On the Robustness of Clone Detection to Code Obfuscation

Sandro Schulze
TU Braunschweig
Braunschweig, Germany
sandro.schulze@tu-braunschweig.de

Daniel Meyer
University of Magdeburg
Magdeburg, Germany
Daniel3.Meyer@st.ovgu.de

***Abstract*—Code clones are a common reuse mechanism in software development. While there is an ongoing discussion about harmfulness and advantages of code cloning, this discussion is mainly centered around aspects of software quality. However, recent research has shown, that code cloning may have legal implications as well such as license violations. From this point of view, a developer may favor to hide his cloning activities. To this end, he could obfuscate the cloned code to deceive clone detectors. However, it is unknown how robust certain clone detection techniques are against code obfuscations. In this paper, we present a framework for semi-automated code obfuscations. Additionally, we present a case study to evaluate the robustness of selected clone detectors against such obfuscations.**

I. Introduction

Reusing code by copy&paste (and modification) is a common practice in software development. Numerous studies report about code clones in open source as well as industrial systems, ranging from 7 % up to 50 % [1]–[3]. To detect such replicated code, several approaches have been proposed in the field of *clone detection*, mainly differing in their underlying technique. Furthermore, there is an ongoing, and partly heated, discussion on the harmfulness of clones. On the one hand, code clones are criticized to cause increased maintenance effort or introduce and propagate bugs [4], [5]. On the other hand, certain patterns of cloning and the reliability of cloned code is mentioned as advantage [6]. However, this discussion is mainly about the aspect of software quality.

Recently, the aspect of legal implications of code cloning gained attention. For example, people may copy code from open source systems or libraries, but violate the respective license when reusing it in the new context [7]. Another example are internal restrictions that may prohibit the replication of code across subsystems. In such a case, replication of code may have severe consequences for the developer. Hence, a developer may aim at concealing his cloning activities for not being prosecuted. A possibility to do this is to obfuscate the cloned code using program transformations. However, while large studies exist that compare cloned detection techniques tools (e.g., [8]), nothing is known how these techniques deal with obfuscated clones.

In this paper, we address this issue to find out how robust certain clone detection techniques are with respect to code (clone) obfuscation. To this end, we present ARTIFICE, a framework that provides different program transformations to (semi-automatically) obfuscate the code. By means of this framework and three clone detection tools, we analyze how certain transformations affect the clone detection result. We show, that even more sophisticated clone detection techniques (e.g., AST-based or PDG-based) can be deceived by obfuscating cloned code. Overall, we make the following contributions:

- We provide an extensible framework for semi-automated code obfuscations using program transformations.
- We present a cases study to analyze the effect of code obfuscations on the clone detection results. Particularly, we want to know whether and to what extent obfuscations decrease the amount of detected clones.
- We provide an extensive discussion on our results and figure out why certain detection techniques are affected by code obfuscations. With this, we want to initiate a more general discussion on how code obfuscation may hamper clone detection and how to address this issue.

II. Background

In this section, we provide basic information on clone detection techniques and code obfuscation, which is necessary for the remainder of the paper.

A. Clone Detection Techniques

To detect code clones, different techniques exist. The main difference between these techniques is the underlying source code representation and their respective characteristics. In the following, we explain the four most common techniques.

Text-based: This technique compares the textual source code line-by-line with only minor normalization such as removing whitespace, line breaks, or comments [9]. As a result, it is a flexible and language-independent approach to detect clones. However, it is limited to detect only type-I clones and partly type-II clones (in case of only minor changes).

Token-Based: With this technique, the source code is transformed into a *token stream* by performing a lexical analysis. During this transformation, different normalization can be applied to the token stream such as identifiers or literals [10]. Consequently, this technique can detect all kinds of type-II clones. To detect clones, the token stream is searched for similar token sequences of maximum length.

AST-Based: This technique uses an *abstract syntax tree (AST)* as source code representation to detect clones. Based on this

978-1-4673-6445-4/13 © 2013 IEEE

IWSC 2013, San Francisco, CA, USA

tree, different algorithms can be used to find similar subtrees and thus to detect clones [11], [12]. Because an AST provides a more abstract view on the source code, more code clones (e.g., type-III) can be detected than with the aforementioned techniques.

