Precise Code Fragment Clone Detection

Mariam Arutunian
Center of Advanced Software Technologies
Russian-Armenian University
Yerevan, Armenia
mariam.arutunian@rau.am

Matevos Mehrabyan
Center of Advanced Software Technologies
Russian-Armenian University
Yerevan, Armenia
matevos.mehrabyan@rau.am

Sevak Sargsyan
Center of Advanced Software Technologies
Russian-Armenian University
Yerevan, Armenia
sevak.sargsyan@rau.am

Hayk Aslanyan
Center of Advanced Software Technologies
Russian-Armenian University
Yerevan, Armenia
hayk.aslanyan@rau.am

***Abstract*— Detecting duplicate code fragments referred as "clones", is essential for various aspects of software management, maintenance, and security. This article presents a novel method for detecting code fragment clones, applicable to source and binary code. The method addresses the limitations of existing tools, which often focus on detecting clones of entire functions and are typically specialized for either source or binary code, but not both simultaneously. The developed algorithm analyzes input code fragments against the target project, and outputs all detected fragment clones. For fragment clone detection, it uses program dependence graphs - a data structure unifying data and control flow for the function. In the first step source and binary code are converted to program dependence graph representation. Then unified algorithm is applied for maximal similar subgraphs detection. Code fragments corresponding to detected similar subgraphs are considered as clones. The experimental evaluation of the proposed method demonstrates its effectiveness providing an average 96.9% precision, 92.9% recall for binary code, and 96.5% precision, 93.8% recall for source code.**

Keywords- code clones; program static analysis; binary code; source code.

I. INTRODUCTION

Identifying copied code fragments, referred as fragment clones, are vital for software management, maintenance, and security. It can be applied for several purposes:

1. Software plagiarism detection: identifying copied code helps ensure originality and protect intellectual property,
2. Malware detection and classification: researchers can identify new malware variants by finding similar code patterns of known malicious software fragments,
3. Finding known vulnerabilities and avoiding bug propagation: Sometimes, code fragments containing bugs and vulnerabilities are also copied, making the detection of these fragments crucial for preventing the spread of bugs.

Beyond these specific applications, identifying and managing code clones improves overall software quality and reduces maintenance costs. Code clones can arise for a variety of reasons. For instance, they can occur when software developers copy-paste existing code fragments into their projects with or without modifications. Studies [1] show that about 20% of code is duplicated in software packages. In binary code, compiler optimizations like inlining, and transformations can also create clones.

Modern software projects highly use third-party packages and libraries. A 2024 report by Synopsys [2] revealed that over 96% of commercial software packages incorporate open-source code. Another study of 7,800 open-source projects has shown that 44% of them have at least one pair of identical code fragments [3]. These studies reveal the extensive use of code duplication in software development.

Despite the variety of code clone detection methods and tools, only a few can detect clones of fragments rather than whole functions. Besides, existing tools are focused either on source or binary code clone detection. There is no unified approach to detect both of them.

We propose a novel approach for accurate source and binary code fragments' clones' detection. For accuracy Program Dependence Graphs (PDGs) are utilized, which capture most of the software semantics and robust to code changes. Code clones are identified as maximum similar subgraphs for corresponding source and binary code. The core of the developed tools is the same for the source and binary code clones' detection, where the PDG creation parts are code specific. We consider code fragments as a sequence of instructions for binary or source code. A fragment can correspond to a function, basic blocks, or sequences of instructions in a function. Two code fragments are considered clones if they are similar or identical. Section II gives more strict definitions of both binary and source code fragment clones. The proposed method is implemented as a tool named Fragment Clone Detector (FCD) that takes as input a code fragment, a project, and a percentage of similarity. The tool then outputs all fragments from the target project that are clones of the given fragment with the given percentage of similarity.

In addition to evaluating the quality of the implemented method, we have designed and implemented a testing system, which generates tests, based on real-world projects. Then it executes FCD and calculates precision, recall, and Root Mean Square Error (RMSE) for it. The rest of the paper is organized

as follows: Section II defines code clone types for binary and source code and describes PDG. Section III explores existing research in the field. Sections IV and V detail the proposed approach for detecting code fragment clones. The testing system structure is presented in Section VI. Section VII of the paper presents the results of the experimental evaluation. The final section concludes the paper.

II. BACKGROUND

In this section main ideas used in the work are introduced: code clone types and PDG. Both source and binary code clone types are defined in the Subsection A. And the Subsection B will cover the description of the PDG, its components, and its uses.

