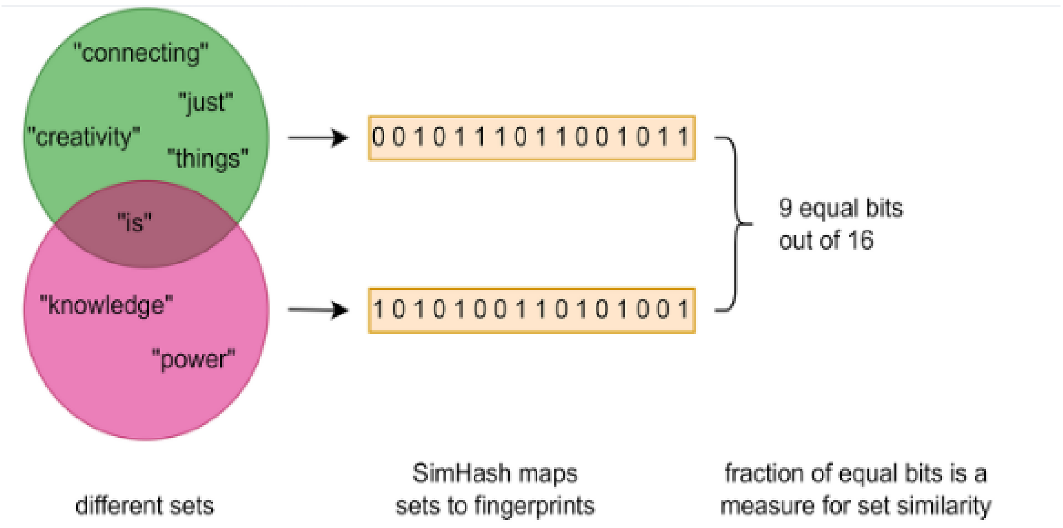


Figure 2: Representation of the SimHash algorithm

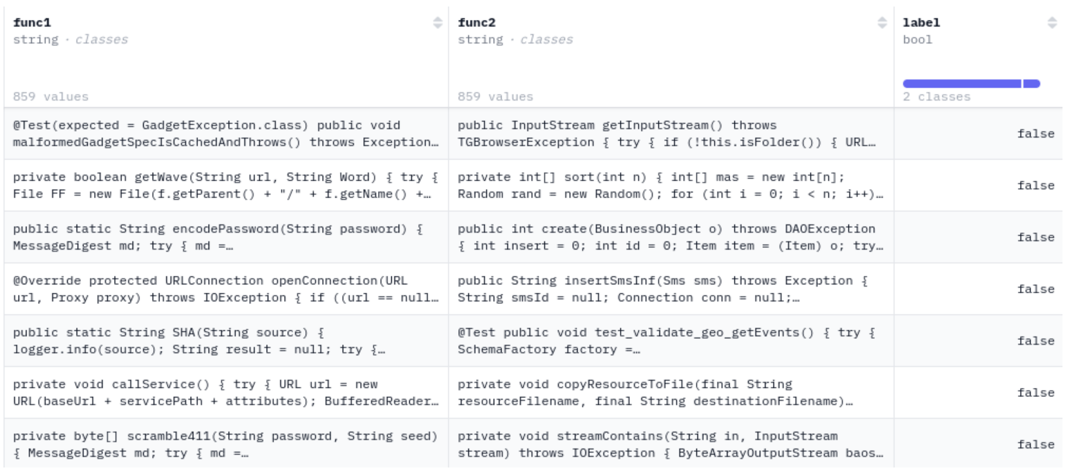
### 5.2.3 Partial Conclusion

These algorithms will be very effective in detecting type I or II cloned code due to their speed in processing large data sets. However, they will have more difficulty with a semantic resemblance between two code snippets that comes from the textual comparison of these algorithms. We use them and compare them to a threshold in the same way than the Chroma similarity calculation method we discuss just before

# 6 Accuracy

## 6.1 BCB

We used the BigCloneBench (BCB) dataset from CodeXGLUE [6], designed for code clone detection via a binary classification task. This involves determining whether two Java methods are semantically equivalent (label '1') or not (label '0'). However, using this dataset presented challenges in evaluating the effectiveness of our methods and algorithms, particularly with our approach based on Chroma embeddings.