PDG-Based: This technique goes even beyond the AST-based by taking information about control and program flow into account. As a result, this technique may even neglect structural differences of code fragments. Hence, with this technique, we could even detect type-IV clones.

B. Code Obfuscation

Generally, code obfuscation describes the process of changing the source code of an application while keeping its functionality. Hence, it is quite similar to refactoring, which aims at improving the structure of code by preserving its behavior. Collberg et al. define code obfuscation in a more formal way as follows [13]:

Definition 1: Let $P \xrightarrow{\tau} P'$ be a transformation of a source program P into a target program P'. $P \xrightarrow{\tau} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behavior*.

Consequently, the following conditions must be fulfilled for an obfuscating transformation:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

A variety of transformations are available that can be applied for obfuscating source code. Collberg et al. categorize these transformations into four groups. First, *layout obfuscations* aim at rendering the source code unreadable. Examples are removing of comments, scrambling of identifiers, and changes in formatting the code. Second, *data obfuscations* mainly affect data and data structures of the program. For instance, a developer may split or upgrade variables (e.g., `int` to `Integer`), or restructure arrays. Third, *control obfuscations* affect the control flow of a program. For instance, reordering of statements, algorithmic changes, or inserting dead code are examples for this category. Finally, the goal of *preventive obfuscation* is to impede the reverse engineering of a compiled program by so called deobfuscators.

The aforementioned obfuscations can be applied for different purposes. For instance, one main goal of code obfuscation is to protect intellectual property and knowledge contained in (commercial) software systems. Another use case for code obfuscation is introducing malware. For instance, authors of malware may introduce additional assembler instructions to obfuscate those instructions containing the malformed code [14]. Finally, code obfuscation may be applied to the plain source code level to conceal plagiarism or license violations, which is of special interest in this paper.

III. A Framework for Code Obfuscation

For analyzing the effect of obfuscated source code on clone detection, we need different versions of a program, each changed by certain obfuscations. Even for small programs of some KLOC, this task is time-consuming and error-prone if done manually. To make this step more reliable and less tedious, we developed ARTIFICE, a simple code obfuscator for Java as an Eclipse plug-in. At the bottom line, ARTIFICE enables a developer to apply semi-automated code transformations to obfuscate the source code. For our case study, we implemented the following transformations on top of the Eclipse refactoring engine.

- **Renaming:** Sequential renaming of variables, fields and methods
- **Expansion:** Elimination of assignment and in-/decrement operators.
- **Contraction:** Insertion of assignment and in-/decrement operators, if possible.
- **Loop Transformation:** Transformation from `for` to `while` loops and vice versa.
- **Conditional Transformation:** Transformation of if/else clauses to the equivalent short form and vice versa.

In Figure 1, we show an example for some of these transformations. First, we apply renaming to different fields of the original code fragment (e.g., the field `i` is renamed into `m`). Furthermore, we apply loop transformation to transform a `while` loop into a `for` loop. As a result, the incrementor variable in Line 9 of Figure 1 a is moved to the loop initialization of the `for` loop (Line 3 in Figure 1 b). Finally, we applied a conditional transformation to the if-statement in Line 6 of Figure 1 a.

To execute one or more of the aforementioned obfuscations, the developer has to choose a java project within Eclipse. At this point, our plugin already checks internally the whole project for renaming opportunities such as methods and variables. Furthermore, a *logging unit*, which we use for tracking all obfuscations, is initialized. After this initialization, a dialog appears where the developer can select the obfuscations he wants to apply. Furthermore, he can specify new names for each renaming obfuscation. Because this can be a tedious task in the presence of hundreds of methods and variables, we provide an alternative: The developer can trigger the plugin to generate artificial names for all methods and/or variables. Finally, the selected obfuscations are applied to the source code. To this end, we make use of the *abstract syntax tree (AST)* that is provided by Eclipse. For a more detailed overview of the obfuscation process we refer to [15]. The plugin is available at https://www.tu-braunschweig.de/isf/research/artifice.

IV. Case Study

In this section, we describe the general setup of our case study and the methodology of the obfuscation and clone detection process.