A. Code clone types

It is accepted [4] that source code clones have four types. While the definition of source code clones is well-established, the definition of binary code clones has minor differences due to its specifics. The definition of source code clone types:

- **Type 1**: Two source code fragments that are identical except for variations in whitespaces and comments,
- **Type 2**: Two source code fragments that can differ by identifiers, literals, and types. This type also includes Type 1 clones,
- **Type 3**: Two source code fragments with additions, deletions, or modifications of instructions. Includes Type 2 clones too. Type 3 clones are also referred to as non-exact clones,
- **Type 4**: Two source fragments that perform the same calculations but use different instructions. Type 4 clones are also referred as semantic clones.

TABLE I. EXAMPLE OF SOURCE CODE CLONE TYPES.

Original code	*Type-1*

float sum = 0.0;
for (int i = 0; i<n; i++){
sum = sum + F[i];
}
	float sum = 0.0; // Comment
	for (int i = 0; i<n; i++){
	___ sum = sum + F[i];
	}

Type-2	*Type-3*

int sum1 = 0; // Comment
for (int i = 0; i<n; i++){
___ sum1 = sum1 + F[i];
}
	int prod = 1; // Comment
	for (int i = 0; i<n; i++) {
	___ prod = prod * F[i];
	}

Type-4	

int factorial_rec (int n) {
if (n <= 1) {
return 1;
} else {
return n * factorial_rec (n - 1);
}
}
	int factorial_iterative(int n) {
	int result = 1;
	for (int i = 1; i <= n; ++i) {
	result *= i;
	}
	return result;
	}

As there are no comments and whitespaces in binary code, a slightly different definition for binary code clone types is used. Binary code clone types [5] are:

- **Type 1**: Two identical binary code fragments.
- **Type 2**: Two binary code fragments that can differ by registers, literals, and operand sizes. This type also includes Type 1 clones.
- **Type 3**: Two binary code fragments with additions, deletions, or modifications of instructions. Includes Type 2 clones too. Type 3 clones are also called non-exact clones.
- **Type 4**: Two binary fragments that have the same calculations but use different instructions.

TABLE I and TABLE II present examples of source and binary clone types, respectively. In both tables, original code and all clone types are presented.

TABLE II. EXAMPLE OF BINARY CODE CLONE TYPES.

Original code	*BinType-1*

mov [ebp+var_1], 5
mov eax, [ebp+var_1]
iadd eax, [ebp+var_4]
	mov [ebp+var_1], 5
	mov eax, [ebp+var_1]
	iadd eax, [ebp+var_4]

BinType-2	*BinType-3*

mov [ebp+var_1], 10
mov ecx, [ebp+var_1]
iadd ecx, [ebp+var_4]
	~~mov [ebp+var_1], 10~~
	mov ecx, [ebp+var_1]
	iadd ecx, [ebp+var_4]

BinType-4	

factorial_rec:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl %edi, -4(%rbp)
cmpl $1, -4(%rbp)
jg .L2
movl $1, %eax
jmp .L3
.L2:
movl -4(%rbp), %eax
subl $1, %eax
movl %eax, %edi
call factorial_rec
imull -4(%rbp), %eax
.L3:
ret
	factorial_O3:
	movl $1, %eax
	cmpl $1, %edi
	jle .L1
	.p2align 4,,10
	.p2align 3
	.L2:
	movl %edi, %edx
	subl $1, %edi
	imull %edx, %eax
	cmpl $1, %edi
	jne .L2
	.L1:
	ret

B. Program dependence graph

PDG is a directed graph that combines data and control dependencies. The vertices of PDGs are program statements and the edges are data and control dependencies between them. PDGs are used in various applications, such as compiler optimizations, program analysis, and software engineering tasks (like refactoring, debugging). As PDG makes explicit both the data and control dependencies between operations of the program, that makes it useful for understanding complex program behaviors and improving software quality and efficiency.

III. RELATED WORK

There are many works related to code clone detection. However, most of them can find only clones of a whole function. Our method deals with every fragment of code inside a function. Obviously, it also finds function clones.

Code clone detection techniques are divided into the following groups: text-based, token-based, tree-based, metrics-based, graph-based, and machine-learning based. Also, there are numerous hybrid methods combining several techniques for clone detection.

In the case of a text-based approach [6] [7] [8] [9], two code fragments are compared in the form of text/strings. It only finds Type 1 clones. In the case of a token-based approach [10] [11] [12] [13], the entire code is transformed into a sequence of tokens. It is more robust against code changes than text-based techniques, which allows it to find Type 1 and Type 2 clones.

