PRECINCT: An Incremental Algorithm to Prevent Clone Insertion

Mathieu Nayrolles,
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
Montreal, Canada
m_nayrol@ece.concordia.ca

Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
Montreal, Canada
abdelw@ece.concordia.ca

Abstract—**Software clones are considered harmful in software maintenance and evolution. However, after a decade and a half of research, only few approaches have targeted the prevention aspect of clones detection. In this paper, we propose a novel approach named PRECINCT (PREventing Clones INsertion at Commit Time). PRECINCT focuses on near-miss software clones are copied — or reinvented — fragments where minor to extensive modifications have been made and more specifically on detecting at commit time by means of pre-commit hooks. Efficiently detecting near-miss software clones at commit time might call for further refactoring or simply hint developers that they reinvented one piece of code. We apply and validate PRECINCT in terms of precision and recall on seven systems developed independently with a wide range of technologies, size and purposes. The validation demonstrates that our approach detects near-miss software clones before they reach the source version system with a 100% precision and a 93% recall.**

I. Introduction

Code or software clones appear when developers reuse code with little to no modification. Previous research has shown that clones can account for 7% to 50% of a given software [1], [2]. Developers often reuse code and create clones in their software on purpose [3]. Nevertheless, clones are considered a bad practice in software development and therefore, harmful [4]–[6]. The most obvious way a clone can hazardous for the quality of a software system is if a default or bug is discovered in one segment of code that has been copied pasted several times, then the developers would have to remember the places where this segment has been reused in order to fix the default in each of them.

In order to help developers to deal with clones, researchers and practitioners have published hundreds of studies and dozens of tools using different approaches: (1) textual where the source code is considered as text and transformation or normalization is applied to it in order to compare it with order code fragment [7]–[10]. (2) Lexical where the source code is sliced into sequences of tokens as a compiler would [1], [6], [11]–[13]. (3) Syntactic where the source code is converted into trees, more particularly abstract syntax tree (AST) and then, the clone detection is performed using tree matching algorithms [14]–[17].

Henceforth, clones detection can be considered as a mature and still active field of research with two decades of research and hundreds of publications. Consequently, practitioners and engineers know that copy-pasting segment of code can hinder the maintenance and the quality of their systems and that approaches exist to detect them. However, the use of such approaches and tools is not as widespread as one might think. Indeed, as for automatic bug finding tools, the main reasons for this lack of infatuation are that these tools are known to have (1) massive outputs which is (2) hard to understand and contain (3) a high amount of false positives. Moreover, these tools are (4) hard to configure and there is (5) little to no integration of these tools in the day-to-day workflow of a developer [18].

In this paper, we present PRECINCT (PREventing Clones INsertion at Commit Time) that focuses on preventing the insertion of clones at commit time. More specifically, our approach provides a clone detection process that eases the five major reasons that limit the adoption of such tools. To do so, we use pre-commit hooks capabilities of modern source code version control. A pre-commit hook is a process that one can implements to receive the latest modification to the source code done by a given developer just before the code reaches the central repository. Then, analysis can be done using bash programming or calling external programs, allowing the changes to go through or not. PRECINCT is, in fact, a pre-commit hook that detects clones that might have been inserted in the latest changes with regard to the rest of the source code. Consequently, only a fraction of the code is analyzed and PRECINCT reduce the output by 70% (compared to the clone detector PRECINCT is built upon NiCad [9]). Moreover, the detected clones are presented in using a classical diff output that developers are used to. Hence, concerns (1) *massive outputs* and (2) *hard to understand* are less than in other tools. Also, PRECINCTS can be installed and configured using only one command line (*hard to configure*). Finally, our approach leverages the pre-commit hook capabilities of modern source code version control and therefore, integrates itself at the heart of the programmers workflow (*little to no integration of these tools in the day-to-day workflow*).

We assessed the capabilities of PRECINCT in terms of precision and recall on seven systems developed independently with a wide range of technologies and purposes. The validation demonstrates that our approach detects near-miss software clones before they reach the source version system with a

100% precision and a 93% recall while increasing the size of the repository by only 3% to 5%.

The rest of this paper is organized as follows: In Section II we present the works related to PRECINCT and with a particular attention of Nicad. Then, in Section III we present the PRECINCT approach and Section IV shows the experimentations we conduct to assess the capacities of PRECINCT. Finally, we propose some concluding remarks in Section V.

