Binary–Source Code Matching Based on Decompilation Techniques and Graph Analysis

Ghader Aljebreen[1], Reem Alnanih[2], Fathy Eassa[3], Maher Khemakhem[4], Kamal Jambi[5], Muhammed Usman Ashraf[6]
Department of Computer Science-Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia[1, 2, 3, 4, 5]
Software Engineering and Distributed System Research Group, King Abdulaziz University, Jeddah 21589, Saudi Arabia[2, 3, 4, 5, 6]
Department of Computer Science, Government College Women University, Sialkot, Pakistan[6]

***Abstract*—Recent approaches to binary–source code matching often operate at the intermediate representation (IR) level, with some applying the matching process at the binary level by compiling the source code to binary and then matching it directly with the binary code. Others, though less common, perform matching at the decompiler-generated pseudo-code level by first decompiling the binary code into pseudo-code and then comparing it with the source code. However, all these approaches are limited by the loss of semantic information in the original source code and the introduction of noise during compilation and decompilation, making accurate matching challenging and often requiring specialized expertise. To address these limitations, this study introduces a system for binary–source code matching based on decompilation techniques and Graph analysis (BSMDG) that matches binary code with source code at the source code level. Our method utilizes the Ghidra decompiler in conjunction with a custom-built transpiler to reconstruct high-level C++ source code from binary executables. Subsequently, call graphs (CGs) and control flow graphs (CFGs) are generated for both the original and translated code to evaluate their structural and semantic similarities. To evaluate our system, we used a curated dataset of C++ source code and corresponding binary files collected from the AtCoder website for training and testing. Additionally, a case study was conducted using the widely recognized POJ-104 benchmark dataset to assess the system's generalizability. The results demonstrate the effectiveness of combining decompilation with graph-based analysis, with our system achieving 90% accuracy on POJ-104, highlighting its potential in code clone detection, vulnerability identification, and reverse engineering tasks.**

***Keywords*—Binary–source code matching; call graphs; code clone detection; control flow graphs; decompiler**

I. Introduction

Since free software has become more popular, companies have adopted it widely and integrated it into closed-source projects. In addition to its economic appeal, its popularity is primarily driven by its convenience and flexibility for customization. Therefore, it is common for code to be modified, adapted, or reused before being redistributed or republished.

The practice of reusing or cloning code has become widespread. The reuse of code snippets, however, can introduce other risks besides license violations, including potential harm or security flaws that have already been addressed in the original code [1].

There have been numerous tools created in recent years that handle clone detection at a lower level than source code, including Java Bytecode [2] and LLVM IR [3]. Several techniques have been proposed to detect code clones, at binary–binary [4], source–source [5], or binary–source code level [6, 7]. Token-based [8], tree-based [9], and graph-based methods are among these techniques [10]. Combining two or more techniques can also be achieved, as in [11], which combines tree-based and graph-based methods. Binary–source code matching is used in many security software engineering activities, including malware detection [12], vulnerability searches [10], reverse engineering [13], and code clone or similarity detection [10] , etc. Using semantic features extracted from binary and source code, binary–source code matching calculates the semantic similarity between binary and source code [14].

Some previous studies in the field of binary-source code matching and clone detection have adopted the approach of matching source code with binary code or decompiler-generated pseudo-code [15]. However, most efforts have concentrated on binary-source code matching and clone detection at the intermediate representation (IR) level. Nevertheless, a recent study [16] identified significant disparities between the intermediate representation (IR) obtained through the decompilation of binary code and the IR derived from the corresponding source code. These disparities can hinder the learning process, as the decompiled IR is often difficult to comprehend. Consequently, all the aforementioned approaches are constrained by the loss of the rich semantic information inherent in the original source code or by the introduction of noise during compilation and decompilation, making accurate matching challenging and often requiring specialized expertise. This limitation makes it difficult to detect and capture clones based on semantic similarities rather than solely on structural similarities. Conversely, binary–source code matching at the source code level offers advantages in terms of language familiarity, compatibility with existing source code analysis tools, contextual understanding, maintainability, and developer productivity, making it a valuable approach to code matching and clone detection in real-world software systems.

Hence, the main goal of this study is to match a binary code (target) with a source code (reference) at the source code level, where the binary code may have been compiled on different machines or compilers. By leveraging the source code

(reference), we can incorporate more semantic information, such as variable names and types, improving the accuracy of the matching process. This approach not only helps in identifying whether a corresponding binary code is included in a binary file—thus warning against potential vulnerabilities [7]—but also dramatically enhances the performance of binary-source code matching and clone detection compared to binary–binary code matching or decompiling binary (target) to pseudo-code and matching it with the original source code (reference).

In binary–source code matching, the main challenge is to bridge the semantic gap between the low-level machine code and high-level programming languages [14]. Binary files are obtained through the compilation process, and analyzing their similarity with the source code typically requires decompilation. Decompilation is the process of reconstructing high-level source code from a binary file [17].

For this purpose, we utilize decompilation techniques, specifically the Ghidra decompiler [18], along with a custom transpiler or translator to convert binary code compiled from C++ source code back into its corresponding high-level source code in C++, which serves as our target high-level language. A graph similarity analysis is then performed on both the original and generated C++ source codes (code pairs). By generating graph representations for both code snippets, specifically call graphs (CGs) and control flow graphs (CFGs), and measuring the similarity between these graphs using the weighted Jaccard index, this approach can effectively identify potential matching code pairs. We use only statically extracted code features (CGs and CFGs) in our binary–source code matching system. Due to this, BSMDG is easily scalable to programs with sizes in the hundreds of thousands and requires minimal RAM resources. The goal of BSMDG is to detect similarities between source code and binary code, which are syntactically different. Thus, it computes similarity based on semantic code features such as function declarations.

This approach goes beyond traditional methods that rely solely on textual or token-based comparisons, as it takes into account the underlying program structure and the relationships between elements. By representing code snippets as graphs, it becomes possible to capture complex dependencies and control flow within the code.

Overall, the contributions of this study are as follows:

- To the best of our knowledge, we have developed the first translator or transpiler (source-to-source compiler) that translates Ghidra's decompiler output (C-like pseudocode) from an input C++ binary file into its corresponding high-level C++ source code. This innovation enables binary–source code matching directly at the source code level (C++), rather than at the binary, IR, or pseudocode levels. Matching at the source code level significantly improves matching accuracy.
- We developed graphs (CGs, CFGs) generator based on the C++ source code generated by the transpiler.
- As function-level binary–source code matching is vital in computer security, we developed a prototype system for function-level binary–source code clone detection based on decompilation techniques and graph similarity at the source code (C++) level, focusing on both semantic and syntactic clones.
- We used the weighted Jaccard index as a similarity measure for graph-based binary-source code matching and clone detection at the source code level.
- We curated a new C++ dataset from Atcoder website [19]. Then, we conducted comprehensive experiments to train and test the proposed approach based on this dataset.
- As a case study, we evaluated the proposed rule-based approach against several baseline AI-based methods that detect C++ code clones at the IR level. These AI-based systems typically require extensive data training to achieve accurate results, often involving large datasets and significant computational resources. We evaluated our approach using the POJ-104 dataset—a widely recognized benchmark in the field of code clone detection—which served as unseen data for our method. Despite the lack of such extensive training, our approach demonstrated superior performance, offering significant time-saving while achieving better accuracy.

The remainder of this study is organized as follows: The literature is reviewed in Section II. In Section III, we describe the proposed materials and methods in detail. Section IV and Section V show the experimental setup and discuss the experimental results, respectively. Section VI presents the case study on clone detection and evaluates our proposed system by comparing it with baseline studies. Section VII illustrates the limitations of the current work. Lastly, Section VIII concludes the study and suggests future work directions.

II. Literature Review

This section covers the literature related to binary–binary, source–source, and binary–source code similarity (matching) and clone detection.

A. Binary–Binary Code Similarity

Binary code similarity approaches date back to 1999. For example, Baker et al. [20] developed a prototype diffing tool called Exediff for compressing differences of executable code. Exediff was one of the first approaches that studied binary code similarity by disassembling raw bytes into instructions and utilizing the code structure.

In the decades that followed Exediff, several binary code similarity approaches were developed. Some of these are highly influential as they extend binary code similarity beyond purely syntactical similarity to encompass semantic similarity as well.

In 2004, Thomas Dullien, also known as Halvar Flake, proposed a graph-based binary code diffing approach [21]. This method involved constructing a call graph isomorphism and aligning functions of different binary program versions. This advancement marked the foundation for the BinDiff binary code diffing plugin for the Interactive DisAssembler (IDA) [22].

During the last decade, binary code similarity has gained popularity, as it has the integration of machine learning and deep learning.

In a recent study [23], the authors presented a novel approach to detect function-level clones in binary code. With their proposed control flow graph (CFG) refinement algorithm, code reuse can be easily tracked, even in binaries compiled for different processor architectures. The CFG refinement algorithm works by extracting various function flows and reconstructing a higher-level structure, leveraging architectural differences and allowing efficient comparison in linear time with structural hashing. The study mentions several limitations and threats to validity. One limitation is that the approach is based on the assumption that the same function will have the same behavior across different architectures, which may not always be true. Another limitation is that the approach may not be effective in detecting clones that have been obfuscated or transformed in some way. Finally, the study acknowledges that the approach may not be suitable for detecting clones in certain architectures, such as ARM, where predication is used as an alternative to branching.

B. Source–Source Code Similarity

In [24], the authors presented a framework for code clone detection at the level of source code using either control flow graphs (CFGs) or PDG (Program Dependency Graph). While effective, the approach's reliance on deep learning requires substantial data and computational resources, impacting its practical utility.

Another study [25] also used deep learning, introducing a novel approach for detecting functional code clones with different structures but matching functionality. The approach combines fusion embedding and fine-grained functionality identification using abstract syntax trees (ASTs) and CFGs. Despite promising results, the fused code representation might not encompass all possible syntax and semantic variations, leading to potential false negatives in clone detection.