Tree-based approaches [14] [15] [16] [17] use parse trees or Abstract Syntax Trees (AST) of the analyzable code. Then, similar subtrees are detected using tree-matching algorithms. It can find all three types of clones. But as a rule, this approach suffers in precision for Type 3 clone detection, because instructions difference strongly changes the underlying tree structure.

In the case of a metrics-based approach [18] [19] [20] [21], different types of metrics are calculated for code fragments. Then these metrics are compared to find similar code fragments. Usually, for calculating different types of metrics the code is converted into some graph representation, such as AST or PDG. This approach suffers in precision and produces many false positives.

In the case of a graph-based approach [22] [23] [24] [25], a PDG or just a Control Flow Graph (CFG) is generated from the code. Then maximal isomorphic or similar (it may be defined differently for each method) subgraphs are searched. PDG-based approaches are robust to the insertion and deletion of code, reordered instructions, intertwined and non-contiguous code. However, they have higher asymptotic complexity and may not be scalable.

In the case of machine learning-based techniques [26] [27] [28] [29], the focus is on training models to classify or cluster similar code fragments. Patterns are learned from a dataset containing examples of both similar and dissimilar codes. Learning algorithms are well-suited for code clone detection tasks because they can learn and identify complex patterns. However, learning-based techniques need large and clean datasets of code clones to work properly, but these are not available for all programming languages. Many methods rely on existing code clone detection tools to gather data for machine learning, but these tools are often unreliable and prone to errors.

In addition, there are hybrid methods, which combine several techniques for clone detection. Some examples are text-based and tree-based [30], token-based and tree-based [31], metric-based and graph-based [32], tree-based and learning-based [33] [34], etc. They addresses the challenge of individual methods.

Thus, each of the discussed techniques has its advantages and disadvantages. An appropriate method can be selected based on the problem that needs to be solved.

IV. Code Fragment Clone Detection

The developed algorithm takes a code fragment, a project, and a percentage of similarity as its input. It analyzes all the functions within the project and identifies clones of the specified fragment. The identified clones must have at least the specified percentage of similarity. It is important to note that we assume the provided code fragment is within a single function. Figure 1 provides architecture of the proposed method. It has two primary components: the construction of PDGs and the matching of these graphs.

A. Construction of PDGs

PDGs are constructed for the specified fragment and all functions of the target program. Vertices of the PDG represent instructions of Intermediate Representation (IR), and edges are constructed based on data and control dependencies between them. The construction process of PDGs varies for binary and source code as the code representation differs, and the specific details are outlined in the implementation section. For the vertices of the PDG, instead of "original form" instructions of IR are used, as it simplifies and standardizes the code, allowing tools to be reused across different languages and architectures.

To construct the PDG for the specified fragment, the PDG for the entire function containing the fragment is first created. Then, a subgraph corresponding to the specified fragment is extracted to serve as the final PDG of the fragment. Basically, it is the smallest induced subgraph of the entire function's PDG that includes all instructions of the specified fragment. For simplicity, we will call it a fragment's PDG. The constructed graphs are then utilized in the next step, where instructions from the specified fragments are matched against all instructions within the functions throughout the entire project.

B. Graphs' matching

Once the PDGs are constructed, the algorithm starts matching the vertices of the fragment's PDG with the vertices

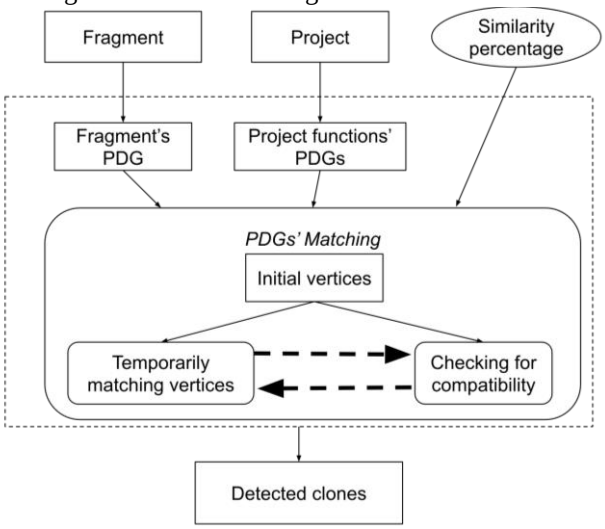

Figure 1. Architecture of the method.

of each function's PDG. It is important to note that within a single function's PDG, there can be detected multiple matches indicating the existence of several clones of the specified fragment within that function.