A. Objectives

Our main goal is to investigate *how* robust specific clone detection techniques (or corresponding tools, respectively) are with respect to code obfuscation. To narrow down this rather

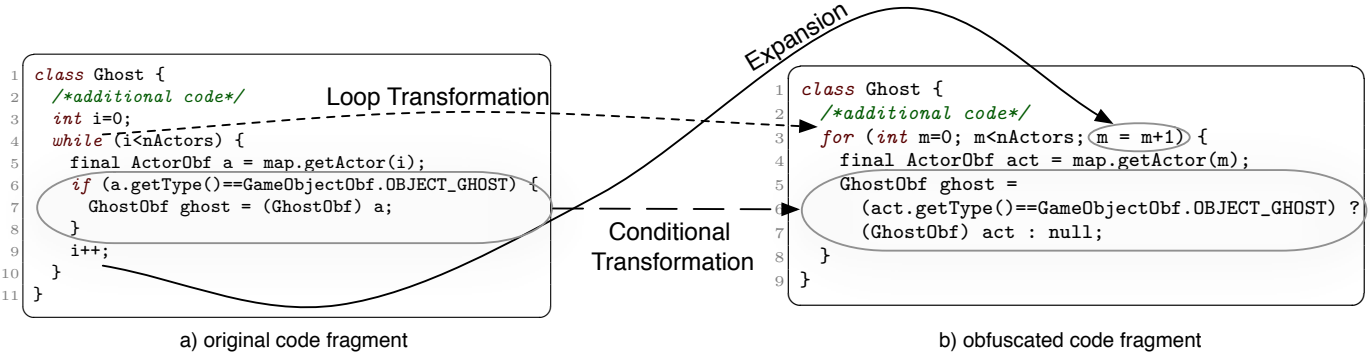

Fig. 1. Examples for expansion, loop transformation, and conditional transformation.

general target, we pose two research questions, we want to answer:

RQ1 – To what extent does code obfuscation effect the clone detection?

With this questions, we want to investigate two separate aspects of code obfuscation. First, we want to analyze, whether code obfuscation has an effect on the clone detection result, that is, the detected clones after code obfuscation. For instance, if less code clones are detected due to code obfuscation, this is annoying, because it implies that we miss copied code during its detection. Especially in the context of license violation and plagiarism, which we consider as use cases, a low recall value may have consequences such has impeding prosecution of the aforementioned violations. A second aspect we focus on is how the effect of code obfuscation differs with respect to the different clone detection techniques. For instance, we would expect that simple, text-based clone detection is more prone to such obfuscations than token- or tree-based techniques.

RQ2 – Are there differences between the particular obfuscations regarding clone detection? For our analysis, we implemented different program transformations to obfuscate the source code. These transformations mainly differ in the way *how* they change the structure of the underlying source code. Consequently, there may be also differences regarding the effect of certain obfuscations to the clone detection result. We are interested in two issues, based on this questions: First, whether certain obfuscations are more likely to affect the clone detection result. And second, whether obfuscations exist that have a relatively high effect on certain clone detection techniques (and how this differ between the considered techniques).

B. Subject System

For our case study, we selected a java implementation of the well-known game *Pacman* as our subject system. It consists of 21 files and about 2400 lines of code. Although this is rather a small (”toy”) system, it has some advantages. First, we could check our obfuscations by inspecting (a subset of) the transformed code fragments. Similar, we could randomly select a subset of detected code clones to check whether they are clones or not. Hence, this manual inspection increases the reliability of our case study, which is an important aspect. The source code of the system is available at https://code.google.com/p/pacman-rkant/.

C. Clone Detection

For evaluating the different clone detection techniques, we need at least one tool that implements the corresponding technique, respectively. In the past, a variety of clone detection tools has been proposed for each technique. Even if we omit those tools that turned out to be useless or not available any more, we end up with a considerable selection of tools. However, since our goals is not to be complete with respect to the used tools, we finally selected three clone detection tools that turned out to be useful and applicable for our purposes. We list the tools with some information about the clone detection parameters in Table I.

TABLE I
CLONE DETECTION TOOLS USED WITHIN THE CASE STUDY TOGETHER WITH INFORMATION ON CORRESPONDING DETECTION TECHNIQUE AND IMPORTANT PARAMETERS.