As a means of exploiting control and data flow information, the authors of [11] created a graph representation of programs named the flow-augmented abstract syntax tree (FA-AST). The FA-AST was constructed by adding explicit control and data flow edges to the source code's ASTs. Two different types of graph neural networks (GNNs) were then applied to FA-AST to measure code similarity. The authors were the first to use GNNs to detect code clones.

Similar to this, source-code-level exploitation of the data from the CFG and DFG was used in [26]. Program Graphs for Machine Learning (PROGRAML) is a low-level, portable format that leverages machine learning models that may be utilized to carry out challenging downstream tasks. It can be used to offer a unique graph-based program representation. The types and orders of operands and instructions, as well as control, data, and call relationships, are recorded, compiled, and represented using the PROGRAML representation. Learnable models may perform several kinds of program analyses using the general-purpose program representation provided by PROGRAML.

Existing program dependence graph (PDG) generators for C and Java code have limitations as they only support compilable programs, restricting their practical application. Addressing this issue, the authors of [27] introduced CCGraph, a novel code clone detection tool. CCGraph focuses on identifying code clones within PDG-based environments. To achieve this, CCGraph utilizes graph kernels and an approximate graph matching technique. This approach aims to overcome the constraints posed by traditional PDG generators and expand the scope of code clone detection on the Weisfeiler-Lehman (WL) graph kernel. Compared to current state-of-the-art technologies, this approach improves efficiency and finds more semantic clones. However, it necessitates using complete compilable programs as test datasets, constraining the applicability of the PDG-based clone detection approach. Developing a PDG generator capable of handling code segments is recommended to broaden implementation.

Moreover, the authors of [28] investigated the use of CFGs for static analysis in grading programming assignments. The study assesses the degree of similarity between students' codes submissions and teacher reference code through an experiment using a CFG comparison algorithm. The research concludes that CFG comparison is more suited for boosting students with minor errors rather than being employed as the primary scoring algorithm for all submissions. The study solely assesses the CFG structure, neglecting the content of CFG nodes, which could lead to inaccurate scoring.

Another study [29] presented CODE-MVP, a model integrating multiple source code views—plain text, abstract syntax tree (AST), and control or data flow graphs (CFGs or DFGs)—through multi-view contrastive pre-training. The model learns complementary information across these views, augmented by fine-grained type inference during pre-training. Experiments demonstrated CODE-MVP's superiority over state-of-the-art baselines across five datasets and three downstream tasks. However, the exclusion of call graphs limits the capture of essential program behavior aspects, hindering a comprehensive view of functions and their relationships.

C. Binary–Source Code Similarity

Certain studies adopt binary–source code similarity detection techniques to enhance similarity results. These approaches integrate both source code and corresponding binary code to achieve improved accuracy compared to traditional binary–binary, and source–source code similarity methods.

An example is described in [7], which proposed a framework for function-level binary-source code matching that involves extracting semantic features and code literals from both source and binary code, merging them into embeddings, and using triplet loss to learn the relation. The proposed model uses a deep pyramid convolutional neural network (DPCNN) on character-level source code and graph neural network (GNN) models on binary code, as well as integer-LSTM and hierarchical-LSTM for code literals. LSTM stands for long short-term memory, which is a type of recurrent neural network architecture commonly used for processing sequential data such as text. The study also discusses the potential benefits and drawbacks of the proposed model, as well as some

limitations and future research directions. Overall, the proposed model achieves promising results on two datasets and could have practical applications in computer security. However, the model relies on the availability of large-scale source-binary code pairs for training, which may not always be feasible in practice.

In [10], the authors used two steps to identify similarities between source code and binary code. They first generated source code using the provenance of the target binary code. Code similarity was then ranked using a unique graph triplet loss network. The method performs better for syntactic code clones but is less effective against semantic clones.

In [14], a novel approach for cross-language binary–source code matching was introduced, leveraging intermediate representations (IRs). These IRs provide high-level code representation that abstracts away language-specific intricacies. The methodology involves the conversion of both binary and source code into IRs, followed by the utilization of a transformer-based neural network to learn the correlation between these two IRs. The evaluation, conducted on tasks involving cross-language binary–source and source–source code matching, shows the method's superiority compared to other state-of-the-art techniques. However, this approach still requires a number of enhancements, including the need for a larger dataset and large pre-training corpora to overcome the challenges related to cross-language information retrieval.

In few studies, binary-source code matching was conducted at the level of decompiler-generated pseudo-code by first converting binary code into pseudo-code and then comparing it with the source code. For example, a recent study [15] introduces a framework (DBSM) that enhances binary-source function matching by decompiling binaries into pseudo-code using the IDA Pro decompiler and utilizing a self-attention-based siamese network for function comparison. Although this approach outperforms other methods, its limitation lies in its reliance on pseudo-code, which lacks the rich semantic nuances of high-level source code, potentially leading to less accurate results compared to matching at the source code level. Additionally, their evaluation was conducted solely on two self-curated datasets (R0 and R3) without testing against any benchmark dataset, which hinders direct comparison with other baseline studies.

From the aforementioned studies' limitations, we can conclude that the proposed solution should utilize code graphs, namely call graphs (CGs) and control flow graphs (CFGs), to provide more semantic (contextual) information; these are more stable during code transformations (obfuscation resilient). This enhances the detection of semantic clones, which reflects positively on the performance of the target down-stream tasks, such as clone detection and vulnerability analysis. Moreover, some of the previously mentioned works that used AI techniques suffered from limitations, such as depending on existing datasets that were not applied in real operational cases. Furthermore, most research in the field of binary–source code matching and clone detection focuses on implementing the matching process at the IR level, with some studies addressing binary or decompiler-generated pseudo-code. These approaches typically emphasize the structural representation of code, often lacking the rich semantic context present in the original source code. This limitation makes it challenging to capture and detect clones that rely on semantic similarities rather than structural ones. Additionally, the analysis often requires expertise in IR languages. Therefore, to address these gaps, we focus on software reverse engineering and rule-based techniques for binary–source code matching and clone detection at the source code level, specifically in C++ in our study. Detecting matches and clones at the source code level offers benefits in terms of accuracy, interpretability, and direct applicability to the source code that developers interact with.

III. Materials and Methods

The aim of this research is to match or determine the similarity between a given C++ source code (reference) and a binary code compiled from the same or a different C++ source code (target) based on the percentage of binary code functions that match source code functions. High similarity scores indicate that the binary was compiled from the given source code. The proposed system (BSMDG) remains unaffected by minor alterations, including the alteration or elimination of sections in the source code that do not compile (e.g., comments or white space), or changes to variable names, function names, or the order of declarations. We measure the likelihood rather than conclusively determining that the given source code contributed to the binary's compilation, highlighting the nuanced approach needed for code clone detection within security contexts.

Three main steps are taken to accomplish this goal: preprocessing, graph generation, and measuring code similarity. Fig. 1 shows the detailed design of the proposed system. In the preprocessing step, the binary executable is decompiled into C-like pseudocode, which is then translated into C++ source code to facilitate comparison with the original C++ source code (reference). The graph generation step involves creating call graphs (CGs) and control flow graphs (CFGs) for both the original and generated C++ source code, capturing the structural and functional relationships within the code. Finally, in the measuring code similarity step, these graphs are compared using the weighted Jaccard index to quantify the similarity between the source and binary code, providing a nuanced assessment of whether the binary was likely compiled from the given source code. This systematic approach ensures a thorough and reliable evaluation of code similarity.

A. Preprocessing

This step involves decompiling the binary executable into pseudo-code and then translating this pseudo-code into its corresponding C++ source code.

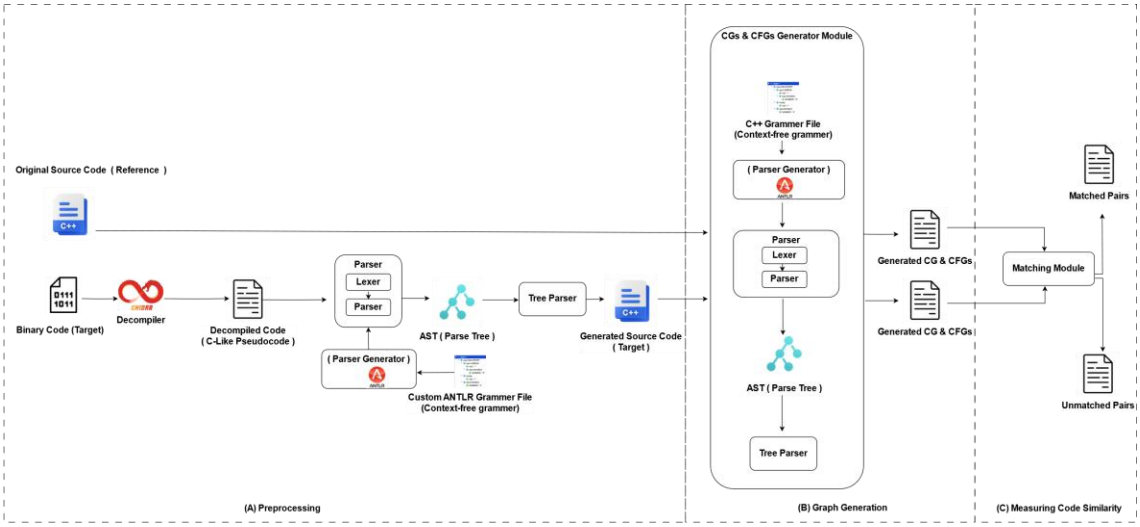

Fig. 1. Detailed design of the proposed system.