Similarity percentage for the detected fragment clone is calculated by the following formula:

$$similarity = \frac{matched\ common\ vertices\ count}{fragment\ PDG's\ vertices\ count} * 100\%$$

The matching algorithm between the fragment's and a function's PDGs involves the following phases:

1. Construction of the set of initial matched vertex pairs,
2. Iterative expansion of matched vertex pairs.

The first vertex of each pair is from the fragment's PDG and the second is from the function's PDG. Corresponding instructions for the vertices of each pair have the same operation code. The algorithm then selects one of the unconsidered pairs from the set to start expanding process. From the selected pair, the algorithm temporarily matches previously unmatched pairs of vertices using specific subroutines. These subroutines match vertices based on their features and adjacent edges, ensuring that vertices with identical operation codes are paired. If the temporarily matched vertices meet all specified conditions, they are finally matched. This process is repeated for all vertices that are not matched yet. The expanding phase stops when no new temporarily matched pairs can be identified. The output of this process is the list of sets, where each set contains matched vertex pairs. Further details will be provided later in the text. For simplicity, we will be using some notations that are described below:

- $function\_PDG$ - PDG of the given function,
- $fragment\_PDG$ - PDG of the given fragment,
- $initial\_pairs$ - the set of initial pairs of vertices $(v, v*)$, where $v \in fragment\_PDG$, $v* \in function\_PDG$,
- $temporarily\_matched\_pairs$ - the set of pairs $(v, v*)$, where $v \in fragment\_PDG$, $v* \in function\_PDG$, which are temporarily matched, but need to pass several checks before final matching,
- $matched\_pairs$ - the set of pairs $(v, v*)$, where $v \in fragment\_PDG$, $v* \in function\_PDG$, which are finally matched,
- $matched(G)$ – the set of finally matched vertices of graph G,
- $incompatible\_pairs$ - the set of $(v, v*)$ incompatible pairs of vertices, where $v \in fragment\_PDG$, $v* \in function\_PDG$,
- $opcode(v)$ - is an operation code corresponding to a vertex v,
- $pred\_ctrl(v)$/ $succ\_ctrl(v)$ - the set of predecessor / successor vertices of v by control dependence,
- $pred\_data(v)$ / $succ\_data(v)$ - the set of predecessor / successor vertices of v by data dependence,
- $bb(v)$ - the list of vertices in the same basic block as vertex v,
- $pred\_bb(v)$/ $succ\_bb(v)$ - the list of vertices in the predecessor / successor basic blocks of vertex v.

1) Construction of the set of initial matched vertex pairs.

The phase of selecting initial pairs of vertices aims to find such pairs of vertices from $fragment\_PDG$ and $function\_PDG$, which are likely to be matched together. Afterward, they are used as a starting point for the graphs' matching process. The amount of such vertices should be as small as possible for efficiency. To achieve this, the initial vertices in PDGs are selected using various subroutines, chosen based on their effectiveness during the experimental evaluation.

The first subroutine selects all vertices $(v, v*)$ with no incoming edges in both PDGs, where $v \in fragment\_PDG$ and $v* \in function\_PDG$. These vertices typically correspond to the first instructions of the specified fragment and the function. Then, from the obtained sets of vertices, the subroutine constructs all possible combinations of pairs, where the corresponding instructions have the same operation code, and adds them to $initial\_pairs$.

The second subroutine collects vertices with the maximum incoming data dependencies in $fragment\_PDG$. Then it collects vertices from $function\_PDG$ that have an equal or greater number of incoming data dependencies. Like the first subroutine, this one also creates all possible combinations of pairs from the obtained sets (ensuring that the corresponding instructions have the same operation code) and adds them to $initial\_pairs$ set.

The third subroutine identifies all the instructions from the code fragment that have the maximum number of corresponding IR instructions. It then selects instructions from the function with the same number of corresponding IR instructions. Subsequently, the subroutine collects vertices corresponding to the first IR instructions of the mentioned instructions. Finally, similar to other subroutines, it generates all possible combinations of pairs from the obtained sets, ensuring that the corresponding instructions have the same operation code, and adds them to $initial\_pairs$ set.

2) Iterative expansion of matched vertex pairs.

