Detection of Similar Functions Through the Use of Dominator Information

André Schäfer
Institute of Computer Science
Friedrich Schiller University Jena
Jena, Germany
Andre.Schaefer@uni-jena.de

Wolfram Amme
Institute of Computer Science
Friedrich Schiller University Jena
Jena, Germany
Wolfram.Amme@uni-jena.de

Thomas S. Heinze
Institute of Data Science
German Aerospace Center (DLR)
Jena, Germany
Thomas.Heinze@dlr.de

***Abstract*—The detection of code clones is an important technique for finding malware and malicious code. Many existing methods work on source code to detect code clones. Recent work in this area has started to focus on the analysis of compiled code to find malicious code. We introduce a new method to detect code clones using Java bytecode and control flow information from dominator trees. A prototype implementation was developed and compared with the state-of-the-art clone detector NiCad to evaluate the basic functionality of the method. First experiments have shown that the method can reliably find Type 1 and Type 2 clones and even find additional clones.**

***Index Terms*—Clone Detection, Malware Detection, Dominator Tree, Control Flow, Path Embedding**

I. Introduction

Malicious code and malware have become a striking problem for the security of today's computer systems. Effectively protecting systems against compromising attacks based on malware is therefore of high importance. The huge numbers of malware and malicious code are partially due to the adoption of code reuse schemes in the malware industry and a large fraction of its produced malicious programs seem to be mere modifications of previous ones [1], [2]. Identifying malware using static or dynamic analysis often requires reverse engineering and an in-depth understanding of the behavior of the suspicious code. Considering instead known malicious code as malware patterns, which can be used to check systems for unknown malware, therefore seems to be a promising approach. *Code clone detection* [3] is an actively researched technique for automatically identifying duplicated or similar code and is used commonly for applications like code refactoring, license violation detection, or bug and vulnerability detection. Applying these techniques to detect malicious code allows for protecting against malware-based threats [2], [4].

In this paper, a novel code clone detection method for identifying clones of malware and malicious code is introduced. The method is based upon encoding pieces of code according to their paths in the *dominator tree*, a compiler-style code representation, which is more abstract compared to the raw code, tokens, or abstract syntax trees, as usually used by code clone detection methods, and therefore achieves better results in the detection of code clones with weaker syntactical similarity. Even more, the path-based encoding supports detecting clones at various code granularities, not only at the function or method level. In a preliminary evaluation, the feasibility of the approach is studied. We therefore identified and labeled very similar code clones (Type 1/2 clones [3]) in an open-source Java production library using the state-of-the-art clone detector *NiCad* [5] and then applied our method to the library. We found that the method is able to find the same code clones as NiCad with almost perfect recall. In addition, we also found more code clones (Type 3/4 clones [3]), which are more difficult to find due to their large syntactical differences.

The rest of the paper is structured as follows: In Sect. II, we first discuss the different types of code clones and the existing work on code clone detection. Our approach based upon paths in the dominator tree is conceptually introduced in Sect. III, where we provide details on four versions, with or without path splitting and path abstraction. In Sect. IV, we present a prototypical implementation and the results of a preliminary evaluation. Finally, in Sect V, we conclude the paper.

II. Code Clones

A. Definition and Classification

A *code fragment* is a continuous piece of code, usually certain lines of code. Code fragments can differ in their granularity, so that a code fragment can be a sequence of statements, a block, or a function. *Code clones* are two code fragments, which share a degree of similarity according to a certain definition, usually derived from the fragments' syntax or semantics. Two similar code fragments then form a *clone pair*. If more than two code fragments are involved, the fragments form a *clone class*. In the literature, there can be found four types of code clones, which differ in the kind and degree of similarity [3]:

- *Type 1:* Code fragments which only differ in whitespace, layout, and comments but are otherwise identical.
- *Type 2*: Code fragments which in addition to Type 1 code clones differ in identifiers, types, and constants.
- *Type 3:* Code fragments which in addition to Type 2 code clones have statements inserted, altered, or deleted.

DOI 10.1109/ACSOS-C51401.2020.00057

- *Type 4:* Code fragments which differ in their syntax but have similar semantics or perform similar computations.