Technique	CD tool	Parameters
Text-based	JPLAG (v.2.2.1)	Min clone length: 5 string token
Token-based	JPLAG (v.2.2.1)	Min clone length: 9 token
AST-based	CloneDigger (revision 211)	Min clone length: 10 tree nodes, distance: 200
PDG-based	Scorpio (access: 03.03.2011)	Min clone length: 5 PDG nodes

For the text-based as well as the token-based, we used *JPLAG*[1] as a clone detector. JPLAG has been designed for plagiarism detection and thus is designed to detect illegitimately copied code. Beyond this, JPLAG has been developed for more than 10 years and has also been used by numerous people since then. Hence, it is in a mature state and provides reliable results. For the text-based clone detection, we selected a minimum length of five string token, which actually means five words. We tested the text-based detection mechanisms with different lengths, but five turned out to achieve the best results. Next, we determined the minimum length for the token-based detection to be nine tokens. While this appears to be quite low compared to previous work with other token-based tools (e.g., CCFinder recommends 30 token as minimum

[1]Website: https://www.ipd.kit.edu/jplag/

length [10]), this value is mainly caused by the way JPLAG performs tokenization. In Figure 2, we show examples for a `for` and `while` loop after tokenizations, respectively. In particular, JPLAG seems to summarize certain elements to rather coarse-grained tokens. For instance, the token `VARDEF` encompasses the whole initialization of the counting variable. Hence, a token in JPLAG is somewhat identical to a part (or possibly the entire) statement. We tried other values for the clone length as well, but nine provided the best result.

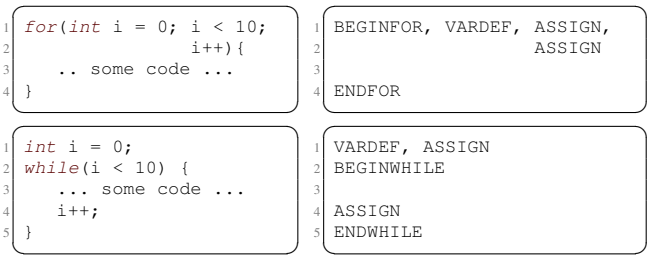

Fig. 2. Token-based representation of `for` and `while` loop in JPLAG.

For the AST-based clone detection, we use the tool *CloneDigger*[2]. To detect code clones, based on an syntax tree representation, CloneDigger uses anti-unification [16]. For our purpose, we specified 10 AST nodes as the minimum clone length. Furthermore, we set the distance to a value of 200. In the context of CloneDigger, the distance provides the possibility to specify the degree of *dissimilarity* that is allowed between subtrees. For instance, if a subtree has a similar structure but differs in terms of variable names or operators, those differences are neglected if the distance value is relatively high. Finally, we use *Scorpio*[3] for the PDG-based detection. Scorpio uses a modified program dependency graph with an additional execution edge between two nodes to overcome some limitations of PDG-based clone detection [17].

D. Methodology

For the analysis of the robustness, we apply our code obfuscations to the original source code and perform clone detection by using the introduced tools. In the following, we provide details about both steps.

Code obfuscation: This is the initial step of our analysis process, where we apply the code obfuscations, introduced in Section III. To this end, we use our framework to find appropriate code fragments and apply the respective code obfuscations. It is worth to note, that we apply the obfuscations on a copy of the original system, because we need the original version for the clone detection step. Beside the particular obfuscations, we perform each possible combination of obfuscations, e.g., renaming and extraction. As a result, we can even analyze whether certain combinations have a considerable effect on the clone detection result. After each kind of obfuscation, we performed an acceptance test to ensure that the observable behavior of the game did not change. Finally, we log all of our obfuscations together with the file name and position (i.e., which lines of code have been affected) using the logging mechanism of our framework.

Clone Detection: In this step, we aim at detecting clones between the original source code (i.e., the subject system without obfuscations) and all obfuscated versions of system. We start by comparing the original system with itself and take the resulting clone ratio as a *reference value* for each of the other comparisons. Next, we perform clone detection on the original system with one of the obfuscated systems. Since we are interested in *cross-system clones* (i.e., clones that occur *between* both investigated systems), we filter out all other clones that occur in the result set. Afterwards, we determine the overall similarity between both, the original and the obfuscated system. In this context, similarity is defined as $\frac{cr(org,obfuscate)}{cr(org,org)}$, where $cr(org, obfuscate)$ is the clone ratio between the original system and the respective obfuscated version of the system and $cr(org, org)$ represents the reference value. We do this as a kind of normalization step and thus to achieve comparability between the particular clone detection tools. Otherwise, the absolute values are only hardly comparable, because each tool uses another source code representation such as plain text and nodes of an AST. Nevertheless, we also store the amount and length of clones for each of the comparisons. We repeat the aforementioned clone detection process for each obfuscated systems (always compared too the original one) and for each clone detection tool.