| func1 | func2 | label |
|---|---|---|
| string · *classes* | string · *classes* | bool |
| 859 values | 859 values | 2 classes |
| @Test(expected = GadgetException.class) public void malformedGadgetSpecIsCachedAndThrows() throws Exception… | public InputStream getInputStream() throws TGBrowserException { try { if (!this.isFolder()) { URL… | false |
| private boolean getWave(String url, String Word) { try { File FF = new File(f.getParent() + "/" + f.getName() +… | private int[] sort(int n) { int[] mas = new int[n]; Random rand = new Random(); for (int i = 0; i < n; i++)… | false |
| public static String encodePassword(String password) { MessageDigest md; try { md =… | public int create(BusinessObject o) throws DAOException { int insert = 0; int id = 0; Item item = (Item) o; try… | false |
| @Override protected URLConnection openConnection(URL url, Proxy proxy) throws IOException { if ((url == null… | public String insertSmsInf(Sms sms) throws Exception { String smsId = null; Connection conn = null;… | false |
| public static String SHA(String source) { logger.info(source); String result = null; try {… | @Test public void test_validate_geo_getEvents() { try { SchemaFactory factory =… | false |
| private void callService() { try { URL url = new URL(baseUrl + servicePath + attributes); BufferedReader… | private void copyResourceToFile(final String resourceFilename, final String destinationFilename)… | false |
| private byte[] scramble411(String password, String seed) { MessageDigest md; try { md =… | private void streamContains(String in, InputStream stream) throws IOException { ByteArrayOutputStream baos… | false |

Figure 3: Example data from the Big Clone Bench dataset

The main problem encountered was the correlation between the similarity scores generated by Chroma and the binary labels in the dataset. For example, in cases where Chroma gave a high similarity score, such as 90%, and the corresponding label was 'true' (equivalent to '1'), this seemed to validate the effectiveness of the algorithm. However, there were also many cases where despite a similarity score of 55% reported by Chroma, the label was still 'true'. These imprecise results made it difficult to draw reliable conclusions about the actual performance of the chroma, simhash, and minhash algorithms.

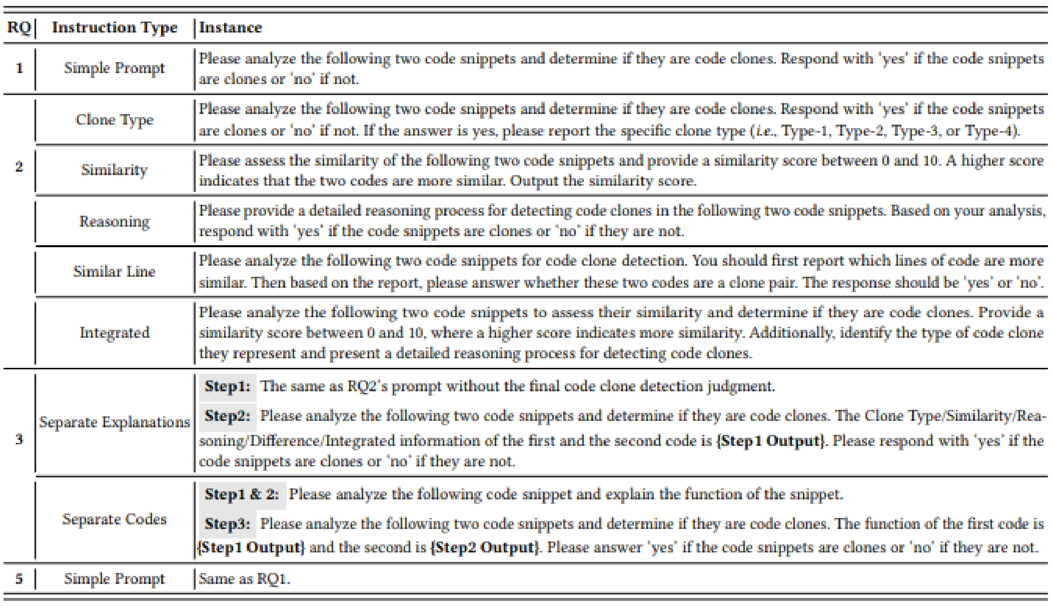
Consequently, this method of evaluation using the BCB dataset was inadequate for our research. The types of clones detected are not explicitly stated in the data, which adds uncertainty : a similarity score of 55% may indicate less direct semantic equivalence (such as type 3 or 4 clones), which our algorithms assume difficult to detect. Furthermore, the lack of differences between different types of clones in BCB labels prevents us from better understanding how our detection methods work. BCB's solution to calculate accuracy was therefore ruled out.