1) Decompilation: The binary executable is decompiled using the Ghidra decompiler (version 10.1.2) [18] into C-like pseudo-code. Our focus in the Ghidra decompiler output is primarily on the (main) function, excluding libraries (DLL files) and compiler functions. Analyzing the (main) function in the decompiler output (C-like pseudo-code) is crucial not only for reducing analysis costs but also for providing more accurate similarity results. Since it represents the actual user code, this focus offers valuable insights into the overall functionality and behavior of the binary executable, assisting in debugging and determining the program's purpose.

```
#include <iostream>
using namespace std;

int main() {
    int i, a[9];
    for(i = 0; i < 8; i++)
    {
        cin >> a[i];
    }
    a[8] = 1001;
    int flag = 1;
    for (i = 0; i < 8; i++)
    {
        if(!(a[i] >= 100 && a[i] <= 675 && a[i] % 25 == 0 && a[i] <= a[ i +
 1 ]))
        {
            flag = 0;
            break;
        }
    }
    if ( flag == 0 )
    {
        cout << "No\n";
    }
    else
    {
        cout << "Yes\n";
    }
    return 0;
}

```

Fig. 2. Example C++ source code from atcoder.

Fig. 2 illustrates example C++ source code obtained from the Atcoder website [19]. Fig. 3 shows the C-like pseudo-code generated by Ghidra for the binary file compiled from the C++ source code shown in Fig. 2.

2) Translation: During this step, a dedicated transpiler or translator is used to translate Ghidra decompiler's output (C-like pseudocode) into its corresponding C++ source code. Throughout this study, the term 'translated code' refers to the C++ source code obtained from decompiled binary via the custom transpiler. This translation bridges the gap between binary and high-level code, enabling more accurate comparisons. The transpiler module utilizes ANother Tool for Language Recognition's (ANTLR's) lexer and parser version 4.13.0 [30, 31], which is a parser generator, to tokenize and parse the C-like pseudo-code, based on customized grammar (. g4) files developed by the authors of this study. The grammar files are specifically tailored to meet the unique requirements and specifications of the translation process, ensuring accurate and precise conversion of the C-like pseudo-code into corresponding C++ source code. Fig. 4 displays the output of the translation phase for the C-like pseudo-code example depicted in Fig. 3. In some cases, the translated code may not be fully compilable due to certain syntax errors, which result from the current limitations of the transpiler. These issues include missing type declarations, unmatched brackets, or irregular function signatures that occasionally arise from the decompiled pseudo-code structure. Although these syntax errors do not prevent the generation of call graphs (CGs) and control flow graphs (CFGs), they may introduce minor inaccuracies in the graph structure, such as missing or misrepresented nodes. As a result, these inaccuracies could slightly affect the computed similarity scores, particularly in cases, where structural details play a key role in clone detection. However, based on our empirical observations, the overall impact on semantic similarity is limited, as the primary structural patterns and control flow are still preserved. Due to time constraints, resolving these syntax issues is part of our planned future work to further enhance translation accuracy and improve similarity measurement reliability. Fig. 5 illustrates Algorithm 1 which is a depiction of the translation process. This algorithm is designed to systematically translate the C-like pseudo-code output from Ghidra into its corresponding C++ source code. The algorithm begins by

initializing an ordered collection to store the translated lines (step 1). Then, it iterates through the C-like pseudo-code, line by line (steps 2 and 3), modifying each line to adhere to the syntax of C++ as closely as possible. The modified lines are then added to the ordered collection in the order they appear in the pseudo-code (step 4). Once the translation of all lines is complete, a new source code file is created to store the translated code (step 5). The lines from the ordered collection are then written to the source code file in the same order (step 6), ensuring the translated code maintains the original sequence.

```
undefined8 main(void)
{
    int local_38[10];
    int local_10;
    int local_c;
    for (local_c = 0; local_c < 8; local_c = local_c + 1)
    {
        std::basic_istream<char,
        std::char_traits<char>>::operator>>((basic_istream<char,
            std::char_traits<char>> *)std::cin,local_38 + local_c);
    }
    local_38[8] = 0x3e9;
    local_10 = 1;
    local_c = 0;
    do
    {
        if ( 7 < local_c)
        {
        LAB_0040120f:
            if (local_10 == 0)
            {
                std::operator<<((basic_ostream *)std::cout, "No\n");
            }
            else
            {
                std::operator<<((basic_ostream *)std::cout, "Yes\n");
            }
            return 0;
        }
        if (((local_38[local_c] < 100) || ( 0x2a3 < local_38[local_c])) ||
            (local_38[local_c] % 0x19 != 0)) ||
            (local_38[local_c + 1] < local_38[local_c]))
        {
            local_10 = 0;
            goto LAB_0040120f;
        }
        local_c = local_c + 1;
    } while(true);
}
```

Fig. 3. Example C-like pseudo-code (output of Ghidra's decompiler).

```
int main(void)
{
    int local_38[10];
    int local_10;
    int local_c;

    for(local_c = 0; local_c < 8; local_c = local_c + 1)   {
        std::cin >> local_38 + local_c;
    }
    local_38[8] = 0x3e9;
    local_10 = 1;
    local_c = 0;

    do {
        if ( 7 < local_c)   {
            LAB_0040120f: if (local_10 == 0) {
            std::cout << "No\n";
        }else {
            std::cout << "Yes\n";
        }
        return 0;
        }
    if ((((local_38[local_c] < 100) || (0x2a3 < local_38[local_c])) ||
        (local_38[local_c] % 0x19 != 0)) ||
        (local_38[local_c + 1] < local_38[local_c])) {
        local_10 = 0;
        goto LAB_0040120f;
    }
    local_c = local_c + 1;
    } while(true);
}
```

Fig. 4. Example output of the translation module.

Algorithm 1: Translate the C-like pseudo-code to its corresponding C++ source code

Input: C-like pseudo-code (Ghidra's decompiler output)
Output: C++ source code
1. Initialize an ordered collection to store the translated lines.
2. Traverse the C-like pseudo-code line by line.
3. For each line:
4. Modify the line to match the C++ code syntax as closely as possible, and add the modified line to the ordered collection.
5. Create a new source code file to store the translated code.
6. Write the lines from the ordered collection to the source code file in the same order.

Fig. 5. Algorithm 1: Translate the C-like pseudo-code to its corresponding C++ source code.

B. Graph Generation

Once the binary code is translated into a high-level representation (C++ source code), call graphs (CGs) and control flow graphs (CFGs) are generated for both the original C++ source code (reference) and the generated C++ source code produced by the translation module (target). CGs and CFGs are essential tools in software analysis that capture the relationships and dependencies between different components of the code, facilitating a more comprehensive analysis.

1) Call Graph (CG): A CG is a directed graph that represents calling (caller-callee) relationships between different functions or methods within a program. It captures the flow of control between different functions, providing insights into how the program's components interact with each other.

2) Control Flow Graph (CFG): A CFG is a directed graph that represents the control flow within a function or method. It illustrates the flow of execution through the function, depicting the sequence of statements and the decision points, such as conditional branches or loops, within the function.

The generation of CGs and CFGs in this study is carried out using a proprietary graph generator, developed by the authors of this study, in Java (version JDK17). Off-the-shelf graph generators are unsuitable for this purpose due to the potential presence of syntactical errors in the generated C++ source code from the translation process, rendering it non-compilable. Consequently, we employ a dedicated graph generator to overcome this limitation. To generate CGs and CFGs, ANTLR's lexer and parser are used to tokenize and parse the C++ source code pairs (original and generated) based on the C++ grammar file (CPP14.g4) [32] to produce abstract syntax trees (ASTs), which are then utilized to generate call and control flow graphs. Algorithms 2 and 3, in Fig. 6 and Fig.7, show how CGs and CFGs are generated, respectively.

In clone detection, call graphs represent how functions are called, identifying function-level clones in programs. Using call graphs, you can detect both direct copies and complex clones by analyzing the structure and flow of function calls. Therefore, they are essential for identifying code similarities with a high degree of reliability.

Algorithm 2: Call Graph Generation

Input: C++ source code
Output: Call Graph (CG)
1. Class CallGraphListener //Define a class named CallGraphListener responsible for processing the C++ source code to generate the call graph.
2. Declare root, parent, child as GenericGraphItem //Declare variables root, parent, and child of type GenericGraphItem to represent nodes in the call graph.
3. Declare callGraph as new GenericGraph //Initialize callGraph as a new instance of GenericGraph to store the entire call graph.
4. Procedure enterFunctionDefinition () //Triggered when the parser enters a function definition in the source code.
5. Create new root node with function name and add it to callGraph //Create a new node representing the function being defined and add it to callGraph.
6. Procedure enterStatement() //Triggered whenever the parser encounters a statement within a function.
7. If statement is a cout, cin, function call, or method call //Check if the current statement is relevant (output, input, function call, or method call).
8. Create new childItem with statement name and parent //Create a new node for the statement, associating it with the current parent function.
9. Add child to callGraph //Add the child node to the callGraph, linking it to the parent function.
10. Procedure exitFunctionDefinition() //Triggered when the parser exits a function definition in the source code.
11. Reset parent and child to null //Clear the parent and child variables to indicate the end of the function scope.
12. Procedure generateOutput() //Responsible for generating the final call graph output after processing the source code.
13. Try to generate graphs from callGraph //Attempt to generate textual (.dot file) and visual representations of the call graph.
14. If an exception occurs //Handle any exceptions that might occur during graph generation.
15. Print error message //Print a relevant error message if an error occurs during graph generation.
16. End Class //End of the CallGraphListener class, concluding the call graph generation process.

Fig. 6. Algorithm 2: Call Graph (CG) generation.

