PRECINCT: An Incremental Approach for Preventing Clone Insertion at Commit Time

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 since they may cause the same buggy code to appear in multiple places in the code, making software maintenance and evolution tasks challenging. Clone detection has been an active research field for almost two decades. Interestingly, most existing techniques focus on detecting clones after they are inserted in the code. In this paper, we take another look at the clone detection problem by designing a novel approach for preventing the insertion of clones in the first place. Our approach, called PRECINCT (PREventing Clones INsertion at Commit Time), detects efficiently near-miss software clones at commit time by means of pre-commit hooks. This way, changes to the code are analysed and suspicious copies are flagged before they reach the central code repository in the version control system. The application of PRECINCT to seven systems developed independently shows that PRECINCT achieves 100% precision and 93% recall.**

I. Introduction

Code clones appear when developers reuse code with little to no modification to the original code. Studies have shown that clones can account for about 7% to 50% of code in a given software system [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 since they can introduce new bugs in the code [4]–[6]. If a bug is discovered in one segment of the code that has been copied and pasted several times, then the developers will have to remember the places where this segment has been reused in order to fix the bug in each of them.

In the last two decades, there have been many studies and tools that aim at detecting clones. They can be grouped into three categories. The first category includes techniques that treat the source code as text and use transformation and normalization to compare various code fragments [7]–[10]. The second category includes methods that use lexical analysis, where the source code is sliced into sequences of tokens, similar to the way a compiler operates [1], [6], [11]–[13]. The tokens are used to compare code fragments. Finally, syntactic analysis has also been performed 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].

Despite the advances in clone detection, the use of exiting clone detection tools is not as widespread as one might think. The main factors that contribute to this are summarized in what follows [18]: These tools are known to output a large number of data, making it hard to understand and analyse their results. They tend to have a high amount of false positives. They are also hard to configure and do not integrate well with the day-to-day workflow of a developer.

In this paper, we present PRECINCT (PREventing Clones INsertion at Commit Time) that focuses on preventing the insertion of clones in the first place at commit time, more particularly by using pre-commit hooks capabilities of modern source code version control systems. A pre-commit hook is a process that one can implement to receive the latest modification to the source code done by a given developer just before the code reaches the central repository. PRECINCT intercepts this modification and analyses its content to see whether a suspicious clone has been introduced or not. A flag is raised if a code fragment is suspected to be a clone of an exiting code segment. In fact, PRECINCT itself is 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. This said, only a fraction of the code is analysed and PRECINCT reduces the output by 70%, compared to leading clone detection techniques such as NICAD (Accurate Detection of Near-miss Intentional Clones) [9]. Moreover, the detected clones are presented using a classical 'diff' output that developers are familiar with. PRECINCTS is easy to use and configure. It only takes one command line to configure PRECINCT. PRECINCT is also well integrated with the workflow of the developers since it leverages the pre-commit hook capability of modern source code version control systems such as Git and Jira.

We evaluated the effectiveness of PRECINCT using precision and recall on seven systems developed independently with a wide range of technologies. The results show that PRECINCT 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 **??**, we present the studies related to PRECINCT, with a particular emphasis on NICAD [9], used to build PRECINCT. Then, in Section III, we present the PRECINCT approach. The evaluation of PRECINCT is the subject of Section IV. Finally, we propose concluding remarks in Section V.

II. Related Works

[19]

III. The PRECINCT Approach

The PRECINCT approach is composed of six steps. 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 the 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 has access to the changes in terms of files that have been modified, more specifically, the lines that have 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 candidate clones using NICAD [9]. Finally, the output of NICAD is further refined and presented to the user for a decision round. These steps are discussed in more detail in the following subsections.

A. Commit

In version control systems, a commit adds the latest changes made to the source code to the repository, making these changes part of the head revision of the repository. 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 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 hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, whereas server-side hooks run on network operations such as receiving pushed commits. These hooks can be used for all sorts of reasons (GIVE ONE OR TWO EXAMPLES).

The pre-commit hook is run first, before one even types in a commit message. It is used to inspect the snapshot that is about to be committed. This is used to allow developers to check if they have not forgotten anything, to run tests, or to examine whatever the need to inspect the code. Exiting [WAHAB: I DO NOT UNDERSTAND THIS SENTENCE] non-zero from this hook aborts the commit, although one can bypass it with a no-verify commit (a feature that exists in Git and other version systems). Developers can do things like check for code style (run lint or something equivalent), check for trailing white spaces (the default hook does exactly this), or check for appropriate documentation on new methods.