For these reasons, we decided to move towards other methods for the validation of our code duplication detection approaches, looking for sources that offer better granularity and more detailed information on the types of clones present in the data. This adaptation is crucial to refine our algorithms and to guarantee a precise and meaningful evaluation of their performance.

## 6.2 LLM Based

After attempting to use HuggingFace's BCB dataset. The idea came to use LLMs to create a reference answer. We then seek here to determine specific prompts to make the most precise requests to LLMs in order to have correct and error-free responses.

We start by creating prompts for the LLM which will compare two code segments. For example, a typical prompt might be: "Examine the following two functions. Determine whether or not they perform similar tasks, and indicate whether they can be considered clones.". In this case the prompts which were used in our case are the prompts taken from another research document cited below:



| RQ | Instruction Type | Instance |
|---|---|---|
| 1 | Simple Prompt | Please analyze the following two code snippets and determine if they are code clones. Respond with 'yes' if the code snippets are clones or 'no' if not. |
| 2 | Clone Type | Please analyze the following two code snippets and determine if they are code clones. Respond with 'yes' if the code snippets are clones or 'no' if not. If the answer is yes, please report the specific clone type (*i.e.*, Type-1, Type-2, Type-3, or Type-4). |
|  | Similarity | Please assess the similarity of the following two code snippets and provide a similarity score between 0 and 10. A higher score indicates that the two codes are more similar. Output the similarity score. |
|  | Reasoning | Please provide a detailed reasoning process for detecting code clones in the following two code snippets. Based on your analysis, respond with 'yes' if the code snippets are clones or 'no' if they are not. |
|  | Similar Line | Please analyze the following two code snippets for code clone detection. You should first report which lines of code are more similar. Then based on the report, please answer whether these two codes are a clone pair. The response should be 'yes' or 'no'. |
|  | Integrated | Please analyze the following two code snippets to assess their similarity and determine if they are code clones. Provide a similarity score between 0 and 10, where a higher score indicates more similarity. Additionally, identify the type of code clone they represent and present a detailed reasoning process for detecting code clones. |
| 3 | Separate Explanations | **Step1:** The same as RQ2's prompt without the final code clone detection judgment. **Step2:** Please analyze the following two code snippets and determine if they are code clones. The Clone Type/Similarity/Reasoning/Difference/Integrated information of the first and the second code is **{Step1 Output}**. Please respond with 'yes' if the code snippets are clones or 'no' if they are not. |
|  | Separate Codes | **Step1 & 2:** Please analyze the following code snippet and explain the function of the snippet. **Step3:** Please analyze the following two code snippets and determine if they are code clones. The function of the first code is **{Step1 Output}** and the second is **{Step2 Output}**. Please answer 'yes' if the code snippets are clones or 'no' if they are not. |
| 5 | Simple Prompt | Same as RQ1. |

Figure 4: Prompts tirés du document nommé en référence [7]

After receiving the prompt, the LLM generates a response which will give an evaluation of the similarity between the segments. This response is then used to create a binary label - cloned or not cloned. This label is compared with the ground truth label, allowing for example to calculate the F1 score.

However, this method also has its limits. The quality of LLM responses may vary depending on the wording of prompts and the model's ability to correctly interpret complex code snippets.

# 7 Prompts

The interest of this section is to present prompting methods which allow LLMs to give more precise and efficient answers. The two methods that will be presented here are the 'Chain of Thought' or the flow of thought and finally the RISEN prompt which is a type of advanced prompt.[8]

## 7.1 Chain of thought

The "Thought Chain" is a way of writing prompts that involves prompting the model to generate an explanatory sequence of its reasoning steps before providing a final response. This process mimics the way a human thinks about a problem, breaking complex tasks into more manageable intermediate steps.