Overall, we could apply all obfuscations to our subject systems, except for the conditional transformation. In particular, we applied 616 renaming obfuscations (120 methods, 155 fields, 341 variables), 54 expansions, 8 contractions, 34 loop transformations. These four obfuscations and all of its combinations result into 16 different obfuscated versions of the original system that had to be considered during clone detection. In the next section, we present and discuss the results of our case study.

V. Results

In the following, we present the results of our case study. Based on these results, we discuss the robustness of each detection tool in particular. Furthermore, we put the result in the context of the underlying detection technique and discuss its influence on the detection results. Finally, we discuss the threats to validity of our case study. In Figure 3, we show an overview of the clone detection results for each detection tool and each (combination of) obfuscation. All similarity values are shown relatively to the reference value (i.e., 100%)

A. Robustness of Text-Based Clone Detection

Text-based clone detectors rely solely on the textual representation of source code. Although they provide minor normalization facilities such as removing whitespaces or comments,

[2]Website: http://clonedigger.sourceforge.net/index.html

[3]Website: http://sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio-e, we use version v201103030634

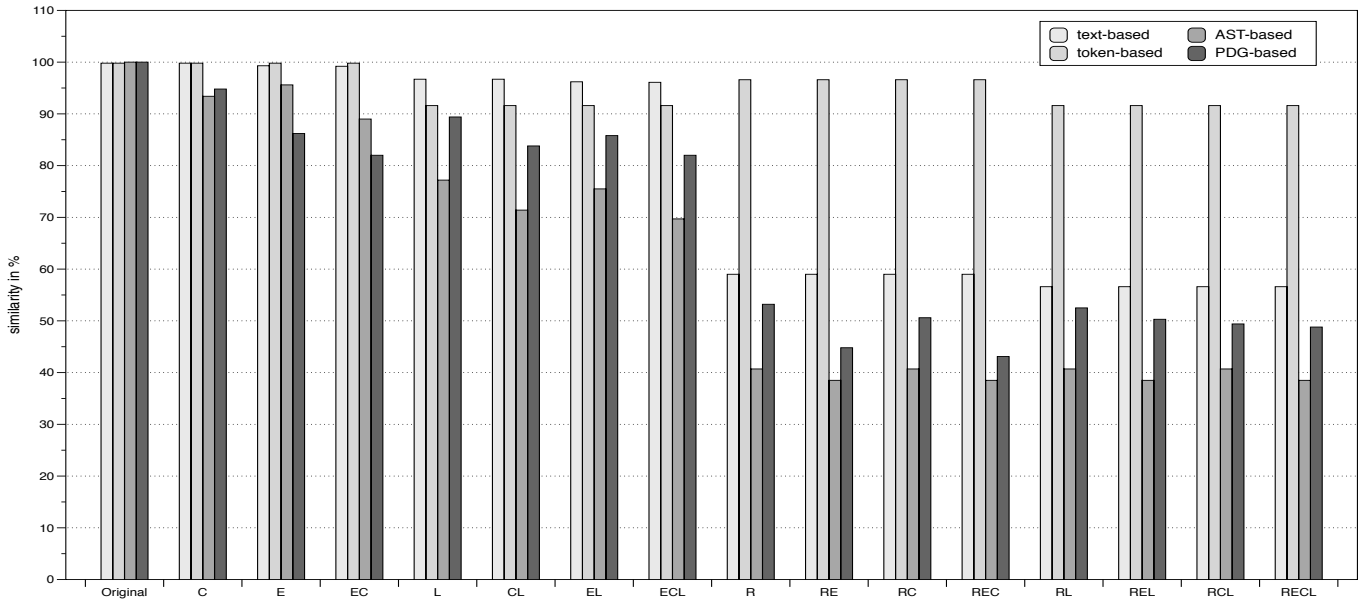

Fig. 3. Similarity of original program compared to obfuscated programs for each clone detection technique. The obfuscations are abbreviated as follows: C – contraction, E – expansion, L – loop transformation, R – renaming,

we generally expect that nearly each obfuscation has an effect on the clone detection result.