II. Related Works

[19]

III. The PRECINCT Approach

In this section, we present the PRECINCT approach and its steps in details. Our approach is composed of six different steps. The first and the last steps are part of the day-to-day workflow of a programmer using a source version control. Indeed the first step is the commit step where developers send their latest change to the central repository and the last step is the reception of said commit by the central repository. The second step is the pre-commit hook which kicks in as the first operation when one wants to commit. The pre-commit hook have access to the changes in terms of files that has been modified and more specifically, lines that has been modified. The modified lines of the files are sent to TXL [20] for block extraction. Then, the blocks are compared to previously extracted blocks in order to identify clones candidates using NICAD [9]. Finally, the output of Nicad is further refined and presented to the user for a decision round.

In the rest of this section, we present each step in details.

A. Commit

In version control systems, a commit adds the latest changes to the source code to the repository, making these changes part of the head revision of the repository. Unlike commits in data management, commits in version control systems are kept in the repository indefinitely. Thus, when other users do an update or a checkout from the repository, they will receive the latest committed version, unless they specify they wish to retrieve a previous version of the source code in the repository. Version control systems allow rolling back to previous versions easily. In this context, a commit within a version control system is protected as it is easily rolled back, even after the commit has been done.

B. Pre-Commit Hook

Hooks are custom scripts set to fire off when certain important actions occur. There are two groups of these hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits. You can use these hooks for all sorts of reasons.

The pre-commit hook is run first, before you even type in a commit message. It's used to inspect the snapshot that's about to be committed, to see if you've forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code. Exiting non-zero from this hook aborts the commit, although you can bypass it with git commit –no-verify. You can do things like check for code style (run lint or something equivalent), check for trailing whitespace (the default hook does exactly this), or check for appropriate documentation on new methods.

PRECINCT implementation is a suite of bash scripts and the entry point of these scripts lies in the pre-commit hooks. Pre-commit hooks are easy to create and implement as depicted in Listing 1. This pre-hook is shipped with Git[1] which is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. From lines 3 to 11, the script identifies if the commit is the first one in order to select the revision to work against. Then, in lines 18 and 19, the script checks for trailing whitespace and fails if any are found.

Listing 1. Git Pre-Commit Hook Sample

```
#!/bin/sh                                        1
                                                 2
if git rev-parse --verify HEAD > \               3
 /dev/null 2>&1                                  4
then                                             5
 against=HEAD                                    6
else                                             7
 # Initial commit: diff against                  8
 # an empty tree object                          9
 against=4b825dc642....                          10
fi                                               11
                                                 12
# Redirect output to stderr.                     13
exec 1>&2                                        14
                                                 15
# If there are whitespace errors,                16
# print the offending file names and fail.       17
exec git diff-index --check \                    18
        --cached $against --                     19
```

For PRECINCT to work, we just have to add the call to our script suite instead or in addition of the whitespace check.

C. Extract and Save Blocks

A block is a set of consecutive lines of code that will be compared to all other blocks in order to identify clones. To achieve this critical part, we rely on TXL [20] which is a well-known first-order functional programming over linear term rewriting. For TXL to work, one has to write a grammar describing the syntax of the source language and the wanted transformations. TXL has three main phases: *parse, transform*, *unparse*. In the parse phase, the grammar is controlling not only the input but also the output form. Indeed, Listing 2 — extracted from the official documentation[2] — shows a grammar matching a *if-then-else* statement in C with some special keywords: [IN] (indent), [EX] (exdent) and [NL] (newline) that will be used for the output form.