The expanding phase temporarily matches unconsidered vertices from $fragment\_PDG$ and the $function\_PDG$. Next, it checks temporarily matched vertices for conditions. If a pair passes conditions checking, it is placed to $matched\_pairs$ list, otherwise it is placed to $incompatible\_pairs$ list. Expanding starts from $initial\_pairs$ and iteratively matches vertices until no temporarily matched vertices can be detected.

a) Temporarily matching.

The matching algorithm involves five temporary matching subroutines. The results obtained from these subroutines are then checked against several conditions (described in the next section), and some of the temporarily matched pairs may be filtered out. The matching process is complete when no new pairs of vertices are temporarily matched, meaning that the algorithm has exhausted all possible matches between the fragment's PDG and the function's PDG.

For each pair of vertices $(u, u*)$ temporary matching is allowed if $opcode(u) == opcode(u*)$, the size of $pred\_ctrl(u)$ equals to the size of $pred\_ctrl(u*)$, and the size of $succ\_ctrl(u)$ equals to the size of $succ\_ctrl(u*)$, where $(u, u*) \notin matched\_pairs$ and $(u, u*) \notin incompatible\_pairs$.

In all subroutines, two vertices $(v, v*)$ can be temporarily matched if $(v, v*) \notin matched\_pairs$, $(v, v*) \notin incompatible\_pairs$ and corresponding instructions have the same opcode. The subroutines are applied in the specific order, and if one of them temporarily matches a pair, the others will not be applied. At the beginning $temporarily\_matched\_pairs \leftarrow \emptyset$. Below are descriptions of five temporarily matching subroutines:

1. For each pair $(v, v*) \in matched\_pairs$ temporarily match vertices $(u, u*)$, where $u \in pred\_ctrl(v)$ and $u* \in pred\_ctrl(v*)$ sets, and add them to $temporarily\_matched\_pairs$ if temporary matching is allowed. Do the same for vertices $(u, u*)$ from $succ\_ctrl(v)$ and $succ\_ctrl(v*)$ sets. If $temporarily\_matched\_pairs$ is not empty, go to conditions checking phase.
2. For each pair $(v, v*) \in matched\_pairs$ temporarily match vertices $(u, u*)$, where $u \in bb(v)$ and $u* \in bb(v*)$ lists, and add them to $temporarily\_matched\_pairs$ if temporary matching is allowed. If $temporarily\_matched\_pairs$ is not empty, go to conditions checking phase.
3. For each pair $(v, v*) \in matched\_pairs$ temporarily match vertices $(u, u*)$, where $u \in pred\_bb(v)$ and $u* \in pred\_bb(v*)$ lists, and add them to $temporarily\_matched\_pairs$ if temporary matching is allowed. Do the same for vertices from $succ\_bb(v)$ and $succ\_bb(v*)$. If $temporarily\_matched\_pairs$ is not empty, go to conditions checking phase.
4. For each pair $(v, v*) \in matched\_pairs$ temporarily match vertices $(u, u*)$, where $u \in pred\_data(v)$ and $u* \in pred\_data(v*)$ sets, and add to $temporarily\_matched\_pairs$ if temporary matching is allowed. Do the same for vertices from $succ\_data(v)$ and $succ\_data(v*)$ sets. If $temporarily\_matched\_pairs$ is not empty, go to conditions checking phase.
5. Temporarily match pairs $(u, u*) \in initial\_pairs$, and add to $temporarily\_matched\_pairs$, if $(u, u*) \notin matched\_pairs$ and $(u, u*) \notin incompatible\_pairs$.

b) Conditions checking.

The next stage is the checking of temporarily matched pairs. After each iteration of temporarily matching, each pair $(v, v*) \in temporarily\_matched\_pairs$ is checked for conditions. If the pair satisfies all conditions, it is moved to $matched\_pairs$, otherwise to $incompatible\_pairs$. The conditions are described below:

1. $pred\_condition(v, v*)$ returns $false$ if $\exists p \in pred\_ctrl(v)$ where $p \in matched(fragment\_PDG)$ and $\nexists p* \in function\_PDG$ such that $p* \in pred\_ctrl(v*)$ and $(p, p*) \in matched\_pairs$, otherwise returns $true$.
2. $succ\_condition(v, v*)$ returns $false$ if $\exists s \in succ\_ctrl(v)$ where $s \in matched(fragment\_PDG)$ and $\nexists s* \in function\_PDG$ such that $s* \in succ\_ctrl(v*)$ and $(s, s*) \in matched\_pairs$, otherwise returns $true$.