In contrast to our assumption, our data reveal that only a few obfuscations have a considerable effect on the result of the clone detection. Most notably, *renaming* reduces the number of detected clones. This is indicated by a decrease of 40%, regarding the similarity measure in Figure 3. From our point of view, this is caused by two reasons. First, we applied renaming more frequently than all other obfuscations and thus this obfuscation causes more structural changes. Second, we expect that renaming occurs in consecutive lines of code and thus result into large gaps between the detected clones. This is also reflected by the increase in the number of detected clones (608 compared to 21 in the reference clone detection) as well as their length (13.9 compared to 725 on average).

Regarding the *loop transformation*, our data reveal that is causes an observable decrease (-4%), if we take into account that we applied this obfuscation only rarely. This is also supported by the number of clones (96 compared to 21) and the clone length on average (144 compared to 725).

All other obfuscations (and their combinations) have only a minor effect on the detection result. Even though we applied these obfuscations less frequently, this result is rather unexpected. One reason could be how these obfuscation are distributed across the source code. If they occur across the whole code base, they possibly cause only minor gaps and thus large portions of copied code remain unchanged.

Generally, we argue that text-based clone detection is not very robust against code obfuscation, which coincides with our assumption. Nevertheless, the minor effect of more sophisticated obfuscations is not entirely clear and worth to investigate further.

B. Robustness of Token-Based Clone Detection

A token-based detection tool transforms the source code into a token stream and provides more profound normalizations such as parameterizing identifiers. Hence, such a detection tool even detects clones that differ to some extent (mostly type-I and type-II). In particular, we assume that renaming should have no effect on the detection rate.

Our data reveal that loop transformation has the biggest impact on the detection result, indicated in a decrease of 10% regarding the similarity measure. The reason is how the detection tool (JPLAG) transforms the source code into token. Considering our example in Figure 2, we can see that the token sequence of a `for` and a `while` loop exhibit considerable differences. Consequently, they are not detected as clones, even if they realize the same functionality (which can be considered as type-IV clone).

Beside the loop transformation, only the renaming obfuscation decreases the detection result. This, however, is an observation that we did not expect. When investigating the respective code fragments, we recognized that this is caused by a peculiarity of our renaming obfuscation. In case that it renames a variable initialization (e.g., `int i=1`), the obfuscation splits declaration and initialization of the respective variable (e.g., `int m; m=1;` with line break). Even in case of a `for` loop, our obfuscation separates declaration and initialization of the counting variable (by placing the declaration directly before the loop initialization). Consequently, the respective statements differ even in their token representation and thus are not entirely detected as clones.

Overall, we argue that the token-based clone detection exhibits a relatively high robustness against code obfuscation. Even though two obfuscations have an effect on the detection result, the overall similarity is still about 90%. However, our

results indicate that the concrete realization of the token-based techniques may be an important aspect for the degree of robustness. Especially, the way how JPLAG transform a loop into tokens is related to the robustness against our loop transformation.

C. Robustness of Tree-Based Clone Detection

Both, AST- and PDG-based clone detection, use an abstract syntax tree as underlying structure and thus can even detect type-III clones. Additionally, the PDG-based techniques makes use of control and data flow and thus may even detect type-IV clones. Consequently, we assume that both approaches are robust against our code obfuscations.

AST-based detection: Our data reveal that, again, the *renaming* obfuscation has a considerable effect not he detection result. Particularly, it decreases the overall similarity by 60%, which is even worse than the result of the text-based detection tool. Due to this unexpected result, we had a closer look on the detected clones. We observed that the separation of declaration and initialization of variables during the renaming obfuscation is the main reason for this result. This separation introduces a new node in the underlying AST, which changes its structure. Furthermore, if such separation occurs repeatedly within few lines, this may break one large clone into a set of small clones, which not exceed the required threshold of nine AST nodes. Consequently, these clones are not detected.

Next, the *loop transformation* leads to a considerable lower similarity (-23%). As with the renaming obfuscation, the introduction of new AST nodes is the main reason. Particularly, by transforming a `for` into a `while` loop, we move the counting variable outside the loop. This seems to affect the structure of the AST in such way, that the affected code fragments are not detected as clones after the obfuscation.

Similarly, the *contraction* obfuscation is worth to mention, because of its influence on the detection result. Again, this obfuscation affects the structure of the AST by removing the AST node that corresponds to the original assigning statement.