Algorithm 2 starts by establishing a class called CallGraphListener (step 1) and initializing three variables (root, parent, and child), all of which are of the type GenericGraphItem (step 2). These variables serve as nodes in the call graph. Additionally, a new instance of the GenericGraph class, callGraph, is created to act as the container for the call graph (step 3). The algorithm proceeds by defining procedures for entering function definitions (step 4), statements (step 6), and exiting function definitions (step 10). These procedures handle the creation of nodes and their connections within the call graph. Steps 6 to 11 of the Call Graph Generation algorithm describe the handling of individual statements within a function and the management of the call graph structure. When the parser encounters a statement within a function (step 6), it checks whether the statement is relevant, such as an output operation (cout), input operation (cin), a function call, or a method call (step 7). If the statement is relevant, a new node (childItem) representing the statement is created and associated with the current parent node, which represents the context or function in which this statement resides (step 8). This new childItem is then added to the callGraph, linking it to the parent node and integrating the statement into the call graph (step 9). When the parser exits a function definition (step 10), the parent and child variables are reset to null, clearing the current function's context and ensuring that subsequent function definitions start with a fresh state (step 11). This process ensures that the call graph accurately reflects the function calls and control flow in the C++ source code. Finally, the algorithm includes a generateOutput procedure (step 12) responsible for generating the desired output from the callGraph. It also incorporates error handling to address any exceptions that may occur during the graph generation process (steps 14 and 15). The algorithm concludes with the termination of the CallGraphListener class (step 16), marking the end of the CG generation process and the encapsulation of all related functionalities within the class.

Algorithm 3: Control Flow Graph Generation

Input: C++ source code
Output: Control Flow Graph (CFG)
1. Class ControlFlowGraphListener // Define a class named ControlFlowGraphListener, responsible for processing the C++ source code to generate the control flow graph.
2. Declare root, parent, child as GenericGraphItem // Declare three variables (root, parent, and child) of type GenericGraphItem. These represent nodes in the control flow graph, with root as the starting node, parent as the current context node, and child as a node created by control statements.
3. Declare flowGraph as new GenericGraph // Initialize flowGraph as an instance of GenericGraph. This data structure will store the entire control flow graph, with nodes representing control statements and edges representing the flow between them.
4. Procedure enterFunctionDefinition () // Triggered when the parser enters a function definition in the source code.
5. Create new root node with name, add as root for the flowgraph // Create a new root node representing the function and add it as the root node in the flowGraph. This marks the starting point of the control flow within the function.
6. Procedure enterControlStatement () // Triggered whenever the parser encounters a control statement (e.g., if, while, for).
7. Create new child with name, parentItem // Create a new node (child) representing the control statement, associating it with the current parentItem (the control context in which this statement occurs).
8. Add child to flowGraph and to parent's subItems if parent exists // Add the child node to the flowGraph and also to the list of subItems under parent (to establish the hierarchical relationship between the control statements).
9. Update parent to child // Set parent to the newly created child, indicating that the new control context is now the child node.
10. Procedure exitControlStatement () // Triggered when the parser exits a control statement in the source code.
11. Update parent to parent of child // Update parent to the parent node of the current child, indicating that the control flow is moving out of the current control statement back to its enclosing context.
12. Procedure generateOutput(out, enableImages) // Responsible for generating the output of the control flow graph after processing the source code.
13. Try to generate graphs from flowGraph, print error message if exception occurs // Attempt to generate textual (.dot file) and visual representations of the control flow graph. If an error occurs during generation, print an error message to notify the user.
14. EndClass // End of the ControlFlowGraphListener class, concluding the process of generating the control flow graph.

Fig. 7. Algorithm 3: Control Flow Graph (CFG) generation.

Control Flow Graphs (CFGs) are essential in clone detection as they show the execution flow within functions, highlighting both structural and semantic similarities between code snippets. By capturing the sequence of statements and decision points, such as if-else conditions or switch-case statements, CFGs help to identify function-level clones, even when syntactic variations, such as variable renaming or code reordering, are present. This makes CFGs crucial for detecting deeper, logic-based similarities that go beyond syntactic-level code comparisons.

Algorithm 3 starts with the definition of a class named ControlFlowGraphListener (step 1), which is responsible for managing the generation of the CFG. Within this class, three key variables—root, parent, and child—are declared as instances of GenericGraphItem (step 2). These variables represent the nodes within the control flow graph, where root serves as the starting point of the graph, parent indicates the current node or context within the graph, and child represents new nodes created by control statements. Additionally, a new instance of GenericGraph, referred to as flowGraph, is initialized (step 3). This flowGraph will store the entire structure of the CFG, capturing the relationships between control statements in the C++ source code.

The algorithm proceeds by defining a procedure for entering function definitions (step 4), which is executed upon entering a function definition. Within this procedure, a new root node is created using the appropriate name, and it is added as the root node of the flowgraph (step 5).

In (step 6) the enterControlStatement procedure is then defined to handle the entry of control statements, such as statements or loops. When encountering a control statement, a new child node is created with the corresponding name and parentItem (step 7). The child node is added to the flowGraph, and if a parent node exists, it is also added as a subItem of the parent (step 8). The parent variable is updated to the child node, reflecting the current hierarchy within the control flow graph (step 9).

To ensure the correct structure of the control flow graph, the algorithm includes the exitControlStatement procedure (step 10). This procedure updates the parent variable to the parent of the current child node, facilitating the proper traversal of the control flow hierarchy (step 11).

The algorithm proceeds with the generateOutput procedure (step 12), which is responsible for generating the desired output from the flowGraph. It attempts to generate graphs from the flowGraph and, if an exception occurs during the process, it prints an error message to indicate the issue (step 13).

The algorithm concludes with the termination of the ControlFlowGraphListener class (step 14), marking the end of the CFG generation process and the encapsulation of all related functionalities within the class.

The output of this phase is a set of (.dot) files that utilize the (Graphviz) library [33] to visualize the CGs and CFGs for both code snippets being matched, providing valuable insights into the relationships and structure of the code components.

Fig. 8 and Fig. 9 show the CGs and CFGs for the C++ code snippet (a) depicted in Fig. 2 and its corresponding translated binary (b) in Fig. 4.

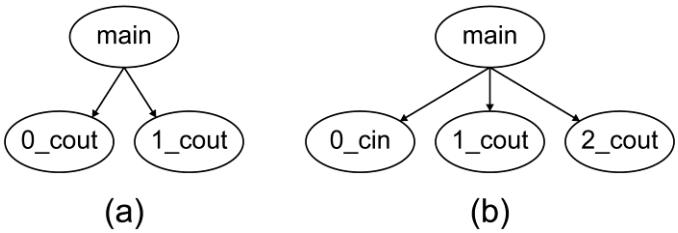

Fig. 8. CG example: (a) CG for the C++ source code; (b) CG for the corresponding translated binary code.

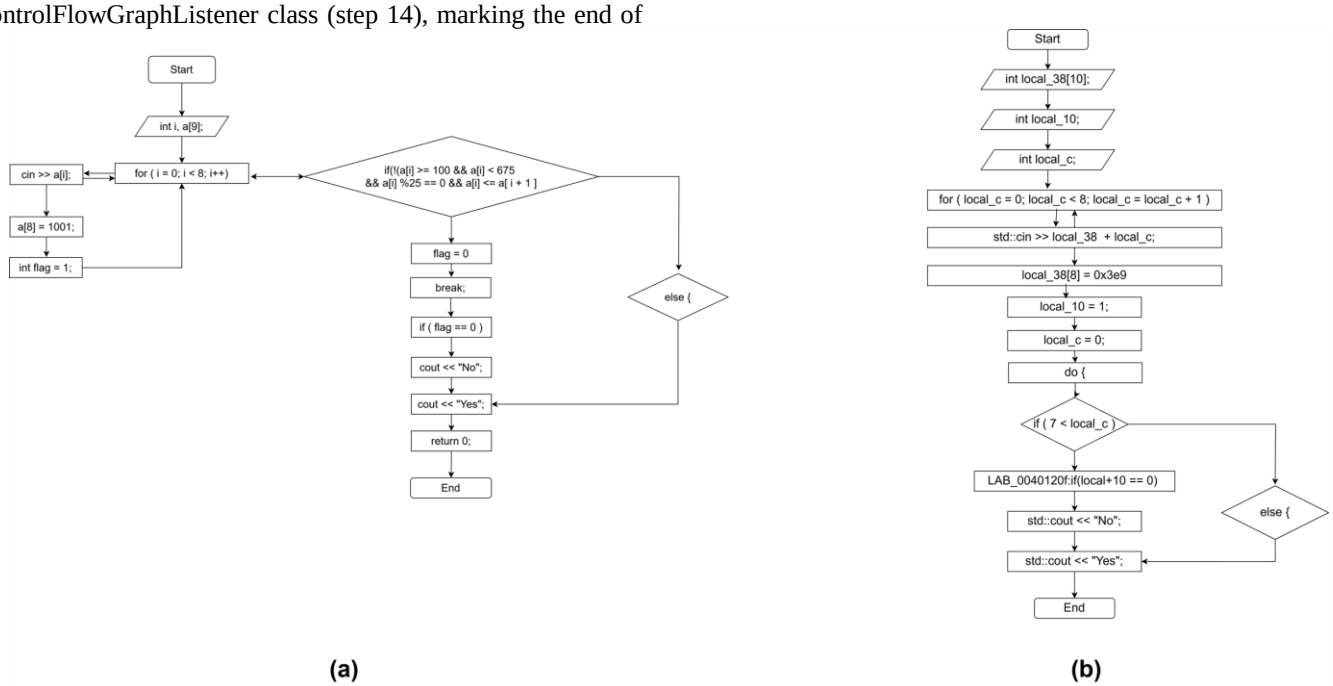

Fig. 9. CFG example: (a) CFG for the C++ source code; (b) CFG for the corresponding translated binary code.

C. Measuring Code Similarity

In this phase, the generated CGs and CFGs of both the original and generated C++ source code are matched using appropriate similarity measurement technique to quantify the degree of similarity between the two code snippets. This step helps in identifying the common patterns and structures shared by the two representations.

We adopt the weighted Jaccard index as a similarity metric, which is a similarity measure that compares the similarity between elements of two sets, taking into account the weights associated with the elements in the sets. It extends the standard Jaccard index by considering both the presence of common elements and their respective weights. By incorporating weights, it provides a more nuanced measure of similarity, allowing for a more accurate comparison of sets. The regular Jaccard index treats all elements equally, ignoring any variation in their significance, whereas the weighted Jaccard index acknowledges the diverse impact that elements may have on the overall similarity.