While the first approaches to code clone detection focused on Type 1 and Type 2 code clones, research has progressed towards identifying Type 3 and Type 4. Therefore, Type 3 and Type 4 code clones have been further classified into four types according to the degree of their syntactical similarity, i.e., ratio of shared lines of code or tokens after normalizing code fragments for Type 1 and Type 2 [6]:

- *Very-Strongly Type 3:* Code fragments with more than 90% syntactical similarity.
- *Strongly Type 3:* Code fragments with syntactical similarity between 70% and 90%.
- *Moderately Type 3:* Code fragments with syntactical similarity between 50% and 70%.
- *Weakly Type 3/Type 4:* Code fragments with less than 50% syntactical similarity.

B. Code Clone Detection

Code clone detection has to identify pieces of code with high similarity in a given system's code. Numerous methods and tools have been proposed for solving this task. According to ROY ET AL. [3], the methods and tools for code clone detection can be classified according to the used information into *textual*, *lexical*, *syntactical*, and *semantic* approaches:

1) Textual Methods: Textual methods use the raw source code with little or no normalization for clone detection. JOHNSON, as early as 1993/94, used a sliding window and fingerprints, i.e., hashes, on code substrings with fixed length for finding clones [7], [8]. Textual methods are though also used today due to their efficiency. As an example, *Vuddy* allows for the scalable detection of vulnerable code clones, looking up function fingerprints, consisting of the functions length and hash, in a central vulnerability repository [9]. Cybersecurity is also the motivation in [4], where *NiCad* [5], [10] is used as a code clone detector for identifying malware in Android apps. NiCad applies a mostly textual method, but additionally employs lightweight parsing and tree-based structural analysis for source code normalization and filtering. NiCad is a state-of-the-art tool and detects most code clones of Type 1 and 2 and part of Type 3 in practice at various granularities and customizable syntactical similarity [6]. NiCad is used as baseline for the preliminary evaluation in Sect. IV.

2) Lexical Methods: In order to deal with minor code changes like formatting and variable renaming, lexical methods apply compiler-style lexical analysis to transform source code into a sequence of tokens which is then used to detect code clones. An early example is *CCFinder* [11], which applies token-based normalizations and suffix trees for finding similar pieces of code. Frequent subsequence data mining is used on token sequences in *CP-Miner* to achieve the same goal [12]. A recent token-based tool, *SourcererCC*, employs prefix filtering to reduce the number of code pair candidates which need to be compared, thus increasing scalability of detecting up to Type 3 clones on large code repositories with similar sensitivity as *NiCad* [13]. Scalability is further improved using adaptive prefix filtering by NISHI AND DAMEVSKI [14]. Finding large-gap code clones, i.e., code clones with many Type 3 statement insertions/deletions, is the specific feature of *CCAligner* [15]. Token-based clone detection for buggy or vulnerable code is addressed in [16], based on measuring compression ratios of clone pairs.

3) Syntactical Methods: Syntactical methods parse source code to transform it into abstract syntax trees (ASTs), such that code clones can be detected via tree matching or fingerprints, i.e., structural metrics. The pioneering tree-matching tool *CloneDR* uses a compiler generator to create an AST generator, whose ASTs are hashed into buckets. Only pairs of the same bucket are then compared by a tolerant tree matching algorithm, thus reducing tree comparisons [17]. Instead of matching trees, *Deckard* maps ASTs to vectors, capturing the syntactical information, and clusters them using their Euclidean distances for detecting code clones at varying granularities [18]. Recent approaches use deep neural networks for learning how to embed ASTs into vector spaces and select the vectors' features to predict code clones. As an example, *Oreo* works on the function level and uses a Siamese neural net architecture and training data labeled with the *SourcererCC* clone detector, achieving in particular good results for Moderately Type 3 clones and beyond [19]. Type 3/4 code clones are also the focus of other deep learning approaches to clone detection [20]–[22]. In the tool *CCLearner*, the frequencies of eight token categories in an AST and their similarity in clone pairs are used as features for supervised learning of a clone detector [20]. *Yu et al.* use convolutional networks and combine structural information from ASTs with lexical information from tokens in their embedding [21]. AST embeddings are in particular studied in [22], using the cosine similarity of vectors representing code clones for evaluation. As a result, the authors showed the usefulness of pretrained embeddings like *code2vec* [23]. A thorough comparison of our approach with syntactical methods is subject to future work.