Finally, some combinations of obfuscation have an effect on the detection result. However, this mostly results from summarizing the effects of the particular obfuscations.

In summary, the AST-based technique is robust against standard renaming (as known from refactorings), and expansion. In that way, it coincides with our assumption. However, the contraction obfuscation and loop obfuscation had a considerably effect on the detection result. Furthermore, the effect of renaming with separation indicates, that this technique may not be robust against more sophisticated obfuscations such as reordering or adding statements.

PDG-based detection: Similar to the AST-based technique, the *renaming* obfuscation has the highest impact by reducing the amount of clones significantly (-47% regarding similarity). When investigating the obfuscated code, we discovered that a high density of this obfuscation (i.e., several renamings in only few lines of code) is the main reason for the effect of this obfuscation. As a result, the affected PDG nodes have not been considered to be part of a clone. Hence, if such nodes occur consecutive, a larger code fragment is not detected as clone.

Moreover, nearly each other obfuscation has an effect on the detection result as well. The *contraction* obfuscation, for example, reduces the number of PDG nodes (similar to the AST-based technique) and thus the minimum clone length is not reached. We observed a similar behavior for the *expansion* obfuscation and the *loop transformation*. For the latter, this is the case if we transform a `for` into a `while` loop, which implies that the "updater" variable (e.g., `i++`) is moved to the end of the `while` loop.

We conclude that, in contrast to our assumption, every obfuscation influenced the detection results negatively. The authors of the *Scorpio* tool state that PDG-based clone detection is not very suitable for detection contiguous clone detection. This could be an explanation of the results. Nevertheless, we argue that, based on our results, the performance of the PDG-based technique is poor for *each* obfuscation, which can't be entirely explained by that limitation.

D. Threats to Validity

Although we conducted our case study with care, it may exhibit certain threats to validity, especially regarding the generalizability. We discuss these threats in the following.

Clone Detection: First, we used specific implementations of the detection techniques by means of specific clone detection tools. Hence, our results are mainly valid for these tools. However, other tools such as NiCAD or ConQAT [2], [18] may produce different results. Consequently, our results are only hardly generalizable. But even between specific tools commonalities exist regarding the underlying data structure. Consequently, we argue that our results are not totally biased to the selected tools and provide valuable insights on the relation between code obfuscation and clone detection technique.

Subject System: For our case study, we analyzed one small java system. Hence, our results are in first place valid for this systems. Other systems that differ in size and programming language may lead to totally different results. Particularly, the frequency of the applied obfuscations may differ between different systems. Nevertheless, we argue that the effect of theses obfuscations is very similar, independent of the underlying system.

VI. Conclusion

In this paper, we analyzed the robustness of clone detection techniques against code obfuscation. We presented a framework that supports the user by applying code obfuscations semi-automatically to source code. Furthermore, we conducted a case study to measure the influence of these code obfuscations on the clone detection result. In particular, we formulated two research questions that we want to pick up in the following for our conclusion.

To what extent does code obfuscation effect the clone detection?

Generally, code obfuscation has an effect on each of the considered detection techniques (or tools). Nevertheless, we

observed differences regarding the robustness for the particular techniques. While the text- and token-based techniques show no peculiarities, we investigated somewhat unexpected or even surprising results for the tree-based techniques. Particularly, we observed that code obfuscation has a considerable effect on these techniques. The reason is that changing the number or order of statements directly effects the underlying tree structure, representing the source code. Hence, we conclude that even more sophisticated detection techniques, although they operate on an abstract source code representation, are not robust against code obfuscations.

Are there differences between the particular obfuscations regarding clone detection? This question, we can definitively answer wit "yes". Most notably, the renaming obfuscation had the highest impact on detected clones for *all* detection techniques. But also the other obfuscations had an impact on single techniques, respectively. In contrast, we did not find any evidence, that the combination of obfuscations reinforces the effect of the single obfuscations.

Overall, we conclude that code obfuscation is a severe problem, because it has a considerable effect on the clone detection result. Moreover, this effect is already observable for rather simple obfuscations such as renaming and across all clone detection techniques. Hence, we argue that a detailed discussion on that topic is worth to consider. As a possible result, countermeasures to improve clone detection in the presence of obfuscations could be a valuable outcome.