In the context of code clone detection, where the goal is to identify code fragments that are similar or nearly identical to each other, the weighted Jaccard index plays a crucial role. Code clones may not be exact replicas but can have variations due to modifications, such as variable renaming or code reordering. To capture these variations, the weighted Jaccard

index considers elements occurring in code snippets (specific C++ keywords noted in Table I). This makes it particularly useful in detecting code clones that have undergone modifications while maintaining a core similarity. Furthermore, the weighted Jaccard index enables fine-grained clone detection, where different parts of a code snippet can be assigned different weights based on their significance. This allows for more precise code clone detection by focusing on specific parts of the code that are deemed more critical or unique.

The weighted Jaccard index can be calculated as follows:

$$Weighted\ Jaccard\ index = (\sum minimum\ weights\ of\ common\ elements) / (\sum maximum\ weights\ of\ all\ elements) \quad (1)$$

Eq. (1) represents the calculation of the weighted Jaccard index between two sets (nodes contents of CGs and CFGs of the given two code snippets), where the weights associated with the elements (keywords) are taken into consideration. The numerator of the equation involves summing the minimum weights of the common elements (keywords), indicating the combined importance of the shared elements. The denominator involves summing the maximum weights of all elements (keywords) in both sets (nodes contents of CGs and CFGs of the given two code snippets), representing the total potential importance of any element in the sets. By dividing the sum of the minimum weights by the sum of the maximum weights, the weighted Jaccard index provides a value between 0 (completely unmatched) and 1 (completely matched), indicating the degree of similarity between the nodes contents of CGs and CFGs of the given two code snippets, while considering the weights assigned to their elements (keywords).

As for the weights of C++ keywords, we gathered all standard C++ keywords from the Microsoft website (https://learn.microsoft.com/en-us/cpp/cpp/keywords-cpp?view=msvc-170#standard-c-keywords). Then, different weights were assigned to the keywords according to their significance in the context of CGs and CFGs and the thorough analysis of the code within our curated dataset. For one source code file, the weights were assigned to the keywords based on Table I, with 100 total weights.

TABLE I. WEIGHTS ASSIGNED TO C++ KEYWORDS

Keyword type	Weight
Function call (main, cin, cout)	2
Control statements (if, for, while, do)	2
Data types, frequently used	2
Data types, infrequently used	1
All other keywords	0.1 – 0.9

The Jaccard index for measuring the similarity score between two functions has been used in many studies, including [4], in which the authors emphasize that their method is relatively accurate but also slow. They utilize a combination of the Jaccard index and the longest common subsequence (LCS) algorithm, which takes into account the order of elements in two sequences to perform function comparisons. However, in our own work, we are primarily concerned with accurately determining the similarity between graphs of two code snippets, regardless of the order of their nodes or keywords. As a result, we only use the Jaccard index, which has high accuracy, and we do not consider sequence order in our analysis. By avoiding the LCS algorithm, which has a high time complexity of O (n^2), we are able to achieve faster execution times.

We developed the matching module using Python (version 3.10.10) with the libraries NetworkX (version 2.8.4) [34], openpyxl (version 3.0.10) [35], NumPy (version 1.23.5) [36], and PyGhraphviz (version 1.9) [37] to import the generated call and control flow graphs (.dot files) in the previous step, read the node components, and compute the combined weighted Jaccard index between the graphs. By computing the weighted Jaccard index between the call graphs and control flow graphs of the original C++ source code and translated binary code (generated C++ source code), we can obtain a quantitative measure of the similarity between the two code representations. A high-level algorithm for measuring code similarity based on the generated graphs applied in this step is illustrated in Fig. 10. Algorithm 4 is designed to quantify the similarity between two given code graphs represented as text. The algorithm follows a step-by-step process to accomplish this task.

Algorithm 4: Measuring Code Similarity

Input: CGs, CFGs, weights
Output: Similarity score
1. Function get_text_weighted_similarity (graph1Text, graph2Text, weights) // Define a function to calculate the weighted similarity between the textual representations of call and control flow graphs from two code snippets that are being compared, using the provided weights.
2. Clean graph1Text and graph2Text by removing punctuation, digits, and non -related words // Preprocess the input texts by eliminating irrelevant characters and words to focus the comparison on meaningful keywords that affect the similarity score.
3. Initialize total_weights, similarity to 0.0 // Set up variables to accumulate the total weights and the similarity score, both starting at 0.0.
4. Initialize set1 with cleaned graph1Text //Create a set from the cleaned graph1Text, which will contain the unique keywords from the first graph, whether it is the CG or CFG, of the first code snippet.
5. Initialize set2 with cleaned graph2Text // Similarly, create a set from the cleaned graph2Text, which will contain the unique keywords from the second graph, whether it is the CG or CFG, of the second code snippet.

6. For each word in set1 // Begin iterating over each keyword in the first set.
7. If word is in set2 // Check if the current word from set1 also exists in set2, indicating a common keyword between the two graphs, whether it is the CGs or CFGs, of the two code snippet.

8. Add word's weight to similarity and total_weights // If the word is common to both sets, add its weight to both the similarity score and the total weights.
9. Else // If the word is not found in set2.
10. Add word's weight to total_weights //add its weight only to the total weights, as it does not contribute to the similarity.
11. EndIf // End the conditional check.
12. EndFor // End the loop that processes each word in set1.
13. For each word in set2 // Start iterating over each keyword in set2.
14. If word is not in set1 // Check if the current word from set2 is not present in set1, indicating a unique keyword in set2.
15. Add word's weight to total_weights // If the word is unique to set2, add its weight to the total weights.
16. Else // If the word is also in set1.
17. Add word's weight to similarity and total_weights //add its weight to both the similarity score and the total weights, similar to the earlier step.
18. EndIf // End the conditional check.
19. EndFor // End the loop that processes each word in set2.
20. Calculate similarity as similarity / total_weights // Compute the final similarity score by dividing the accumulated similarity score by the total weights, normalizing the result to a value between 0 and 1.
21. Return similarity // Output the calculated similarity score as the result of the function.
22. EndFunction // End of the get_text_weighted_similarity function.

Fig. 10. Algorithm 4: Measuring code similarity.

In step 1, the function "get_text_weighted_similarity" takes three input parameters: "graph1Text" and "graph2Text" (textual representations of CGs and CFGs nodes of the two code snippets being compared), and "weights" (a set of weights associated with each keyword).

Step 2 of the algorithm involves cleaning the "graph1Text" and "graph2Text" by removing punctuation, digits, and non-relevant words, ensuring that only meaningful keywords, which are related to the context of CGs and CFGs, are considered in the similarity calculation.

Next, the variables "total_weights" and "similarity" are initialized to zero, representing the cumulative weights and similarity score, respectively (step 3).

The algorithm then proceeds to create two sets, "set1" and "set2", which contain the cleaned words from the first code snippet "graph1Text" and the second code snippet "graph2Text" respectively (steps 4 and 5).

The algorithm enters a loop where it iterates over each keyword in "set1" (step 6). For every keyword encountered, it checks if the keyword is present in "set2" (step 7). If it is, the keyword's weight is added to both the similarity and total_weights variables (step 8). If the keyword is not found in "set2" (step 9), only the keyword's weight is added to total_weights (step 10). After processing all keywords in "set1", the algorithm proceeds to iterate over each keyword in "set2" (step 13). If a keyword is not present in "set1" (step 14), its weight is added to total_weights (step 15). Conversely, if the keyword is found in "set1" (step 16), its weight is added to both similarity and total_weights (step 17).

Once both sets have been processed, the algorithm calculates the similarity by dividing the similarity score by the total_weights (step 20). This normalization ensures that the similarity value falls within a meaningful range. Finally, the algorithm outputs the calculated similarity (S) as the result (step 21).

The resulting similarity score ranges from 0 (unmatched/completely different) to 1 (matched/ completely identical). Within the scope of our research, achieving a perfect matching score of 1 between a C++ source code and its corresponding translated binary code is unfeasible. This is because the Ghidra decompiler, in processing compiled source code (binary file), renames original variables and introduces auxiliary variables to manage complex data structures such as arrays, vectors, and lists. These modifications inherently decrease the similarity between the original source code and its corresponding decompiled binary. Thus, a perfect similarity score of 1 remains un-attainable for the CFGs of matched pairs, highlighting the inevitable differences caused by such changes.

When assessing the similarity between two different code snippets (C++ source code and a binary code compiled from another C++ source code), a threshold value of up to 0.55 is considered indicative of a high degree of similarity, attributed to the syntactical similarities inherent in C++ programs. Furthermore, these inherent syntactical similarities ensure that code pairs exhibit a significant degree of similarity, necessitating categorization within the range $0<S<1$.

Consequently, a similarity score below the predefined threshold of 0.55 ($0 \leq S < 0.55$) is considered evidence of unmatched pairs (minimal similarity), indicating substantial differences between the code pairs. On the other hand, a score in the range of $0.55 \leq S < 1$ indicates matched pairs (highly similar), reflecting a significant degree of similarity between the code pairs. This differentiation is crucial for discerning the gradations of similarity and the threshold separating matched from unmatched code pairs in our analysis.

The optimal similarity threshold of 0.55 was selected after a thorough analysis of our curated dataset from Atcoder website. This value was determined by examining the similarity levels in the dataset's code pairs (C++ source code and binary code, whether the binary originated from the same source code or a different one). In this study, we tested various thresholds and, through empirical analysis, fine-tuned the similarity values to enhance precision and recall. This refinement contributes to the overall accuracy of the system.