4) Semantic Methods: Instead of comparing syntactical features, semantic methods analyze the computations of code too also find Type 4 code clones. PEWNY ET AL. use the control flow graph and hashed signatures of its basic blocks, as created by statically analyzing their input/output behavior, to find buggy or vulnerable code clones in binary code across architectures [24]. Similarly, semantic signatures are extracted for functions by emulating their execution in [25], to detect cloned functions in binary code. Instrumentation is used in [26] to create dynamic instruction graphs, which are searched for isomorphic subgraphs to find function clones. While they capture runtime information, they may miss similar code because of limited coverage of used test inputs. Recent approaches embed a representation of code, which models semantics, e.g., control/data flow, into a neural net and solve the clone detection problem via deep learning [27], [28].

III. Clone Detection using Dominator Trees

The method we propose is a path-oriented technique in which functions are described by control flow. In contrast to known techniques, in our method, functions are not represented by control flow graphs, but by a more specialized format, the dominator tree of a function. A dominator tree is often used in compilers to describe the execution order of instructions. The use of dominator trees has the advantage that, unlike in control flow graphs, each node in such a tree can be reached exactly via a single and unambiguous path.

A. Basic Methodology

A dominator tree of a program describes a special property of its control flow, namely, which instructions are executed before others each time a program is executed. In the following, let $G = (N, E, s)$ be a control flow graph, where (N, E) describes a directed graph and $s \in N$ stands for its start node. In the control flow graphs used here, nodes are represented by single instructions instead of basic blocks. Mapping of instructions can be specified through a function $map : N \rightarrow Inst$ that assigns a program instruction $s \in Inst$ to each node. Furthermore, let $l, k \in N$. Then, a node l dominates a node k in G (l dom k), if all execution paths starting with the start node s and ending in k are using the node l. A node l strictly dominates k in G, if l dom k in G and $l \neq k$. A node l dominates a node k immediately in G, if l strictly dominates k and there exists no other strictly dominator of k that is strictly dominated from l.

A dominator tree of G then is formally defined as a directed graph $D^G = (N, E^*)$, where $E^* = \{(l, k) \mid l$ immediately dominates k in G$\}$. Figure 1 shows a Java method and its simplified dominator tree, generated by the Soot compiler [31] based on the method's bytecode. In the shown code representation, the variables $p0$ and $p1$ should be initialized with the receiver object and the transferred parameter value, respectively. As can be seen, each node is reachable from the start node via exactly one path. It should also be pointed out that, unlike in control flow graphs, no backward edge is inserted into the dominator tree for the method's for-loop, because the instructions in the loop body do not dominate the loop condition.

The use of a pairwise matching algorithms on dominator trees to find similar functions would be, as with techniques that use such algorithms for control flow graphs [24], inefficient and lead to a high runtime behavior [27].[1] For this reason, we move away from a direct comparison of dominator trees. Instead, descriptive patterns – a kind of fingerprint – are derived for dominator trees and their comparison.

Finding Similar Functions and Subfunctions: The basic idea of our methodology is to derive a set of paths for each dominator tree, which can then be used as pattern for the comparison of the dominator trees. This concept is comparable to the work of Alon et al. [23], who use paths that connect the leaves of abstract syntax trees to build patterns for the description of the abstract syntax trees. For the description of a dominator tree D, all paths contained in D that start with the start node and end in a leaf node are thus used. Patterns created in this way can then be used to uniquely represent a dominator tree. Paths are described in encoded form as a concatenation of the instructions assigned to the nodes. For example, in the dominator tree of Figure 1, the path from the start node to the rightmost leaf node is described as:

$assign(i1 = 0) \rightarrow assign(i0 = p1.Array.size)$
$\rightarrow cond(i1 >= i0) \rightarrow return()$

Let $G = (N, E, s)$ be a control flow graph and D^G its dominator tree, P_{D^G} the set of paths in D^G, and L_{D^G} the set of leaf nodes in D^G. The set of paths uniquely describing the dominator tree, that is used as a pattern for D^G, is defined by $D^G_{set} = \{map(s) \circ \cdots \circ map(n_k) | s \cdots n_k \in P_{D^G}, n_k \in L_{D^G}\}$. For the dominator tree D of the sample program in Figure 1, the set of paths D_{set} is given in Figure 2. In the figure, a constant pool is used to describe the accesses to functions and instance variables. In the paths, a constant pool access is identified by the character # and the entry number.