[1]https://git-scm.com/
[2]http://txl.ca

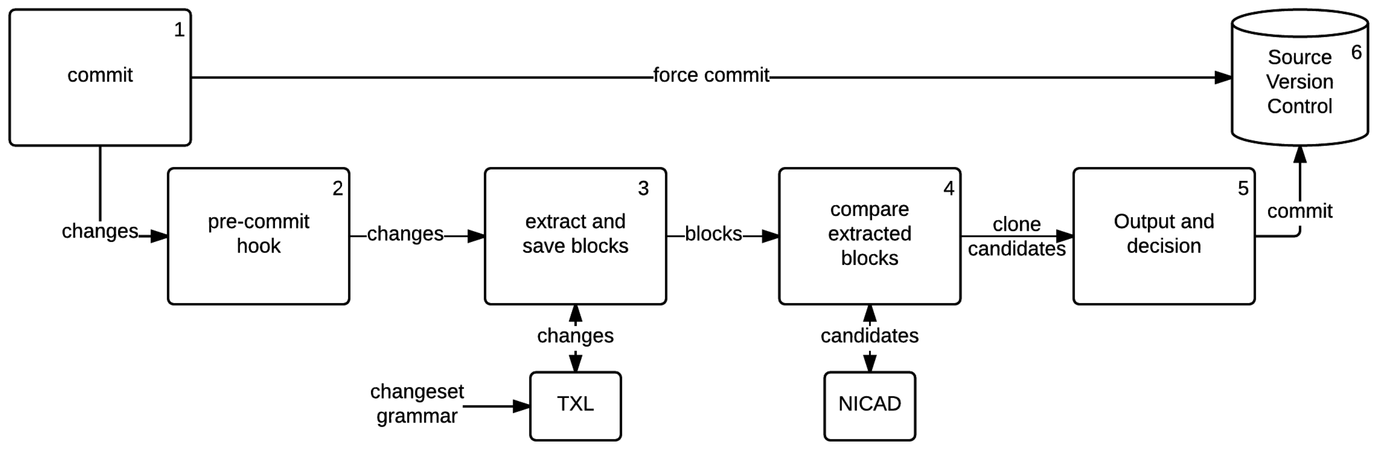

Fig. 1. Overview of the PRECINCT Approach.

Listing 2. Txl Sample Sample

```
define if_statement
  if ( [expr] ) [IN][NL]
     [statement] [EX]
     [opt else_statement]
end define

define else_statement
  else [IN][NL]
     [statement] [EX]
end define
```

Then, the *transform* phase will, as its name suggests, apply transformation rules that can, for example, normalize or abstract the source code. Finally, on the third phase of TXL, called *unparse*, unparses the transformed parsed input in order to output it.

Also, TXL supports what they creators calls Agile Parsing [21], which allow developer to redefined rule of the grammar and therefore, apply different rules than the original ones.

In PRECINCT we took advantage of that by redefining which blocks should be extracted from clone comparison and which blocks are out of scope in an effort to reduce the size (1) and ease the understandability of the output (2).

More specifically, at before each commit, we only extract the blocks belonging to the modified parts of the source code, henceforth greatly reducing the output size.

We have chosen TXL for several reasons. First, TXL is easy to install and to integrate to a developer workflow. Second, it was relatively easy to create a grammar that accepts commit input. Indeed, TXL is shipped with c, java, csharp, python and wsdl grammar that defines all the particularities of these languages and we could tweak them in order to accept changesets. Changesets are chunks of the modified source code that includes the added, modified and deleted lines.

Algorithm 1 presents an overview of the extract and save blocks operation.

```
Data: $Changeset$[] changesets;
  $Block$[] prior_blocks;
  $Boolean$ compare_history;
Result: Up to date blocks of the systems
1  for $i \leftarrow 0$ to $size\_of\ changesets$ do
2      Block[] blocks ← $extract\_blocks(changesets)$;
3      for $j \leftarrow 0$ to $size\_of\ blocks$ do
4          if not $compare\_history$ AND $blocks[j]$ overrides
           one of $prior\_blocks$ then
5              delete $prior\_block$;
6          end
7          write $blocks[j]$;
8      end
9  end
10 Function $extract\_blocks(Changeset\ cs)$
11     if $cs$ is unbalanced right then
12         $cs \leftarrow expand\_left(cs)$;
13     else if $cs$ is unbalanced left then
14         $cs \leftarrow expand\_right(cs)$;
15     end
17     return $txl\_extract\_blocks(cs)$;
```

Algorithm 1: Overview of the Extract Blocks Operation

This algorithm receives as arguments, the changesets, the blocks that have been previously extracted and a boolean named compare_history. Then, from lines 1 to 9 lies of for loop that iterates over the changesets For each changeset (lines 2), we extract the blocks by calling the $extract\_blocks(Changeset\ cs)$ function. In this function, we expand our changeset to the left and to the right in order to have a complete block. As depicted by Listing 3, changesets contain only the modified chunk of code and not necessarily complete blocks. Indeed, we have a block from line 3 to line 6 and deleted lines from line 8 to 14. However, in line 7 we can see the end of a block but we do not have the beginning of the said block. Therefore, we need to expand the changesets to the left in order to have synthetically correct blocks. We do so by checking block's beginning and ending, { and } in C for example. Then, we send these expanded changesets to txl for blocks extraction, formalization.