This technique can enormously transform the way LLMs respond to queries. By using structured prompts that encourage step-by-step thinking, the model can explore code features more methodically, enabling analysis that goes beyond simple text recognition.

To detect duplication, a "Chain of Thought" prompt might look like this:
- Examine each extract: Identify main functions, loops and conditions.
- Compare structures and logics: See if they accomplish the same tasks in the same way.
- Determine whether extracts are cloned based on their functional similarities and differences.

At each step, the model generates explanations that help understand the reasoning behind comparing two code fragments.
This has several advantages :

1. Transparency : Each step of the reasoning is explicit, making the model's conclusions more understandable.
2. Reliability : By breaking down the problem, the model can handle more subtle and complex duplication cases with greater accuracy.

We will therefore use this process to create prompts like the example above in order to refine our query responses.

## 7.2 RISEN

The RISEN prompt is a method designed to structure and break down complex tasks into more manageable components. It is broken down into 5 components: R for Role, I for Instruction, S for Steps, E for End goal and N for Narrowing. Here is a detailed explanation of each component of the RISEN prompts. (An example of a RISEN prompt will be given a little later in the document in section '8 - LLMs Post analysis')

### 7.2.1 R - Role

This component defines the role that the user wants the LLM to adopt to accomplish the task. This can vary depending on the needs, for example, the AI can be considered a research assistant, a content writer, a data analyst... Clear definition of the role helps to frame the objectives of the task at hand.

### 7.2.2 I - Instructions

The instructions detail the main task that the LLM is intended to perform. It is a precise description of what the user expects in terms of response. Instructions must be clear and direct to avoid any ambiguity during execution

### 7.2.3 S - Steps

This section describes the steps the LLM must follow to complete the instruction. Breaking it down into stages helps structure the process. Each step should be numbered and contain concise actions.

### 7.2.4 E - End goal

The end goal sets the goal. It's about defining what the user hopes to have as an answer. The objective must be specific to effectively evaluate the success of the task.

### 7.2.5 N - Narrowing

The last component concerns constraints to be respected in the LLM response. For example a length limit, or specific requirements such as in our case asking for a response with a single unique word which will facilitate automated information retrieval.

## 8 LLMs Post analysis

The interest of this section is to explain how the LLMs could be integrated into the final application. The integration of LLM responses to advanced prompts in our application offers the possibility of more detailed analyses. For example, being able to directly ask what the type of clone is or what the differences are in two extracts. This supports Chroma similarity calculation and makes data processing even more efficient. Below are the two prompts that were created to respond to their problem

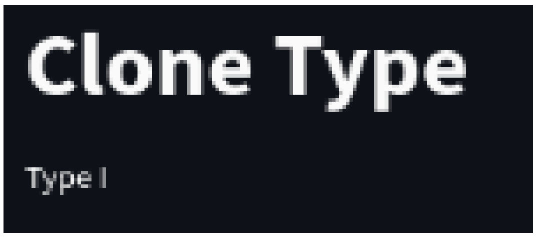
### 8.1 Clone Type Identification

The following RISEN prompt was created to determine the clone type associated with two code snippets:

- Role: Code Review Assistant
- Main Task: Analyze and report the type of clone between two provided code snippets.
- Steps to complete task:
    1. Read both code snippets. Carefully examine the two pieces of code provided below.
    2. Identify differences. Compare each line and identify any discrepancies between the two snippets.
    3. Provide a conclusion

- Goal: The goal is to provide a simple analysis of the type of code cloned.
- Constraints:
    1. Only focus on syntactic and logical differences.
    2. Ignore comments and formatting differences unless they impact the code's functionality.
    3. Answer ONLY by these words : ('Type I', 'Type II', 'Type III', 'Type IV') by chosing the right one

This prompt instructs the LLM to categorize clones into types based on specific differences. We can note a constraint in the number of words to only recover the type of the clone and not a detailed analysis for greater clarity.