IV. Evaluation and Results

A. Dataset

To construct our dataset, we collected 100 C++ source code files from AtCoder [19], a reputable online platform recognized for hosting competitive programming competitions and serving as a hub for programmers. The selection of AtCoder was based on its diverse range of problem types, solution strategies, and user contributed submissions which collectively provide a rich variety of real-world coding styles. This diversity enhances the generalizability and robustness of our system in clone detection scenarios. The collected files span multiple versions of the C++ language, from CPP 11 to CPP 20, ensuring relevance to modern software development practices. The use of AtCoder as a dataset source is supported by recent studies such as CLCDSA [38], ZC3 [39], and TCCCD [40], which employed AtCoder submissions to evaluate clone detection models. Based on these studies, we can confirm AtCoder's capability of capturing diverse coding patterns and use it for training and validating our proposed BSMDG system. Each C++ source code file in the dataset was accompanied by relevant information, including submission time, task title, user who uploaded the code, code size, and execution time.

To facilitate the matching process, each C++ source code file was compiled using the GNU Compiler Collection (GCC 13.2), resulting in the generation of its corresponding binary (.exe) file. We trained and tested the proposed system using this dataset on a Fedora Linux 39 machine with four 2.40 GHz processors and 15.5 GB of RAM.

To organize the dataset, we implemented the 80/20 rule [41], which is a practical guide-line suggesting that approximately 80% of the data should be allocated to training the system and 20% should be used to test its performance. Accordingly, eighty C++ source code files and their corresponding binary files were allocated for training our proposed system, which involved determining thresholds, where 50% of the subset (forty code pairs) was considered matched (binary and matching source code from which it was compiled), and 50% (the other forty code pairs) was considered

unmatched (binary and unmatching source code) to ensure that neither set was biased during training. The remaining 20 C++ source code files and their corresponding binary files from the dataset were reserved for testing the proposed system, where 50% of this subset comprised matched pairs and 50% comprised unmatched pairs.

Afterwards, a series of steps were carried out (illustrated in Section III), resulting in the creation of eleven files for each individual C++ source code file. In total, the dataset contains 1100 files encompassing all of the C++ source code files and the files generated for each of them including the corresponding binary files.

We labeled each pair of C++ source code and binary files in the dataset according to the nature of the matching process. Specifically, the labels indicate whether the matching was performed between a C++ source code file and its corresponding binary file (matched pairs), or between a C++ source code file and a binary file of another C++ source code file (unmatched pairs).

B. Evaluation Metrics

A predefined threshold value of 0.55 was used to test the performance of the proposed system. During this procedure, 20 pairs of C++ source code files and their corresponding translated counterparts—C++ source code generated from binary files by our translation module—were tested. These files were randomly selected from our curated dataset from Atcoder and categorized as matched (M) if the source code file was compared against its corresponding binary file, or unmatched (UM) if the source code file was compared against a binary file compiled from different source code.

To assess the effectiveness of the proposed system, several performance metrics were computed, including precision, recall, F-score, and accuracy. The equations used to calculate these metrics are as follows:

$$Precision = TP / (TP + FP) \quad (2)$$

In the context of our research, precision represents the proportion of correctly classified matched pairs out of all pairs classified as matched.

$$Recall = TP / (TP + FN) \quad (3)$$

Recall signifies the proportion of matched pairs that were correctly identified by the system among all actual matched pairs.

$$F-score = 2 * (Precision * Recall) / (Precision + Recall) \quad (4)$$

The F-score considers both precision and recall simultaneously and provides an overall measure of the system's effectiveness in identifying matched pairs in our dataset.

$$Accuracy = (TP + TN) / (TP + TN + FP + FN) \quad (5)$$

Accuracy provides an assessment of the system's ability to correctly classify both matched and unmatched pairs, representing the overall performance of the system.

Therefore, the precision was 0.75, the recall was 0.9, the F-score was 0.82, and the accuracy was 0.80 for the given test subset. Table II summarizes the performance metrics of the proposed system (BSMDG). Fig. 11 shows the bar chart for these metrics.

TABLE II. PERFORMANCE METRICS OF THE PROPOSED SYSTEM (BSMDG) ON OUR CURATED DATASET FROM ATCODER (THRESHOLD AT 0.55)

Precision	Recall	F-score	Accuracy
0.75	0.9	0.82	0.80

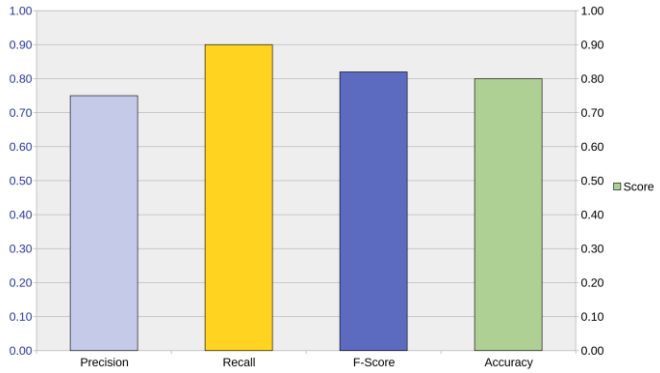

Fig. 11. Performance metrics of the proposed system (BSMDG) on our curated dataset from Atcoder (threshold at 0.55).

V. DISCUSSION

The bar chart (Fig. 11) visualizing the system's performance metrics—precision (0.75), recall (0.9), F-score (0.818), and accuracy (0.8)—provides a comprehensive quantitative assessment of its effectiveness. The precision of 0.75 indicates that 75% of the positive (matched) predictions made by the proposed system are accurate, reflecting a solid performance in specificity. However, this also means that 25% of the positive predictions are false positives, which suggests that the system may be identifying unmatched pairs as matched. This could be due to the system's sensitivity to minor code similarities (syntactical similarities inherent in C++ programs) that do not constitute true matches, indicating a need for refinement in the matching (measuring similarity) algorithms to reduce false positives.

The recall of 0.9 is particularly high, showing that the system successfully identifies 90% of all actual positive cases. This high recall underscores the system's effectiveness in detecting binary–source code matches, which is critical in applications like code clone detection, copyright infringement detection and software forensics, where missing a genuine clone could be highly detrimental. The high recall suggests that the system is comprehensive in its search for potential matches, but this comes at the cost of lower precision, indicating a trade-off between sensitivity and specificity. BinPro [13], B2SFinder [42], and XLIR [14] have also found a trade-off between precision and recall. This trade-off highlights the importance of parameter tuning. Several parameters were empirically adjusted based on dataset characteristics, such as keyword weights in Algorithm 4 and similarity thresholds.

The F-score, which balances precision and recall, is 0.82. This score suggests that while the system performs well

overall, there is still room for improvement, particularly in reducing the false positive rate to enhance precision without sacrificing recall. The fact that the F-score is closer to the recall than the precision indicates that the system is more inclined towards sensitivity, which is advantageous in scenarios, where detecting all potential matches is more important than minimizing false positives.

An overall accuracy of 0.80 confirms the system's reliability in distinguishing between matched and unmatched pairs across the test subset. However, the accuracy metric alone may not fully capture the system's performance, especially given the imbalance between precision and recall. The system's ability to correctly identify true negatives (unmatched pairs) also contributes to this accuracy, but the relatively lower precision suggests that there are still challenges in distinguishing between true matches and near-misses.

In summary, while the BSMDG system demonstrates strong recall and overall accuracy, its lower precision highlights the need for further refinement in its matching (measuring similarity) algorithms. These improvements could involve more sophisticated filtering of minor code similarities that do not represent true matches, thereby enhancing the system's specificity. Such enhancements would be crucial for increasing the precision while maintaining the high recall, leading to a more balanced and effective tool for binary–source code matching detection.

Researchers have demonstrated similar results in recent work with binary-source code similarity and clone detection, such as CCGraph [27] and GraphBinMatch [43], , highlighting the need to combine structural and semantic features to make accurate comparisons.

VI. Case Study

A. Experimental Setup and Evaluation Dataset

We further evaluated the performance of our proposed system (BSMDG) on unseen data using the POJ-104 dataset [44] to assess the system's generalizability and real-world applicability. Several studies have focused on code clone detection using various methodologies and datasets. However, to provide a comprehensive analysis of our findings, we compared our performance metrics (precision, recall, F-score) against those reported in previous research studies that utilized the POJ-104 dataset for code clone detection. POJ-104 is a comprehensive dataset designed for the evaluation of code clone detection methodologies. It consists of C++ source code submissions provided by 500 students in response to 104 distinct programming challenges from an online judge (OJ) platform designed for educational uses, resulting in a total of 52,000 source code files. In this study, the compilation of C++ source code files into binary executables was performed using the GNU Compiler Collection (GCC version 13.2). Source code files that failed to compile were excluded from further analysis, resulting in the generation of 44,096 binary executables. The primary cause for the inability to compile certain files was identified as the pre-processing stage of dataset preparation, during which headers were removed, leading to compilation failures. To address this issue, essential headers, including 'iostream' and 'string', were reintegrated into the source code files to facilitate successful compilation. This process of header reintegration was executed through the deployment of Python scripts, designed to automate the inclusion of frequently used headers. Nonetheless, a subset of the source code files required libraries that are either rarely used or not frequently encountered, which, in turn, led to a reduction in the total number of source code files that could be successfully compiled. Next, we leveraged a virtual machine from Amazon Elastic Compute Cloud (Amazon EC2 t2.2xlarge instance) [45], equipped with 8 vCPUs and 32 GB of RAM, to enhance the efficiency of the Ghidra decompilation process and boost our system's performance. In the decompilation process we used multithreading in our Python scripts to execute Ghidra's headless analyzer [46], a command-line-based (non-GUI) version of Ghidra, through calling the 'analyzeHeadless' shell script, which is located in the Ghidra program path. The 'analyzeHeadless' script facilitates automated, headless (non-interactive) analysis of binary files, enabling users to automate importing, decompiling, disassembling, and other analyses on binary executables and object files. This can be particularly useful in environments, where graphical interfaces are not available, such as servers, or in automated pipelines, where human interaction is not feasible. As such, it is a powerful tool for automating binary file analysis in reverse engineering and security auditing. BSMDG takes the decompiled binary–source code pairs (cloned or matched and non-cloned or unmatched) as input to assess their similarity scores. Next, the system assesses whether the pairs are classified as cloned or non-cloned by utilizing a similarity threshold of 0.7. This threshold was determined through empirical analysis of the POJ-104 dataset, where precision and recall were fine-tuned to identify the optimal value.