V. IMPLEMENTATION

We implemented the proposed method in a tool called FCD. It is a command-line tool, that receives the following inputs:

1. The project path and the function name containing the code fragment to be analyzed,
2. The boundaries of the code fragment: the start and end line numbers for source code, the start and end memory relative addresses for binary code,
3. The project in which to search for clones of the specified fragment,
4. An optional minimum similarity percentage parameter, which is used to filter out clones that are less similar than the specified value. This parameter belongs to (0, 100], and has a default value of 90. The 90% similarity is chosen to detect highly similar code fragments, which is more of the interest to developers.

The process of PDG's generation differs for source and binary code, however, the matching parts are the same. For source code PDG's generation FCD uses LLVM intermediate representation [35]. To get PDGs for source code a new pass is added in LLVM, which uses control flow information, use-def chains and alias analysis. For binary code PDGs generation FCD uses REIL [36] intermediate representation. At first, it uses IDA Pro [37] disassembler to restore assembler and control flow graphs. Then the obtained assembler is translated to the REIL intermediate language using Binnavi [38]. Lastly, it uses Binside [39] to generate PDGs, which was developed by our team previously.

Code fragment clone detection algorithm is implemented in C++ language. The output of the tool consists of a set of JSON files containing information about the detected clones. This information includes functions' names corresponding to matched fragments, similarity percentage, all pairs of matched instructions, and other relevant details.

VI. TESTING SYSTEM

To evaluate FCD algorithm, we have designed and implemented a testing system, which generates tests, executes FCD and calculates precision, recall, and Root Mean Square Error (RMSE) to assess their effectiveness. Test generation is done using PDGs of real-world projects. For each PDG, it creates a duplicate, removes some vertices, and considers it as fragment's PDG. It randomly selects a basic block and removes corresponding vertices until the desired similarity percentage is reached. After removing a vertex, its predecessor vertices are connected with the successor ones. If all vertices in the chosen basic block are removed and the provided similarity is still not met, the system randomly selects a new basic block and starts removing consecutive vertices from that block. This process continues until the required similarity percentage is not met.

It then runs the FCD algorithm on generated PDGs' pairs and compares the resulting similarity percentage with the one specified to testing system. Ideally, the similarity percentages of the created PDGs' pairs by the testing system should match with the results from the FCD algorithm. The testing system saves information about the correspondence of the original and the generated PDG vertices, which is used to calculate precision, recall, and RMSE.

VII. RESULTS

FCD is tested with the discussed testing system on projects OpenSSL, JasPer, c-ares, Rsync. Tables TABLE *III* and TABLE *IV* present the results of source and binary code clone detection, respectively. The results are averaged across similarity thresholds 100%, 90%, 80%, and 70%.

The tool achieves perfect results when generated clones are 100% similar. Furthermore, FCD consistently demonstrated high accuracy across lower thresholds, as reflected in the averaged results in the tables. However, binary code clone detection's speed is slow compared to source code clone's detection time, as for binary bigger PDG's are generated.

TABLE III. SOURCE CODE CLONE RESULTS

Project	*C/C++ code lines*	*Precision*	*Recall*	*RMSE*	*FCD speed*
c-ares 1.15.0	61087	97.5	95.2	6.1	0m 0.29s
jasper 1.900.1	28279	95.4	93	6	0m 15s
openssl 1.0.2t	310922	97	95.1	7.7	0m 2s
rsync 3.1.3	44832	96	91.9	10.7	0m 26s

On average, FCD has 96.5% precision, 93.8% recall and 7.6% RMSE for source code. And on average, FCD has 96.9% precision, 92.9% recall and 5.4% RMSE for binary code. Despite high rates of the tool's precision and recall, there are still certain cases that the tool may not detect correctly. This occurs when the copied code is modified by adding a new instruction between each original instruction, i.e., one instruction from the original code, followed by one new instruction, then another from the original, and so on. However, if the copied code is modified in such a way that a whole basic block is added the tool identifies it correctly.