Clone Type

Type I

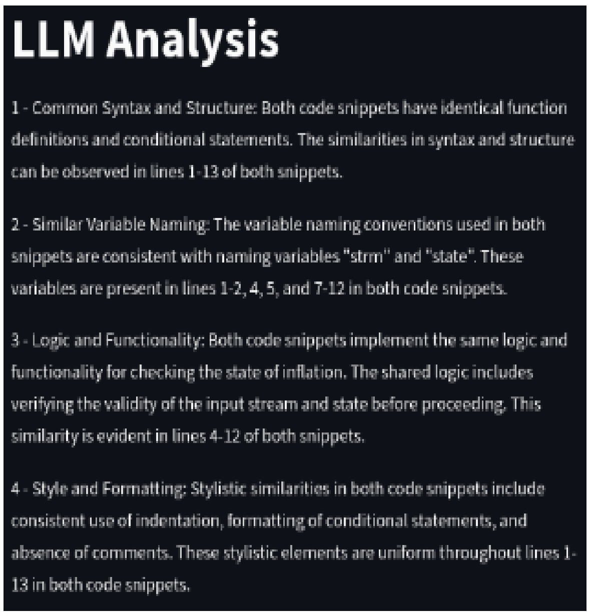
Figure 5: Example answer on the final application

## 8.2 Similarities Analysis

Consider the following RISEN prompt, created for more detailed similarity analysis between two code snippets:

- Role: You are an AI programmed to analyze and compare software code, adept at pinpointing both differences and similarities in code structures.
- Main Task: Perform a detailed analysis of two provided code snippets, focusing on identifying and explaining their similarities, while clearly indicating the specific locations within the code where these similarities are found.
- Steps to complete the task:
    1. Identify Common Syntax and Structure: Review both code snippets for similar syntax and structural patterns. Clearly describe each similarity and specify the line numbers or sections where these similarities occur.
    2. Check for Similar Variable Naming: Analyze the variable naming conventions used in both snippets. Note any similarities in variable names or naming patterns, and indicate exactly where these are used within each snippet.
    3. Assess Logic and Functionality: Look for similarities in the logic and functionality that both snippets are designed to perform. Explain any shared logic or functional aspects, and point out where in the code these parallels are evident.
    4. Evaluate Style and Formatting: Identify any stylistic similarities, including formatting choices, comments, and code arrangement. Discuss how these stylistic elements are similar and highlight the specific parts of the code where these styles match.
- Goal: The goal is to provide a detailed examination of the similarities between the two code snippets, with specific references to the parts of the code where these similarities are found. This should include insights into syntax, structure, variable naming, logic, functionality, and style.
- Constraints: Each identified similarity should be described in one or two sentences, with explicit reference to the exact location in the code. Aim for a comprehensive yet concise analysis, ensuring clarity and precision in locating each point of comparison.

The LLM must identify similarities and locate precisely where they occur in the code which can be defined by 'LLM sourcing'. This answer does indeed provide a clearer and more detailed analysis of why two codes are similar.



**LLM Analysis**

1 - Common Syntax and Structure: Both code snippets have identical function definitions and conditional statements. The similarities in syntax and structure can be observed in lines 1-13 of both snippets.

2 - Similar Variable Naming: The variable naming conventions used in both snippets are consistent with naming variables "strm" and "state". These variables are present in lines 1-2, 4, 5, and 7-12 in both code snippets.

3 - Logic and Functionality: Both code snippets implement the same logic and functionality for checking the state of inflation. The shared logic includes verifying the validity of the input stream and state before proceeding. This similarity is evident in lines 4-12 of both snippets.

4 - Style and Formatting: Stylistic similarities in both code snippets include consistent use of indentation, formatting of conditional statements, and absence of comments. These stylistic elements are uniform throughout lines 1-13 in both code snippets.

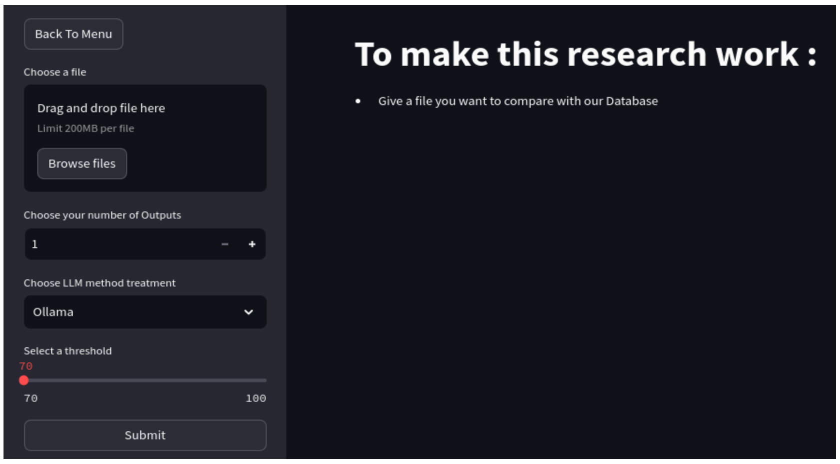
Figure 6: Example answer on the final application