VII. Related Work

Different studies exist that aim at comparing clone detection techniques. Initially, Bellon et al. conducted a case study on eight systems (four java, four C++) to evaluate six clone detectors [19]. As reference value, a *human oracle* manually examined the subject systems to evaluate clone candidates. Based on this reference code clones, the tools are evaluated regarding precision and recall of their clone detection result. Furthermore, Roy et al. presented a comprehensive case study on clone detection techniques [8]. Within their study they propose a scenario-based taxonomy of clone detection techniques for different types of clones. Based on this taxonomy, they provide a qualitative evaluation of clone detectors.

However, both case studies compare detection tools and techniques with each other regarding precision and recall. In contrast, we compare each technique with itself and how they perform in the presence of code obfuscations.

VIII. Future Work

In future work, we want to extend the case study of this paper to address possible shortcomings. In particular, we want to evaluate other clone detection tools to obtain a more comprehensive overview of how certain techniques perform in the presence of code obfuscations. Additionally, we intend to analyze more software systems of different size and domain for investigating the influence of the subject system (and its structure) on our analysis.

Furthermore, we plan to extend our framework. Particularly, we focus on more complex obfuscations such as statement reordering or injecting related statements that do not change the observable behavior. With both, extension of our case study as well as extending our framework, we expect to provide more comprehensive insights and possible countermeasures that may help to making clone detection more robust against code obfuscation.

References

[1] B. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," in *Proc. Working Conference on Reverse Engineering*. IEEE, 1995, pp. 86–95.

[2] C. Roy and J. Cordy, "An Empirical Study of Function Clones in Open Source Software Systems," in *Proc. Working Conference on Reverse Engineering*. IEEE, 2008, pp. 81–90.

[3] J. Cordy, "Comprehending reality-practical barriers to industrial adoption of software maintenance automation," in *Proc. Int'l Workshop on Program Comprehension*. IEEE, 2003, pp. 196–205.

[4] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do Code Clones Matter?" in *Proc. Int'l Conf. Software Engineering*. IEEE, 2009, pp. 485–495.

[5] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone Smells in Software Evolution," in *Proc. Int'l Conf. Software Maintenance*. IEEE, 2007, pp. 24–33.

[6] C. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered Harmful," in *Proc. Working Conference on Reverse Engineering*. IEEE, 2006, pp. 19–28.

[7] A. Monden, S. Okahara, Y. Manabe, and K. Matsumoto, "Guilty or Not Guilty: Using Clone Metrics to Determine Open Source Licensing Violations," *IEEE Software*, vol. 28, no. 2, pp. 42–47, 2011.

[8] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Sci. Comp. Prog.*, vol. 74, no. 7, pp. 470–495, May 2009.

[9] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in *Proc. Int'l Conf. Software Maintenance*. IEEE, 1999, pp. 109–118.

[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code," *Trans. Soft. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[11] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proc. Int'l Conf. Software Maintenance*, nov 1998, pp. 368 –377.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and Accurate Tree-based Detection of Code Clones," in *Proc. Int'l Conf. Software Engineering*. IEEE, 2007, pp. 96–105.

[13] D. L. Christian Collberg, Clark Thomborson, "A Taxonomy of Obfuscating Transformations," Department of Computer Science, Tech. Rep. 148, 1997.

[14] K. Coogan and S. Debray, "Equational Reasoning on x86 Assembly Code," in *Proc. Work. Conf. Source Code Manipulation and Analysis*. IEEE, 2011, pp. 75–84.

[15] D. Meyer, "Analyzing the Robustness of Clone Detection Tools Regarding Code Obfuscation," bachelor thesis, University of Magdeburg, 2012.

[16] J. Reynolds, "Transformational systems and the algebraic structure of atomic formulas," *Machine intelligence*, vol. 5, no. 1, pp. 135–151, 1970.

[17] Y. Higo and S. Kusumoto, "Code clone detection on specialized pdgs with heuristics," in *Proc. Eur. Conf. on Soft. Maint. and Reeng.* IEEE, 2011, pp. 75–84.

[18] E. Juergens, F. Deissenboeck, and B. Hummel, "CloneDetective-A Workbench for Clone Detection Research," in *Proc. Int'l Conf. Software Engineering*. IEEE, 2009, pp. 603–606.

[19] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *Trans. Soft. Eng.*, vol. 33, no. 9, pp. 577–591, 2007.