TABLE IV. BINARY CODE CLONE RESULTS

Project	*Size of the binary*	*Architecture*	*Precision*	*Recall*	*RMSE*	*FCD speed*
libcares 2.3.0 (c-ares 1.15.0)	86 KiB	x86-64	98.9	95.6	4.6	0m 41s
libcares 2.3.0 (c-ares 1.15.0)	96 KiB	x86	97.9	93.4	5.5	0m 43s
libcares 2.3.0 (c-ares 1.15.0)	146 KiB	ARM	98.9	95.6	4.6	0m 49s
jasper 1.900.1	1.5 MiB	x86-64	96	92.1	5.4	3m 5s
jasper 1.900.1	368 KiB	x86	95	90	6.5	2m 1s
jasper 1.900.1	478 KiB	ARM	94.1	89.8	6.1	2m 8s
openssl 1.0.2t	536 KiB	x86-64	99.9	98.1	3.8	1m 10s
openssl 1.0.2t	507 KiB	x86	98.8	95.8	3.9	0m 57s
openssl 1.0.2t	634 KiB	ARM	97.9	95.6	4.4	1m 25s
rsync 1.3.2	1.7 MiB	x86-64	96	91	6.6	3m 34s
rsync 1.3.2	1.6 MiB	x86	94.9	88.9	6.7	3m 21s
rsync 1.3.2	1.8 MiB	ARM	94.1	88.8	7.4	3m 58

The tool is not compared with the related tools as there is no common benchmark for evaluation. While there are some benchmarks available for C/C++ languages, they include only Type-4 clones, which our tool does not detect. Additionally, each tool uses its own method to calculate similarity levels, which results in inconsistent evaluations of the same code fragments.

VIII. CONCLUSION

The study proposes a novel technique to identify duplicated code fragments. It overcomes limitations of existing clone detection tools, which typically target only full functions and specialize in either source or binary code analysis. Experimental evaluation on real-world software projects demonstrates the high precision and effectiveness of the proposed clone detection approach for source and binary code. As conclusion we can clearly see that PDG captures enough information for source and binary code to enable accurate clone detection for both cases. Moreover, a unified algorithm can be used for maximal similar subgraphs detection in both cases.

ACKNOWLEDGMENT

The work was supported by the Science Committee of RA, in the frames of the research project 21SCG-1B003.

REFERENCES

[1] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software systems," in *Proceedings of the 15th Working Conference on Reverse Engineering*, 2008, pp. 81-90.

[2] "Synopsis," 2024 Open Source Security and Risk Analysis Report, [Online]. Available: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2024.pdf. [retrieved: 08.2024].

[3] R. Koschke and S. Bazrafshan, "Software-clone rates in open-source programs written in c or c++," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 3, pp. 1-7, 2016.

[4] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470-495, 2009.

[5] H. K. Aslanyan, "Effective and Accurate Binary Clone Detection," *Mathematical Problems of Computer Science*, vol. 48, pp. 64-73, 2017.

[6] D. Tukaram and U. Maheswari B, "Design and development of software tool for code clone search, detection, and analysis," in *2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA), pp. 1002-1006*, 2019.

[7] C. Ragkhitwetsagul and J. Krinke, "Using compilation/decompilation to enhance clone detection," in *2017 IEEE 11th International Workshop on Software Clones (IWSC), IEEE, pp. 1–7*, 2017.

[8] T. Kamiya, "An execution-semantic and content-and-context-based code-clone detection and analysis," in *2015 IEEE 9th International Workshop on Software Clones, IWSC 2015 - Proceedings, pp. 1–7*, 2015.

[9] J. Chen, M. H. Alalfi, T. R. Dean, and Y. Zou, "Detecting Android malware using clone detection," *Journal of Computer Science and Technology*, vol. 30, pp. 942-956, 2015.

[10] L. Yang, Y. Ren, J. Guan, B. Li, and J. Ma, "FastDCF: a partial index based distributed and scalable near-miss code clone detection," in *Parallel and Distributed Computing, Applications and Technologies: 22nd International Conference, PDCAT 2021, pp. 210-222*, Guangzhou, China, 2021.

[11] Y.-L. Hung and S. Takada, "CPPCD: a token-based approach to detecting potential clones," in *IEEE 14th International Workshop on Software Clones (IWSC), IEEE, pp. 26–32*, 2020.

[12] Y. Wu et al., "SCDetector: software functional clone detection based on semantic tokens analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, , pp. 821–833*, New York, NY, USA: ACM, 2020.

[13] K. E. Rajakumari, "Comparison of token-based code clone method with pattern mining technique and traditional," in *Proceedings of 2019 3rd IEEE International Conference on Electrical, Computer and Communication Technologies, ICECCT 2019, pp. 1–6*, 2019.

[14] Y. Yu, Z. Huang, and G. Shen, "ASTENS-BWA: searching partial syntactic similar regions between source code fragments via," *Science of Computer Programming*, vol. 222, p. 102839, 2022.