# 9 Final Application

Our application, developed with Streamlit, is designed to facilitate code analysis and duplication detection through three main functionalities: comparing two folders, comparing two files, and searching for specific code. The functionality that interests us here is 'Search Your Code' because it is the one that addresses the subject of CVEs and the search for duplication in a large database. Below are the details of each menu

## 9.1 Search Your Code Menu

The application's flagship functionality, Search your code, addresses the issue of CVEs (Common Vulnerabilities and Exposures). It allows users to upload a code file via drag and drop, enter the number of desired results through an input field, and set a similarity threshold via a slider. This part of the application is designed to search for matches or similarities in a code database using Chroma's, SimHash's and MinHash's similarity scores to determine duplications.



Choose your number of Outputs: 1

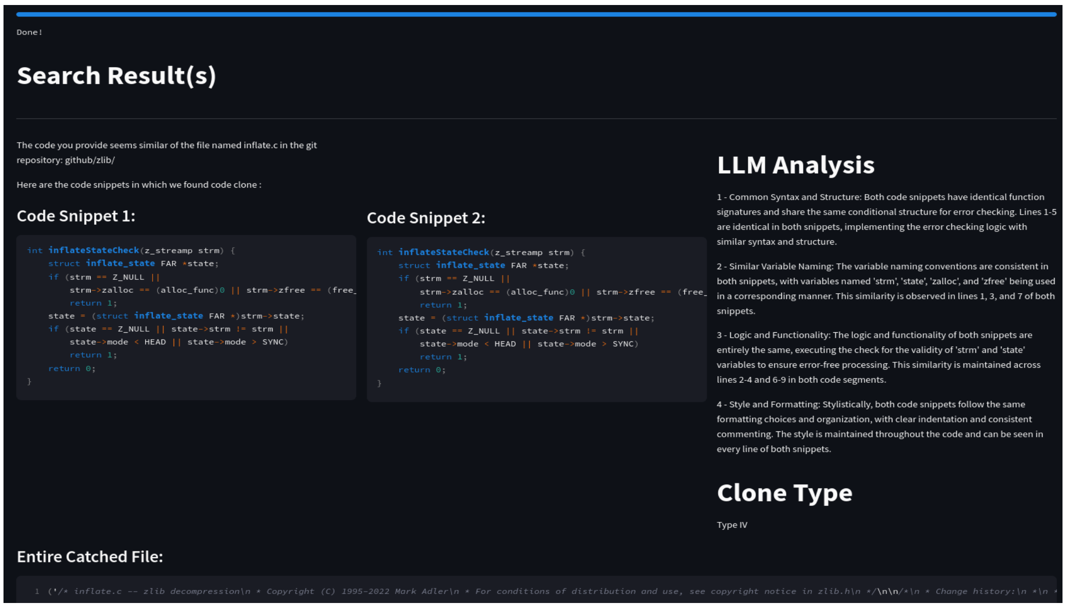
Choose LLM method treatment: Ollama

Select a threshold: 70 (range 70 – 100)

**To make this research work :**

- Give a file you want to compare with our Database

Figure 7: Image of the application on the "Search Your Code" menu



Done!