B. Baselines and Comparison

Since most previous studies have conducted binary-source code matching at the IR level, we compared our proposed system with the pioneering research that utilized the POJ-104 dataset in their analysis to facilitate direct comparison. The baseline studies employing the POJ-104 dataset for clone detection are as follows:

1) BinPro [13]: This model was designed to address the issue of detecting similarities between source code and binary code, even in cases, where the compiler or optimization level remains unspecified. Utilizing machine learning approaches, BinPro identifies the most effective code features (FCGs) for assessing the similarity between binary and source code. Through the application of static analysis tools, these features are extracted and analyzed, enabling the matching of binary and source codes via a bipartite matching algorithm (i.e., Hungarian algorithm).

2) B2SFinder [42]: This model leverages a sophisticated approach to identify binary code clones, extracting seven distinct features from both binary and source code across three key dimensions: strings, integers, and control-flow. It utilizes a weighted feature-matching algorithm designed to accommodate the diverse nature of these features. This algorithm assigns weights to code feature instances, taking

into account their uniqueness and the frequency of their appearance, to efficiently match binary and source code by inferring traceable characteristics.

3) XLIR (Transformer) [14]: This method involves a state-of-the art neural network model based on transformer technology in binary–source code matching. Central to XLIR's methodology is the use of LLVM intermediate representation (IR), as implied by its name. To process LLVM IR tokens, XLIR utilizes a BERT model that has been pre-trained. The initial phase involves pre-training the neural network on a large corpus of external LLVM-IR, employing masked language modeling (MLM) as a preliminary step. This process is designed to capture meaningful representations of LLVM-IR tokens effectively. Following the embedding of these tokens, XLIR maps them into a common space, where the representations of LLVM-IR are jointly learned through a ternary loss function. By adopting this strategy, XLIR can match binary and source code from various programming languages.

4) XLIR (LSTM) is a variant of the proposed approach XLIR (Transformer) [14] in which LSTM network is used to encode the IRs.

5) GraphBinMatch [43]: This model leverages a graph neural network to learn the similarity between binary and source codes. It represents binary and source codes as graphs, incorporating control flow, data flow, and call flow information. This graph-based representation helps the neural network model better understand the structure and semantics of the code.

Table III presents a comparison of performance metrics between our proposed system (BSMDG) and the baselines on the POJ-104 dataset. Fig. 12 visually represents the comparative performance metrics listed in Table III.

TABLE III. PERFORMANCE OF THE PROPOSED SYSTEM (BSMDG) AGAINST BASELINES ON THE POJ-104 DATASET (THRESHOLD AT 0.7)

System	Matching Level	Methodology	Precision	Recall	F-score
BinPro [13]	Source code with Assembly code (IR) of corresponding binary code	AI-based	0.38	0.42	0.40
B2SFinder [42]	Source code with Assembly code (IR) of corresponding binary code	Rule-based	0.43	0.46	0.44
XLIR(Transformer) [14]	LLVM_IR of source code with LLVM-IR of corresponding binary code	AI-based	0.85	0.86	0.85
XLIR (LSTM) [14]	LLVM_IR of source code with LLVM-IR of corresponding binary code	AI-based	0.67	0.72	0.69
GraphBinMatch [43]	LLVM_IR of source code with LLVM-IR of corresponding binary code	AI-based	0.88	0.86	0.87
BSMDG (the proposed system)	**Source code with generated source code of corresponding binary code**	**Rule-based**	**0.97**	**0.83**	**0.89**

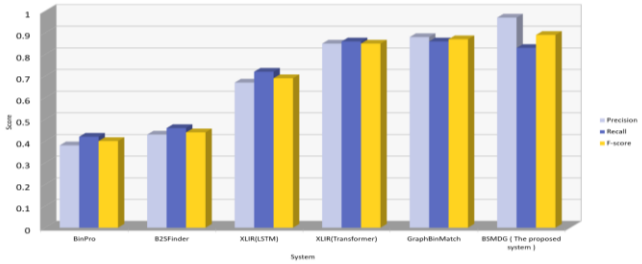

Fig. 12. Performance of the proposed system (BSMDG) against baselines on the POJ-104 dataset (threshold at 0.7).

In comparison to the previous studies on the POJ-104 dataset, our decompilation graph-based clone detection system (BSMDG) demonstrates favorable performance metrics, showcasing its effectiveness in identifying code clones within the POJ-104.

Fig. 12 shows the performance of various binary analysis tools, including our proposed system (BSMDG). All of these, except B2SFinder, employ advanced AI models, such as LSTM and Transformer architectures. This visualization highlights the precision, recall, and F-score of each system, offering a clear comparison across these critical performance metrics.

Our proposed system stands out for achieving the highest precision (0.97) among all the tools evaluated, indicating its exceptional ability to correctly identify true positives while minimizing false positives. This is particularly noteworthy considering that BSMDG achieves this performance without relying on AI models, which are typically associated with higher computational costs and complexities.

The comparison reveals that while AI-based systems like XLIR (using both LSTM and Transformer architectures) and GraphBinMatch demonstrate strong performance, especially in terms of recall and F-score, BSMDG surpasses these systems in precision. Moreover, BSMDG achieves a competitive F-score of 0.89, highlighting its balanced performance in both precision and recall. Despite having a slightly lower recall rate compared to the highest AI-based systems, this calls for enhancements to boost its performance and improve its competitive stance.

The benefit of our proposed rule-based system over AI-based systems lies in its efficiency and simplicity. By not relying on AI, BSMDG avoids the need for extensive training data, computational resources, and tuning of complex models, making it a more accessible and easier-to-deploy solution for code analysis and clone detection scenarios. Additionally, the high precision of BSMDG makes it particularly valuable in contexts, where the cost of false positives is high, ensuring that resources are focused on truly relevant findings.

This comparison underscores the significance of developing innovative, non-AI methodologies for binary analysis, demonstrating that such approaches can not only compete with but, in some aspects, surpass AI-based models.

BSMDG represents a significant advancement in the field, providing a highly accurate, efficient, and practical tool for binary analysis and clone detection that is particularly suited for applications requiring high precision without the overhead of AI models, making it an invaluable asset in the field of software engineering and security.

VII. Limitations

Although our proposed binary–source code clone detection approach is promising, there are some limitations to consider:

1) The proposed approach could lead to either false positives or false negatives based on the selected similarity threshold for comparing code graphs. Setting the threshold too low might lead to an increase in false positives, whereas a higher threshold could cause more false negatives.

2) The task of decompiling and translating code that includes non-standard headers remains a significant challenge that demands specialized knowledge and careful analysis. For instance, decompiling and translating code that employs the <bits/stdc++.h> header presents obstacles due to its non-standard nature, making it difficult to ascertain the precise headers it encompasses. Recognizing that decompilation is a complex process, success in managing particular headers greatly depends on the sophistication of the tools and methodologies employed. Additionally, it is vital to understand that the decompiled code may not be an exact representation of the original source code, as some details and optimizations could be lost in the compilation process.

3) The presence of goto statements in the decompiled code can reduce the similarity level between the original C++ source code and the generated C++ source code produced by the transpiler.

4) The evaluation was performed using specific datasets, and the system's performance could vary when applied to other datasets that have different degrees of code similarity.

VIII. Conclusion and Future Work

A. Conclusion

In this research, we conducted an extensive investigation into code clone detection by integrating innovative techniques and methodologies, focusing on the matching between C++ source code and binary code at the source code level, rather than at the binary, intermediate representation (IR), or pseudo-code levels. A pivotal strategy we employed is decompilation, which plays a crucial role in bridging the semantic gap between binary and source code. This step is facilitated by the use of Ghidra's decompiler and a custom-developed transpiler (source-to-source compiler) based on ANTLR, enabling the transformation of binary code into a high-level C++ source code.

Using these cutting-edge tools, we improved the precision and dependability of our experiments, contributing to the robustness of our findings. We captured the intricate relationships and structures embedded within the code by fusing graph-based code representations, namely call and control flow graphs. We conducted a nuanced analysis that outperforms conventional clone detection methods by employing the weighted Jaccard index as a similarity measure.

The integration of decompilation and graph similarity analysis in our methodology not only enhances the capability to detect code clones in binary–source code by considering structural and semantic similarities, but also addresses the challenges posed by the limited availability of source-level information in binaries and disassemblies and the significant differences between source code and binary or object code after compilation. Our approach contributes a novel perspective to the field, suggesting a shift towards more context-rich, semantic-based analyses for binary-source code matching and clone detection.

As a case study, we evaluated the proposed rule-based approach (BSMDG) against several baseline studies that use AI techniques to detect C++ code clones, using the POJ-104 dataset. The experimental results show that BSMDG outperforms other baseline studies. When compared with BinPro, B2SFinder, XLIR, and GraphBinMatch, BSMDG improves the F-score from 0.40, 0.44, 0.69, 0.85, and 0.87, respectively, to 0.89, achieving a 90% accuracy rate.

The methodology demonstrated in this study not only underlines the importance of analyzing clones at the source code level but also emphasizes the potential of our approach to contribute significantly to areas such as malware detection, vulnerability analysis, and reverse engineering. The BSMDG system can be used in real-world software development environments to improve security and code verification. As an example, it could be integrated into Continuous Integration or Continuous Deployment (CI or CD) pipelines to help catch any unwanted or suspicious changes early in the development cycle. As part of malware analysis, BSMDG can detect reused or copied code within binaries, assisting in threat identification and tracing the code's origin. A further benefit of BSMDG is that it can assist in detecting vulnerabilities by comparing binary code with secure source code versions, thereby identifying unauthorized or accidental modifications that result in vulnerabilities. According to the findings of this study, high-level language analysis within binary-source code matching needs to be further advanced to improve clone detection tools for improved accuracy and applicability to software engineering and cybersecurity.