[15] W. Wen et. al., "Cross-project software defect prediction based on class code similarity," *IEEE Access*, vol. 10, p. 105485–105495, 2022.

[16] Y. Gao et al., "TECCD: A Tree Embedding Approach for Code Clone Detection," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, pp. 145–156*, 2019.

[17] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD : Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[18] S. Parsa, M. Zakeri-Nasrabadi, and M. Ekht, "Method name recommendation based on source code metrics," *Journal of Computer Languages*, vol. 74, no. 101177, pp. 1-13, 2023.

[19] H. Jin, Z. Cui, S. Liu, and L. Zheng, "Improving code clone detection accuracy and efficiency based on code complexity analysis," in *n 2022 9th International Conference on Dependable Systems and Their Applications (DSA), IEEE, pp. 64–72*, 2022.

[20] 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," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, pp. 1026–1037*, 2019.

[21] M. Sudhamani and L. Rangarajan, "Code similarity detection through control statement and program features," *Expert Systems with Applications*, vol. 132, no. 15, pp. 63-75, 2019.

[22] W. Wen et al., "Cross-project software defect prediction based on class code similarity," *IEEE Access*, vol. 10, pp. 105485-105495, 2022.

[23] H. K. Aslanyan, S. F. Kurmangaleev, V. G. Vardanya, M. S. Arutunian, and S. S. Sargsyan, "Platform-independent and scalable tool for binary code clone detection," in *Proceedings of the Institute for System Programming of the RAS, pp. 215-226*, 2016.

[24] Z. Xue et al., "SEED: semantic graph based deep detection for type-4 clones," in *International Conference on Software and Software Reuse, pp. 120–137*, 2022.

[25] N. Mehrotra et al., "Modeling functional similarity in source code with graph-based Siamese networks," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3771-3789, 2022.

[26] A. Zhang et al., "Learn to align: a code alignment network for code clone detection," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC), pp. 1-11*, 2021.

[27] N. D. Q. Bui, Y. Yu, and L. Jiang, "InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1186-1197*, 2021.

[28] Y. Li, C. Yu, and Y. Cui, "TPCaps: a framework for code clone detection and localization based on improved CapsNet," *Applied Intelligence*, vol. 53, p. 16594–16605, 2022.

[29] S. Patel and R. Sinha, "Combining holistic source code representation with siamese neural networks for detecting code clones," in *IFIP International Conference on Testing Software and Systems, pp. 148–159*, 2022.

[30] A. Schafer, W. Amme, and T. S. Heinze, "Stubber: compiling source code into bytecode without dependencies for Java code clone detection," in *2021 IEEE 15th International Workshop on Software Clones (IWSC), IEEE, pp. 29-35*, Oct. 2021.

[31] W. Wang, Z. Deng, Y. Xue, and Y. Xu, "CCStokener: Fast yet accurate code clone detection with semantic token," *Journal of Systems and Software*, vol. 199, p. 111618, May 2023.

[32] H. Aslanyan et al., "Scalable Framework for Accurate Binary Code Comparison," in *2017 Ivannikov ISPRAS Open Conference (ISPRAS), pp. 34-38*, 2017.

[33] A. Zhang, L. Fang, C. Ge, P. Li, and Z. Liu, "Efficient transformer with code token learner for code clone detection," *Journal of Systems and Software*, vol. 197, p. 111557, Mar. 2023.

[34] Y. Wu, S. Feng, D. Zou, and H. Jin, "Detecting semantic code clones by building AST-based Markov chains model," in *37th IEEE/ACM*

International Conference on Automated Software Engineering, pp. 1-13, New York, NY, USA, Oct. 2022.

[35] "The LLVM Compiler Infrastructure," [Online]. Available: www.llvm.org. [retrieved: 08.2024].

[36] "REIL - The Reverse Engineering Intermediate Language. Zynamics," [Online]. Available: https://www.zynamics.com/binnavi/manual/html/reil_language.htm. [retrieved: 08.2024].

[37] "IDA Pro," [Online]. Available: https://hex-rays.com/ida-pro/. [retrieved: 08.2024].

[38] "BinNavi," [Online]. Available: https://www.zynamics.com/binnavi.html. [retrieved: 08.2024].

[39] H. Aslanyan, M. Arutunian, G. Keropyan, S. Kurmangaleev, and V. Vardanyan, "BinSide : Static Analysis Framework for Defects Detection in Binary Code," in *2020 Ivannikov Memorial Workshop (IVMEM), pp. 9-14*, Orel, Russia, 2020.