Two functions f and g, with control flow graphs G and G', respectively, are then called strictly *similar* to each other, if the cardinality of the sets D^G_{set} and $D^{G'}_{set}$ is the same and if for each path $p \in D^G_{set}$ exists also a similar path $p' \in D^{G'}_{set}$. If only the latter condition applies and the sets of paths D^G_{set}

```
void restoreExpandedValues(Array<V> values){
    for(int i=0,n=values.size;i<n;i++){
        node=findNode(values.get(i));
        if (node!=null){
            node.setExpanded(true);
            node.expandTo();
        }
    }
}
```

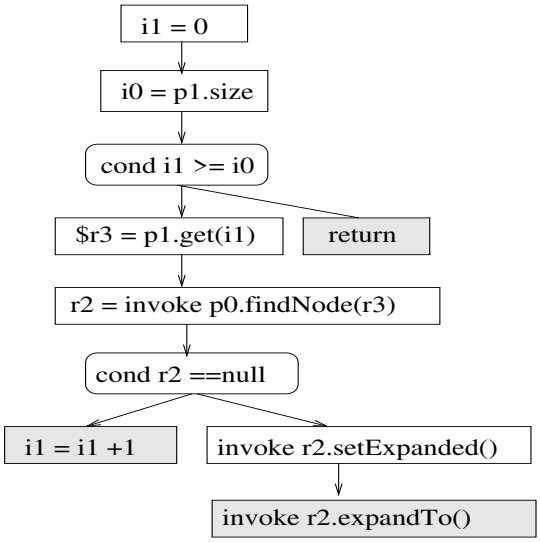


Fig. 1. Sample progam and its dominator tree.

[1] For example, the use of the Hungarian Method would lead to a complexity of O($|E^3|$) for a graph $G(N, E)$ [29].

0:	Array.size
1:	Array.get(int)
2:	N.findNode(V)
3:	N.setExpanded(1)
4:	N.expandTo(V)

Constant Pool

D_{set} = {assign(i1=0)→assign(i0=p1.#0)→
cond(i1>=i0)→assign($r3=p1.#1)→
assign(r2=p0.#2)→cond(r2==null)→
assign(i1=i1+1),
assign(i1=0)→assign(i0=p1.#0)→
cond(i1>=i0)→assign($r3=p1.#1)→
assign(r2=p0.#2)→ cond(r2==null)→
invoke(r2.#3)→invoke(r2.#4),
assign(i1=0)→assign(i0=p1.#0)→
cond(i1>=i0)→return()}

D_{split} = {assign(i1=0)→assign(i0=p1.#0),
cond(i1>=i0)→assign($r3=p1.#1)→assign(r2=p0.#2),
cond(i1>=i0)→return(),
cond(r2==null)→assign(i1=i1+1),
cond(r2==null)→invoke(r2.#3)→invoke(r2.#4)}

Fig. 2. Patterns of the sample dominator tree.

and $D^{G'}_{set}$ thus do not have the same number of elements, then f and g are called only *partially similar*.

The similarity of paths in dominator trees D^G and $D^{G'}$ can be formally described by a function $\sigma : D^G_{set} \times D^{G'}_{set} \rightarrow bool$. In general, the function σ can be freely chosen depending on its application. For our purposes, two paths p and q are considered to be similar, and thus code clones, if the euclidean distance of p and q does not exceed a certain threshold value.

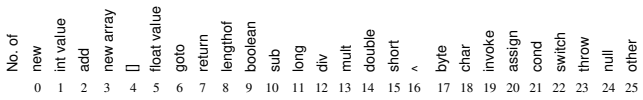
Discovering Similar Subblocks: With the methodology we have introduced so far, the similarity of functions or subfunctions f and g can be analyzed. On the other hand, the technique cannot yet be used to check whether a function f contains certain subblocks, which is, however, indispensable for finding malicious code and malware.

The reason for not finding independent subblocks in functions is due to the way, the paths in the patterns have been chosen for describing dominator trees. Since these always start with the start node and end in a leaf node, similiar subblocks located in the middle of a program, which are characterized as independent of the overall execution of a function, i.e., by the fact that they have similar partial paths, cannot be recognized.