Listing 3. Changeset c4016c of monit

```
@@ -315,36 +315,6 @@
int initprocesstree_sysdep
    (ProcessTree_T **reference) {
        mach_port_deallocate(mytask,
          task);
    }
}
- if (task_for_pid(mytask, pt[i].pid,
-       &task) == KERN_SUCCESS) {
-     mach_msg_type_number_t    count;
-     task_basic_info_data_t
taskinfo;
-     thread_array_t
threadtable;
-     unsigned int
threadtable_size;
-     thread_basic_info_t
threadinfo;
```

For each extracted block, we check if the current block overrides (replace) a previous block (line 4). In such a case, we delete the previous block as it does not represent the current version of the program anymore (line 5). Also, we got an optional behavior of PRECINCT defined in line 4. Indeed, the compare_history is a condition to delete overridden blocks. We believe that overridden blocks have been so for a good reason (bug, default, removed features, ...) and if a newly inserted block matches an old one, it could be worth knowing to improve the quality of the system at hand. This feature is deactivated by default.

In summary, this step receives the files and lines modified by the latest changes of a developer and produce an up to date block representation of the system at hand. The blocks can be analyzed in the next step to discover potential clones.

D. Compare Extracted Blocks

In order to compare the extracted blocks and detect potential clones we could only rely on text-based techniques. Indeed, lexical and syntactic techniques would be of no help as they require a complete program to work, a program that compiles. In the relatively wide-range of tool and technics that exist to detect clone by considering code as text [2], [7], [8], [22]–[24], we choose NICAD [9] for several reasons. First NICAD is built on top of TXL and as we also use TXL on the previous step, the integration between the step was made easy. Second, NICAD, is able to detect all types of clones.

NICAD (Accurate Detection of Near-miss Intentional Clones) works in three phases: *Extraction*, *Comparison* and *Reporting*. During the *Extraction* phase in all potential clones are identified, pretty-printed and extracted. We do not use the *Extraction* phase of NICAD as it has been built to work on programs that are syntactically correct and changesets are not. We replaced NICAD's *Extraction* phase by our own processes described in the previous section.

TABLE I
PRETTY-PRINTING EXAMPLE [26]

Segment 1	Segment 2	Segment 3	S1 & S2	S1 & S3	S2 & S3
for (	for (	for (	1	1	1
i = 0;	i = 1;	j = 2;	0	0	0
i >10;	i >10;	j >100;	1	0	0
i++)	i++)	j++)	1	0	0
Total Matches			3	1	1
Total Mismatches			1	3	3

In the *Comparison* phase, extracted blocks are transformed, clustered and compared in order to find potential clones. Using TXL sub-programs, blocks go through a process called pretty-printing where they are stripped of formatting and comments. Indeed, when code cloned, some comments, indentation or spacing are changed according to the new context this code will be used in. This pretty-printing process ensures that all code will have the same spacing and formatting and ease the comparison. Furthermore, in the pretty-printing process, statements can be broken down into several lines. Table I shows how this can improve the accuracy of clone detection with three `for` statements, `for (i=0; i<10; i++)`, `for (i=1; i<10; i++)` and `for (j=2; j<100; j++)`. The pretty-printing allows NICAD to detect segment 1 and 2 as a clone pair because only the initialization of i changed. This specific example would not have been marked as a clone by other line based tools we tested such as Duploc [25]. In addition to the pretty-printing, code can be normalized and filtered to detect different classes of clones and match user preferences.

Finally, the extracted, pretty-printed, normalized and filtered blocks are marked as potential clones using a Longest Common Subsequence (LCS) algorithm [27]. Then, a percentage of unique statement can be computed and, depending a given threshold (see Section IV), the blocks are marked as clones.

The last step of NICAD, which acts as our clone comparison engine, is the *reporting*. However, we wanted PRECINCT to address the *massive output* concern and implemented our own reporting system which also falls into the *workflow* of developers. This reporting system is the subject of the next section.

As a summary, this step receives potentially expanded and balanced blocks from the extraction step. Then, the blocks are pretty-printed, normalized, filtered and fed to an LCS algorithm in order to detect potential clones.

E. Output and Decision

In this final step, we report the result of the clone detection with regards to the latest changes of the developer. The process here is very simple. Every change made by the developers has been through the previous steps and might have been marked as clones to another part of the system at hand. For each file that is suspected to contain a clone, one line is printed to the command line with the following options: (I) Inspect, (D) Disregard, (R) Remove from the commit as shown by Figure X.