PRECINCT is a set of bash scripts where 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. The pre-hook is shipped in version control systems like Git[1]. Note that even though we use Git as the main version control to present PRECINCT, we believe that the techniques presented in this paper are readily applicable to other version control systems. 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 of PRECINCT, we rely on TXL [20], which is a first-order functional programming over linear term rewriting, developed by Cordy et al. [20]. For TXL to work, one has to write a grammar describing the syntax of the source language and the transformations needed. TXL has three main phases: *parse, transform*, *unparse*. In the parse phase, the grammar controls not only the input but also the output form. 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

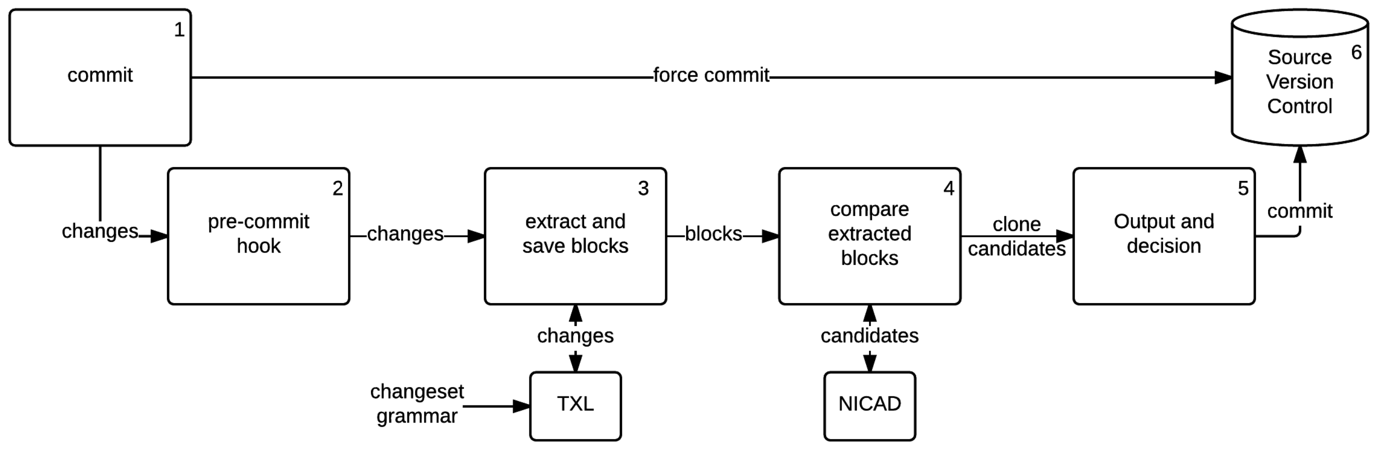

Fig. 1. Overview of the PRECINCT Approach.

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 the name suggests, apply transformation rules that can, for example, normalize or abstract the source code. Finally, the third phase of TXL, called *unparse*, unparses the transformed parsed input in order to output it. Also, TXL supports what the creators call Agile Parsing [21], which allow developers to redefine rules of the grammar and, therefore, apply different rules than the original ones.

PRECINCT takes advantage of that by redefining the blocks that should be extracted for the purpose of clone comparison, leaving out the blocks that are out of scope. More precisely, before each commit, we only extract the blocks belonging to the modified parts of the source code, hence reducing the size of the output, a common issue of clone detection techniques as discussed in the introductory section.

We have selected TXL for several reasons. First, TXL is easy to install and to integrate with a developer workflow. Second, it was relatively easy to create a grammar that accepts commits as input. This is because TXL is shipped with C, Java, Csharp, Python and WSDL grammars that define all the particularities of these languages, with the ability to customize these grammar to to accept changesets (chunks of the modified source code that includes the added, modified, and deleted lines) instead of the whole code.

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

```
**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 $\leftarrow extract\_blocks(changesets)$;
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 lie the for loop that iterates over the changesets. For each changeset (Line 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 its beginning. Therefore, we need to expand the changeset to the left in order to have syntactically 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 block extraction and 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 (replaces) 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 have an optional step in PRECINCT defined in Line 4. The compare_history is a condition to delete overridden blocks.

[WAHAB: I DON'T UNDERSTAND THE FOLLOWING 3 SENTENCES. PLEASE REPHRASE] We believe that overridden blocks have been so far 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 made by the developer and produces an up to date block representation of the system at hand. The blocks can be analysed in the next step to discover potential clones.

D. Compare Extracted Blocks

In order to compare the extracted blocks and detect potential clones we can only resort to text-based techniques. This is because lexical and syntactic analysis approaches (alternatives to text-based compariosn) would require a complete program to work, a program that compiles. In the relatively wide-range of tool and techniques that exist to detect clones by considering code as text [2], [7], [8], [22]–[24], we select NICAD as the main text-based method for comparing clones [9] for several reasons. First, NICAD is built on top of TXL, which we also used TXL in the previous step. Second, NICAD is able to detect all types of clones.

NICAD works in three phases: *Extraction*, *Comparison* and *Reporting*. During the *Extraction* phase all potential clones are identified, pretty-printing, and extracted. We do not use the *Extraction* phase of NICAD as it has been built to work on programs that are syntactically correct, which is not the case of changesets. We replaced NICAD's *Extraction* phase by our own tools, 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. When code fragments are cloned, some comments, indentation or spacing are changed according to the new context where the new code is used. 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 Segments 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 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 statements can be computed and, depending on 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, to prevent PRECINCT from outputting a large number of data (an issue from which many clone detection techniques face), we implemented our own reporting system, which is also well embedded with wthe 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 at commit time with respect to the latest changes made by the developer. The process is straightforward. Every change made by the developers has been through the previous steps and might have been marked as a potential clone. 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 the developer we also propose the remove option (R). 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 types DD at any one point, the results output by PRECINCT will be disregarded and the commit will be allowed to go through. We believe that this simple mechanism will encourage developers to use PRECINCT like they would use any other feature of Git (or any other control version system).

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