A more suitable structure of the patterns, used for the description of dominator trees when searching for similar subblocks, can be achieved by introducing split nodes. We call a node n in a dominator tree D^G a split node, if n has more than one successor node in D^G. The split nodes of dominator tree D^G are referred to as S_{D^G}. In the dominator tree of the sample program, split nodes are represented by comparison nodes, which are shown in Figure 1 in the form of ellipses.

Through the introduction of split nodes, the set of paths that can used as a pattern of $D^G = (N, E^*)$ is defined by the set of all paths that begin with the start node s or a split node, and end at a leaf node or the predecessor node of a split node, and contain no other split node apart from the initial split node. Formally described:

$$D^G_{split} = \{map(n_1) \circ \cdots \circ map(n_k) \,|\, n_1 \cdots n_k \in P_{D^G}, (n_1 = s \vee n_1 \in S_{D^G}), (n_k \in L_{D^G} \vee n_k \in preds(S_{D^G}), \nexists i > 1 : n_i \in S_{D^G})\}$$



D_{set} = {[0.1.1.0.0.0.1.0.$\cdots$ 0.5.2.0.0.0.1.1.],
[0.1.0.0.0.0.0.0.$\cdots$ 2.4.2.0.0.0.1.1.],
[0.0.0.0.0.0.0.1.$\cdots$ 0.2.1.0.0.0.0.0.]}

Fig. 3. Path abstraction of D_{set} of the sample dominator tree.

The set of paths using split nodes for the dominator tree of the sample program is also shown in Figure 2.

As before, for a subblock b and a function f, with control flow graphs G and G', respectively, function f contains a subblock that is *strictly similiar* to subblock b, if for each path $p \in D^G_{split}$ a similar path $p' \in D^{G'}_{split}$ exists. If the latter condition holds for a subset of D^G_{split}, function f contains a subblock that is *partially similar* to subblock b.

B. Abstraction of the Path Representations

The description of paths by a sequence of node labels is efficient, but often not flexible enough for recognizing similar paths. For example, it is difficult to recognize two similar paths, in which computationally independent instructions are executed in exactly the opposite order. Furthermore, it is difficult to identifiy a path as similar to a path, in which additional instructions are executed without changing the semantics.

In principle, recognition of similar paths, in cases like described above, could be achieved by loosening the comparison function σ. However, a weakening of the comparison function could lead to the fact that previous inequalities can become equal. There are various approaches to solve this problem. One possibility would be, for example, to describe paths in a dominator tree using multisets [30] and define the similarity of different paths by means of a subset relation on these sets. However, such a representation of paths would be inefficient from an algorithmic point of view. Another solution could be to use machine learning techniques to train a neural network to predict when one path is a subpath of another.

The use of machine learning techniques would be an elegant solution for the recognition of path resemblances, which we will deal with in subsequent research. Without the use of these techniques, we want to content ourselves for the moment with an abstraction of the path concept used in our technique. Under abstraction of paths, one generally insists that in the description of paths, not exactly the instructions executed during the execution are given, but paths are described by more abstract properties such as the number of executed instructions and their instruction types, or structural properties such as the number of successor nodes of a branch node, etc.

Abstraction is not a novel principle for describing instructions or related program parts and is used, for example, in [27] to abstract basic blocks contained in control flow graphs or in [18] to describe subtrees in abstract syntax trees. In this paper, the principle of abstraction is used to describe paths in the dominator tree. With the abstraction, a path is then

no longer seen as a sequence of node labels representing the instructions executed by the nodes, but simply described as a vector, in which the number of times, different types of instructions contained in the path are processed, is recorded.

Figure 3 describes which types of instructions are considered by our abstraction of paths and contains a concrete example of path abstraction. In the vector descriptions, the term $\cdots$ denotes a sequence of components with value 0.

In the abstraction, paths are described by k-elementary vectors, so that the set of paths in a given dominator tree can be considered as a subset of the k-elementary natural vector space $\mathbb{N}^k, k \in \mathbb{N}$. By specifying an abstraction function $\alpha : Paths \rightarrow \mathbb{N}^k$, where $Paths$ is the set of all paths, the sets D^G_{set} and D^G_{split}, as previously used above to find similar functions or subblocks contained in a function, can be transformed into a form based on the choosen abstraction level and, by forming a corresponding comparison function σ_{abstr}, can also be used to check similar functions or subblocks contained in a function on a corresponding abstraction level.