B. Future Work

While the current study has established a solid foundation for binary–source code matching and clone detection at the source code level, several avenues for future research have been identified to further refine and extend our methodology:

1) Syntax error resolution: One of our priorities moving forward is to improve the translation module so it can generate a higher percentage of compilable C++ code from the C-like pseudo-code output by the decompiler. This will play a key role in making our system more accurate and dependable. To achieve this, we plan to refine the grammar to better handle common patterns in Ghidra's pseudo-code and introduce pre-processing steps to clean up irregular structures before

parsing, ultimately reducing syntax errors and improving translation quality.

2) Optimization of graph similarity algorithm: We plan to explore more advanced graph similarity methods to improve the system's precision and recall, especially for detecting semantic code clones.

3) Obfuscation resistance: While the current system handles simple transformations, future improvements will focus on making it more robust against common obfuscation techniques like instruction substitution and control flow flattening.

4) Cross-language compatibility: In future work, we plan to extend our approach to support cross-language binary–source code matching, making it possible to detect code clones across different programming languages like Java and Python.

5) Hybrid analysis integration: We plan to combine dynamic analysis with our current static analysis to build a hybrid approach. This will offer a more complete view of both code behavior and structure, which could lead to more accurate and reliable clone detection.

By addressing these areas, future research can enhance the utility and applicability of binary–source code matching and clone detection tools, further bridging the gap between binary and high-level source code analysis for security and maintenance purposes.

ACKNOWLEDGEMENT

This research was funded by the Deanship of Scientific Research (DSR) at King Abdulaziz University, Jeddah, Saudi Arabia, under grant number "RG-12-611-43".

REFERENCES

[1] J. Krüger and T. Berger, "An empirical analysis of the costs of clone- and platform-oriented software reuse," presented at the Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, 2020. Available: https://doi.org/10.1145/3368089.3409684

[2] B. Wan, S. Dong, J. Zhou, and Y. Qian, "SJBCD: A Java Code Clone Detection Method Based on Bytecode Using Siamese Neural Network," Applied Sciences, vol. 13, no. 17, p. 9580, 2023.

[3] P. M. Caldeira, K. Sakamoto, H. Washizaki, Y. Fukazawa, and T. Shimada, "Improving Syntactical Clone Detection Methods through the Use of an Intermediate Representation," presented at the 2020 IEEE 14th International Workshop on Software Clones (IWSC), 2020. Available: https://doi.ieeecomputersociety.org/10.1109/IWSC50091.2020.9047637

[4] Y. Hu, H. Wang, Y. Zhang, B. Li, and D. Gu, "A Semantics-Based Hybrid Approach on Binary Code Similarity Comparison," IEEE Transactions on Software Engineering, vol. 47, no. 06, pp. 1241-1258, 2021.

[5] K. Sendjaja, S. A. Rukmono, and R. S. Perdana, "Evaluating control-flow graph similarity for grading programming exercises," 2021, pp. 1-6: IEEE.

[6] Y. Gui et al., "Cross-Language Binary-Source Code Matching with Intermediate Representations," presented at the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022. Available: https://doi.ieeecomputersociety.org/10.1109/SANER53432.2022.00077

[7] Z. Yu, W. Zheng, J. Wang, Q. Tang, S. Nie, and S. Wu, "CodeCMR: cross-modal retrieval for function-level binary source code matching," presented at the Proceedings of the 34th International Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 2020.

[8] S. Feng, W. Suo, Y. Wu, D. Zou, Y. Liu, and H. Jin, "Machine Learning is All You Need: A Simple Token-based Approach for Effective Code Clone Detection," presented at the Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 2024. Available: https://doi.org/10.1145/3597503.3639114

[9] Y.-B. Jo, J. Lee, and C.-J. Yoo, "Two-Pass Technique for Clone Detection and Type Classification Using Tree-Based Convolution Neural Network," Applied Sciences, vol. 11, no. 14, p. 6613, 2021.

[10] Y. Ji, L. Cui, and H. H. Huang, "BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network," presented at the Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, 2021. Available: https://doi.org/10.1145/3433210.3437533

[11] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 261-271: IEEE.

[12] R. T. Yarlagadda, "Approach to computer security via binary analytics," International Journal of Innovations in Engineering Research and Technology [IJIERT], 2020.

[13] D. Miyani, Z. Huang, and D. Lie, "Binpro: A tool for binary source code provenance," arXiv preprint arXiv:1711.00830, 2017.

[14] Y. Gui et al., "Cross-language binary-source code matching with intermediate representations," in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 601-612: IEEE.

[15] X. Wang et al., "Decompilation Based Deep Binary-Source Function Matching," in International Conference on Science of Cyber Security, 2023, pp. 244-260: Springer.

[16] Z. Yu, W. Zhang, and T. Xu, "Leveraging IR based sequence and graph features for source-binary code alignment," in 2024 4th International Conference on Neural Networks, Information and Communication (NNICE), 2024, pp. 175-180: IEEE.

[17] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," arXiv preprint arXiv:2403.05286, 2024.

[18] N. S. Agency. (2019, 13/11/2024). Ghidra. Available: https://ghidra-sre.org/

[19] A. Inc. (2024). AtCoder. Available: https://atcoder.jp/home

[20] B. S. Baker, U. Manber, and R. Muth, "Compressing differences of executable code," 1999.

[21] H. Flake, Structural comparison of executable objects. Gesellschaft für Informatik eV, 2004.

[22] Hex-Rays. (2024, 11/10/2024). IDA Pro. Available: https://www.hex-rays.com/products/ida/

[23] D. Pizzolotto and K. Inoue, "BinCC: Scalable Function Similarity Detection in Multiple Cross-Architectural Binaries," IEEE Access, vol. 10, pp. 124491-124506, 2022.

[24] D. Yu, Q. Yang, X. Chen, J. Chen, and Y. Xu, "Graph-based code semantics learning for efficient semantic code clone detection," Information and Software Technology, vol. 156, p. 107130, 2023.

[25] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, 2020, pp. 516-527.

[26] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "Programl: Graph-based deep learning for program optimization and analysis," arXiv preprint arXiv:2003.10536, 2020.

[27] Y. Zou, B. Ban, Y. Xue, and Y. Xu, "CCGraph: a PDG-based code clone detector with approximate graph matching," in Proceedings of the 35th IEEE/ACM international conference on automated software engineering, 2020, pp. 931-942.

[28] K. Sendjaja, S. A. Rukmono, and R. S. Perdana, "Evaluating control-flow graph similarity for grading programming exercises," in 2021 International Conference on Data and Software Engineering (ICoDSE), 2021, pp. 1-6: IEEE.

[29] X. Wang et al., "CODE-MVP: Learning to represent source code from multiple views with contrastive pre-training," arXiv preprint arXiv:2205.02029, 2022.

[30] T. Parr, The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2013.

[31] A. Project. (15/09/2024). ANTLR. Available: https://www.antlr.org/

[32] GitHub. (2024, 02/10/2024). ANTLR4 Grammar for C++14. Available: https://github.com/antlr/grammars-v4/blob/master/antlr/antlr4/examples/CPP14.g4

[33] T. G. project. (2024, 15/10/2024). Graphviz - Graph Visualization Software. Available: http://www.graphviz.org/

[34] A. A. S. Hagberg, Daniel A.; Swart, Pieter J. (2024, 10/10/2024). NetworkX. Available: https://networkx.org/

[35] E. G. Charlie Clark, and contributors. (2024, 13/10/2024). openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files. Available: https://openpyxl.readthedocs.io/

[36] C. R. H. a. K. J. M. a. S. e. f. J. et al. (2020, 20/09/2024). NumPy. Available: https://numpy.org/

[37] P. Developers. (2024, 18/10/2024). PyGraphviz: Python interface to Graphviz. Available: https://pygraphviz.github.io/

[38] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "CLCDSA: cross language code clone detection using syntactical features and API documentation," presented at the Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, San Diego, California, 2020. Available: https://doi.org/10.1109/ASE.2019.00099

[39] J. Li, C. Tao, Z. Jin, F. Liu, J. Li, and G. Li, "ZC3: Zero-Shot Cross-Language Code Clone Detection," presented at the Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering, Echternach, Luxembourg, 2024. Available: https://doi.org/10.1109/ASE56229.2023.00210

[40] Y. Fang, F. Zhou, Y. Xu, and Z. Liu, "TCCCD: Triplet-Based Cross-Language Code Clone Detection," Applied Sciences, vol. 13, no. 21, p. 12084, 2023.

[41] V. R. Joseph, "Optimal ratio for data splitting," Statistical Analysis and Data Mining: The ASA Data Science Journal, vol. 15, no. 4, pp. 531-538, 2022.

[42] Z. Yuan et al., "B2sfinder: Detecting open-source software reuse in cots software," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 1038-1049: IEEE.

[43] A. TehraniJamsaz, H. Chen, and A. Jannesari, "GraphBinMatch: Graph-Based Similarity Learning for Cross-Language Binary and Source Code Matching," presented at the 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2024. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPSW63119.2024.00103

[44] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in Proceedings of the AAAI conference on artificial intelligence, 2016, vol. 30, no. 1.

[45] I. Amazon Web Services. (2024, 18/11/2024). Amazon EC2 instance types - T2. Available: https://aws.amazon.com/ec2/instance-types/t2/

[46] D. Á. Pérez, Ghidra Software Reverse Engineering for Beginners: Analyze, identify, and avoid malicious code and potential threats in your networks and systems. Birmingham, UK: Packt Publishing, 2021.