**Search Result(s)**

The code you provide seems similar of the file named inflate.c in the git repository: github/zlib/

Here are the code snippets in which we found code clone :

**Code Snippet 1:**

```
int inflateStateCheck(z_streamp strm) {
    struct inflate_state FAR *state;
    if (strm == Z_NULL ||
        strm->zalloc == (alloc_func)0 || strm->zfree == (free_
        return 1;
    state = (struct inflate_state FAR *)strm->state;
    if (state == Z_NULL || state->strm != strm ||
        state->mode < HEAD || state->mode > SYNC)
        return 1;
    return 0;
}
```

**Code Snippet 2:**

```
int inflateStateCheck(z_streamp strm) {
    struct inflate_state FAR *state;
    if (strm == Z_NULL ||
        strm->zalloc == (alloc_func)0 || strm->zfree == (free_
        return 1;
    state = (struct inflate_state FAR *)strm->state;
    if (state == Z_NULL || state->strm != strm ||
        state->mode < HEAD || state->mode > SYNC)
        return 1;
    return 0;
}
```

**LLM Analysis**

1 - Common Syntax and Structure: Both code snippets have identical function signatures and share the same conditional structure for error checking. Lines 1-5 are identical in both snippets, implementing the error checking logic with similar syntax and structure.

2 - Similar Variable Naming: The variable naming conventions are consistent in both snippets, with variables named 'strm', 'state', 'zalloc', and 'zfree' being used in a corresponding manner. This similarity is observed in lines 1, 3, and 7 of both snippets.

3 - Logic and Functionality: The logic and functionality of both snippets are entirely the same, executing the check for the validity of 'strm' and 'state' variables to ensure error-free processing. This similarity is maintained across lines 2-4 and 6-9 in both code segments.

4 - Style and Formatting: Stylistically, both code snippets follow the same formatting choices and organization, with clear indentation and consistent commenting. The style is maintained throughout the code and can be seen in every line of both snippets.

**Clone Type**

Type IV

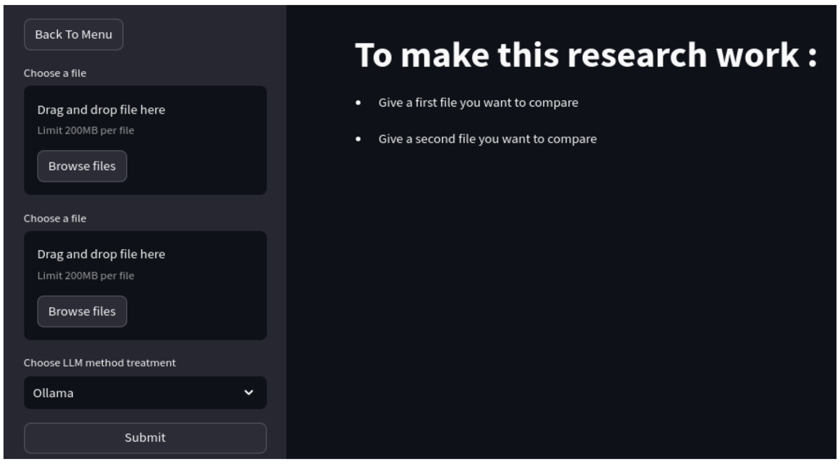
**Entire Catched File:**

```
('/* inflate.c -- zlib decompression\n * Copyright (C) 1995-2022 Mark Adler\n * For conditions of distribution and use, see copyright notice in zlib.h\n */\n\n/*\n * Change history:\n *\n *
```

Figure 8: Example answer on the final application on the "Search Your Code" menu

## 9.2 Compare Files Menu

The Compare Two Folders feature allows users to compare two entire projects. This menu is particularly useful for analyzing and comparing projects that have similar goals but are developed independently. The main objective here was to test the duplication detection algorithms on smaller projects and to be able to draw a two-resemblance score between two files. This allows us to observe how algorithms identify similarities in real-world scenarios where entire projects could be cloned or heavily inspired by each other.



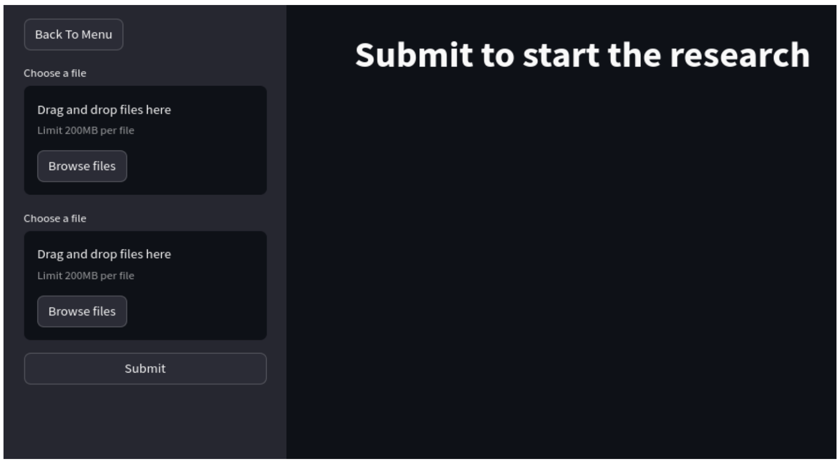
To make this research work :