IV. Implementation and Preliminary Results

The method has been implemented as an additional component in the *Soot framework* [31]. Soot is a system for analyzing and optimizing Java bytecode. To generate the dominator trees, the Soot system loads and processes a single Java class file at a time. The component we implemented in the Soot system takes the dominator tree created for a function, derives the various path sets, and stores them in files. To describe the individual paths for the non-abstract versions, they are mapped to a sequence of natural numbers using a simple coding. If abstraction is used for the description of path sets, paths are described by the vector representation as explained in Sect. III.

In our first experiments, we were particularly interested in identifying code clones for a real-world benchmark and to check whether they can be recognized with our method. As a benchmark, the open-source Java framework *libGDX*[2] and code clones therein, as identified by the state-of-the-art clone detector NiCad [5], [10], were chosen. In the experiments, NiCad's configuration for Type 2 clones was used. Type 2 clones are very similar code clones, such that the clones are assumed real code clones and therefore build a baseline for evaluating our approach. Since manual reviews were performed in some cases, only functions with a manageable length between 8 and 35 lines were compared. Our procedures ran with a threshold Euclidean distance value of 0.1.

Similar to NiCad, identified clones were merged into clone classes. Figure 4 contains a comparison of the clone classes found by our method with those generated by NiCad. At this point, it should be mentioned that, in the experiments, NiCad found 9 classes which contained functions which were much changed when mapping them to Java bytecode and therefore could not be recognized by our method as textually identical. For reasons of fairness, these classes were removed from the comparison. From the remaining benchmark, NiCad generated a total of 236 clone classes. The same clone classes, but also additional classes, were identified with our different methods.

[2](https://github.com/libgdx/libgdx)

	Classes compared to NiCad			
		Same classes		
Method	Total number	Exactly the same	Expanded by functions	Additional classes
NiCad	236			
D_{set}	236	203	33 (∅ plus 3)	90
D_{split}	236	201	34 (∅ plus 3)	99
$D_{set-abstr}$	236	192	45 (∅ plus 4)	126
$D_{split-abstr}$	236	187	49 (∅ plus 4)	133

Fig. 4. Comparison of the found clone classes.

For the found clone classes, when compared with NiCad, it applied that these were either exactly the same or extended by additional functions. For example, for the experiments with the non-abstract version of D_{set}, in which the path sets are given by complete paths from the start node to the leaf nodes, 203 classes were exactly the same and 33 classes were extensions of the classes derived by NiCad, in which on average 3 functions were added. In addition, 90 more clone classes were found by our procedure compared to NiCad.

Since the methods presented here do not work on Java source code like NiCad, variables and constants are recognized, for example, and can be excluded from the comparison. In addition, our methods benefit from the chosen program representation of Java bytecode and the dominator trees, which, through further abstraction, allowed for recognizing semantically similar code despite syntactic differences.

At a first glance, it appears that the experiments for the abstract versions of our method show no improvement, as the number of exactly the same clone classes decreases. On the contrary, more clones could though be found with these versions, so more clone classes also found by NiCad could be extended. This improvement is mainly due to the fact that these versions are completely abstracting from the underlying source code syntax and that code clones could also be detected, which result from the swapping of instructions.

In further experiments, we performed measurements with the abstract methods to determine with what inaccuracy NiCad must be executed to find all the code clones we found in the clone classes. The results obtained for the individual classes were in between 8% and 44% for the textual inaccuracy to be then used. NiCad in this case added between none and 23 additional functions - up to 508 functions for all classes - for the measurements in the individual clone classes. In fact, a direct comparison with NiCad would not be fair, but the measurements at least show the interesting fact that our method is performed at peak values with 44% textual independence.

V. Conclusion

We have presented a method for detecting code clones, which can be used to identify reused or cloned malicious functions and malware, as well as subfunctions or blocks thereof. The method uses the information of dominator trees to locate suspicious program parts, which distinguishes the method from other known techniques for clone code detection. In the preliminary evaluation, the state-of-the-art clone detector NiCad was used as baseline for detecting code clones in the libGDX Java framework to evaluate the basic functionality of our new method. As the experiments show, when considering code clones on a rather textual level, the proposed method does not only achieve comparable results, but also find more clones compared to NiCad. Additional measurements for a case study also show that the method can accurately identify semantically equivalent subblocks where instructions have been swapped. Further experiments show that the method can perform at peak values with 44% textual variance.