(I) Inspect will cause a diff-like visualization of the suspected clones (Figure Y) while (D) disregard will simply ignore the finding. To integrate PRECINCT in the workflow of developer will also propose the (R) remove option. This option will simply remove the suspected file from the commit that is about to be sent to the central repository. Also, if the user types an option key twice, e.g. II, DD or RR, then, the option will be applied to files. For instance, if the developer type DD at any one point, all the findings will be disregarded and the commit will be allowed to go through.

IV. Experimentations

V. Conclusion

References

[1] B. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of 2nd Working Conference on Reverse Engineering*. IEEE Comput. Soc. Press, pp. 86–95. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=514697

[2] S. D. Stéphane Ducasse, Matthias Rieger, "A Language Independent Approach for Detecting Duplicated Code." [Online]. Available: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.6060

[3] M. Kim, V. Sazawal, and D. Notkin, "An empirical study of code clone genealogies," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, p. 187, sep 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1095430.1081737

[4] C. Kapser and M. Godfrey, ""Cloning Considered Harmful" Considered Harmful," in *2006 13th Working Conference on Reverse Engineering*. IEEE, oct 2006, pp. 19–28. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4023973

[5] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, may 2009, pp. 485–495. [Online]. Available: http://dl.acm.org/citation.cfm?id=1555001.1555062

[6] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, mar 2006. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1610609

[7] J. H. Johnson, "Visualizing textual redundancy in legacy source," p. 32, oct 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=782185.782217

[8] ——, "Identifying redundancy in source code using fingerprints," pp. 171–183, oct 1993. [Online]. Available: http://dl.acm.org/citation.cfm?id=962289.962305

[9] J. R. Cordy and C. K. Roy, "The NiCad Clone Detector," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, jun 2011, pp. 219–220. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5970189

[10] C. K. Roy and J. R. Cordy, "An Empirical Study of Function Clones in Open Source Software," in *2008 15th Working Conference on Reverse Engineering*. IEEE, oct 2008, pp. 81–90. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=4656397

[11] B. S. Baker, "A program for identifying duplicated code." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.550.4540

[12] B. S. Baker and R. Giancarlo, "Sparse Dynamic Programming for Longest Common Subsequence from Fragments," *Journal of Algorithms*, vol. 42, no. 2, pp. 231–254, feb 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0196677402912149

[13] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, jul 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=636188.636191

[14] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," p. 368, mar 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853341

[15] R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '00*. New York, New York, USA: ACM Press, jan 2000, pp. 155–169. [Online]. Available: http://dl.acm.org/citation.cfm?id=325694.325713

[16] R. Tairas and J. Gray, "Phoenix-based clone detection using suffix trees," in *Proceedings of the 44th annual southeast regional conference on - ACM-SE 44*. New York, New York, USA: ACM Press, mar 2006, p. 679. [Online]. Available: http://dl.acm.org/citation.cfm?id=1185448.1185597

[17] R. Falke, P. Frenzel, and R. Koschke, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, jul 2008. [Online]. Available: http://link.springer.com/10.1007/s10664-008-9073-9

[18] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" pp. 672–681, may 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486877

[19] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in *Proceedings International Conference on Software Maintenance*. IEEE Comput. Soc, pp. 314–321. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5726968

[20] J. R. Cordy, "Source transformation, analysis and generation in TXL," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM '06*. New York, New York, USA: ACM Press, jan 2006, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1111542.1111544

[21] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider, "Agile Parsing in TXL," *Automated Software Engineering*, vol. 10, no. 4, pp. 311–336. [Online]. Available: http://link.springer.com/article/10.1023/A%3A1025801405075

[22] A. Marcus and J. Maletic, "Identification of high-level concept clones in source code," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE Comput. Soc, pp. 107–114. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=989796

[23] U. Manber, "Finding similar files in a large file system," p. 2, jan 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267074.1267076

[24] R. Wettel and R. Marinescu, "Archeology of code duplication: recovering duplication chains from small duplication fragments," in *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*. IEEE, 2005, p. 8 pp. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1595830

[25] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*. IEEE, 1999, pp. 109–118. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=792593

[26] N. E. A. R. Iss and S. O. C. Lones, "D Etection and a Nalysis of," no. August, 2009.

[27] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, may 1977. [Online]. Available: http://dl.acm.org/citation.cfm?id=359581.359603