- Give a first file you want to compare
- Give a second file you want to compare

Figure 9: Image of the application on the "Compare Files" menu

## 9.3 Compare Folders Menu

The Compare Two Files menu is designed for more granular analyses. This feature allows users to specifically select two files and subject them to detailed code duplication analysis. This menu was created to primarily test the "chunking" process. Additionally it helps to understand what the algorithm detects as duplication. This feature was essential to fine-tune the accuracy of chunking and to ensure that the algorithm worked efficiently with a good level of detail.



Submit to start the research

Figure 10: Image of the application on the "Compare Files" menu

## 10 Conclusion

The multitude of code duplication in software development poses security challenges. This work addressed these issues by developing a methodology to detect codes cloned from CVEs. Using advanced tools like Chroma for embeddings and Streamlit for interactive analysis, we have implemented an application that makes it easier to detect duplicationsThe developed features allow users to most effectively detect code reuse. By identifying potentially vulnerable areas of code, our application helps prevent the spread of security vulnerabilities based on precise analysis.

In the longer term, we plan to expand our database to cover more cloned code that could potentially be affected by CVEs. There could also be modifications and improvements to be made in the chunking part so that better results emerge.

In summary, this report marks an important step towards safer development practices. The web application developed here paves the way for future advancements in software security management, ensuring that developers can not only detect but also effectively rectify vulnerabilities in a rapidly changing development environment.

## 11 Acknowledgments

I express my deepest gratitude to Julien Perez and Idir Benouaret for their invaluable guidance and support throughout my research journey. Their expertise profoundly enriched this work, providing both technical insights and methodological rigor that were essential for the success of this project. I am immensely thankful for their mentorship and the opportunities they have offered me, which have been instrumental in my academic development.

## 12 Références

1 - Httpcs 2024 : CVEs definitions and explanations.
https://www.httpcs.com

2 - Chroma 2024 : An AI-native vector database for embedding management. Description of how Chroma facilitates the development and implementation of LLMs in practical applications.
https://www.trychroma.com/

3 - Streamlit 2023 : The fastest way to build and share data applications. Overview and documentation on using Streamlit for creating interactive data applications.
https://streamlit.io/

4 - Chroma Embedding Documentation 2024 : lightweight wrappers around popular embedding providers
https://docs.trychroma.com/guides/embeddings

5 - Github repository that talk about text deduplication including Minhash and Simhash algorithms.
https://github.com/ChenghaoMou/text-dedup

6 - CodeXGLUE 2023 BigCloneBench: A dataset on HuggingFace that includes code cloned with labels thta indicates if the two code snippets are cloned or not
https://huggingface.co/big-clone-bench

7 - Capability of Large Language Models on Code Clone Detection 2023 : A survey that talk about evaluation of LLMs for clone detection, covering different clone types, languages, and prompts
https://arxiv.org/pdf/2308.01191

8 - prompt frameworks 2023 : Article about how to write evolved prompts
https://www.thepromptwarrior.com

9 - PyTorch 2023 PyTorch-Transformers: Library providing state-of-the-art implementations for various transformer models, making it easy to train and use advanced NLP models in research and industrial application projects.
https://pytorch.org/

10 - BigCode Project. 2023. The Stack v2: a multi-language source code dataset from various open-source repositories, intended for training and evalu- ating code processing models, hosted on Hugging Face.
https://huggingface.co/datasets/bigcode/the-stack-v2

11 - Microsoft 2023 Phi-1 5 (Hugging Face): A Specialized Natural Language Processing Model for Coding Tasks. Designed for code generation, infilling, and code understanding while remaining compact and efficient.
https://huggingface.co/microsoft/phi-1.5