In fact, the method presented in this paper is still in its initial state, but the work done so far introduces the essential approach which is to be extended in future development. As a starting point, an additional abstraction and encoding of instructions used in the paths of the dominator tree should be defined, with which instructions similar to each other can be mapped to the same encoding. This would result in the possibility of abstracting from the path term already in the pure path-oriented comparison. As an alternative, machine learning could be used to train a neural network to recognize partial paths or semantically similar paths and to encode them in the same way, respectively. Above all, however, work needs to be done for comparing our method to syntactical and semantic techniques for code clone detection, e.g., based on the *BigCloneBench* benchmark [6].

References

[1] A. Rahimian, R. Ziarati, S. Preda, and M. Debbabi, "On the reverse engineering of the citadel botnet," in *FPS*, vol. 8352 of *Lecture Notes in Computer Science*, pp. 408–425, Springer, 2013.

[2] A. Calleja, J. Tapiador, and J. Caballero, "The malsource dataset: Quantifying complexity and code reuse in malware development," *IEEE Trans. Information Forensics and Security*, vol. 14, no. 12, pp. 3175–3190, 2019.

[3] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.

[4] J. Chen, M. H. Alalfi, T. R. Dean, and Y. Zou, "Detecting android malware using clone detection," *J. Comput. Sci. Technol.*, vol. 30, no. 5, pp. 942–956, 2015.

[5] C. K. Roy and J. R. Cordy, "NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *ICPC*, pp. 172–181, IEEE Computer Society, 2008.

[6] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *ICSME*, pp. 131–140, IEEE Computer Society, 2015.

[7] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *CASCON*, pp. 171–183, IBM, 1993.

[8] J. H. Johnson, "Visualizing textual redundancy in legacy source," in *CASCON*, p. 32, IBM, 1994.

[9] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *IEEE Symposium on Security and Privacy*, pp. 595–614, IEEE Computer Society, 2017.

[10] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *ICPC*, pp. 219–220, IEEE Computer Society, 2011.

[11] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[12] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 176–192, 2006.

[13] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *ICSE*, pp. 1157–1168, ACM, 2016.

[14] M. A. Nishi and K. Damevski, "Scalable code clone detection and search based on adaptive prefix filtering," *J. Syst. Softw.*, vol. 137, pp. 130–142, 2018.

[15] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: a token based large-gap clone detector," in *ICSE*, pp. 1066–1077, ACM, 2018.

[16] T. Ishio, N. Maeda, K. Shibuya, and K. Inoue, "Cloned buggy code detection in practice using normalized compression distance," in *ICSME*, pp. 591–594, IEEE Computer Society, 2018.

[17] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSM*, pp. 368–377, IEEE Computer Society, 1998.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *ICSE*, pp. 96–105, IEEE Computer Society, 2007.

[19] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: detection of clones in the twilight zone," in *ESEC/SIGSOFT FSE*, pp. 354–365, ACM, 2018.

[20] L. Li, H. Feng, W. Zhuang, N. Meng, and B. G. Ryder, "Cclearner: A deep learning-based clone detection approach," in *ICSME*, pp. 249–260, IEEE Computer Society, 2017.

[21] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *ICPC*, pp. 70–80, IEEE / ACM, 2019.

[22] L. Büch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in *SANER*, pp. 95–104, IEEE, 2019.

[23] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, 2019.

[24] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *IEEE Symposium on Security and Privacy*, pp. 709–724, IEEE Computer Society, 2015.

[25] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *ICPC*, pp. 88–98, IEEE Computer Society, 2017.

[26] F. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. E. Kaiser, and T. Jebara, "Code relatives: detecting similarly behaving software," in *SIGSOFT FSE*, pp. 702–714, ACM, 2016.

[27] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *ACM Conference on Computer and Communications Security*, pp. 363–376, ACM, 2017.

[28] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *ESEC/SIGSOFT FSE*, pp. 141–151, ACM, 2018.

[29] A. Frank, "On Kuhn's Hungarian method — A tribute from Hungary," *Naval Research Logistics*, vol. 52, pp. 2–5, Feb. 2005.

[30] W. Reisig, *Elements of distributed algorithms: modeling and analysis with Petri nets*. Springer, 1998.

[31] P. Lam, E. Bodden, and L. Hendren, "The soot framework for java program analysis: A retrospective," July 15 2013.