PRECINCT: An Approach for Preventing Clone Insertion at Commit Time

Mathieu Nayrolles
SBA Lab, ECE Dept, Concordia University
Montréal, QC, Canada
mathieu.nayrolles@concordia.ca

Abdelwahab Hamou-Lhadj
SBA Lab, ECE Dept, Concordia University
Montréal, QC, Canada
wahab.hamou-lhadj@concordia.ca

***Abstract*—Software clones are generally 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. Existing techniques can be categorized based on whether they detect clones after they reach the central code repository or by having software developers resort to external tools in an IDE. In this paper, we take another look at the clone detection problem by designing a new approach for preventing the insertion of clones at commit-time. We argue that the detection of clones at commit-time integrates well with a developer's workflow. Our approach, called PRECINCT (PREventing Clones INsertion at Commit Time), detects efficiently Type 3 software clones at commit-time using pre-commit hooks. This way, changes to the code are analysed, and suspicious clones are flagged before they reach the central code repository in the version control system. The application of PRECINCT to three systems shows that PRECINCT can prevent the insertion of up to 97.7% of the clones at commit time.**

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 up to 50% of code in a given software system [1], [2]. Although, developers often reuse code on purpose [3], clones are generally considered as a bad practice in software development since they may introduce bugs [4]–[6].

In the last two decades, there have been many studies and tools that aim at detecting clones. They can be grouped into two categories depending on whether they operate locally on a developer's workstation (e.g., [7], [8]) or remotely on server (e.g., [9], [10]).

Local clone detection approaches are typically implemented as IDE plugins or external tools. IDE-based methods tend to issue many warnings to developers that may interrupt their work, hence hindering their productivity [11].

Developers may be reluctant to use external tools unless they are involved in a major refactoring effort. These tools cause overhead because of the context switch they incur [12]–[14]. This problem is somehow similar to the problem of adopting bug identification tools. Literature shows that these tools are challenging to use because they do not integrate well with the day-to-day workflow of developers [15]–[23]. The problem with remote approaches is that the detection occurs too late in the development process. Once the clones reach the central repository, they can be pulled by other members of the development team, further complicating the removal and management of clones.

In this paper, we present PRECINCT (PREventing Clones INsertion at Commit Time) that focuses on preventing the insertion of clones at commit-time (i.e., before they reach the central code repository). PRECINCT is a trade-off between local and remote approaches. The approach relies on the use of 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, source code version control systems such as Git,[1] if a code fragment is suspected to be a clone of an existing code segment. This said, only a fraction of the code is analysed, in an incremental manner, making PRECINCT computationally efficient.

To the best of our knowledge, PRECINCT is the first clones detection technique that operates at commit-time. In this study, we focus on Type 3 clones as they are more challenging to detect [24]–[26]. Since Type 3 clones include Type 1 and 2 clones, then these types could be detected by PRECINCT as well. We evaluated the effectiveness of PRECINCT on three systems, written in C and Java. The results show that PRECINCT prevents Type 3 clones to reach the final source code repository with an average accuracy of 97.7%.

The rest of this paper is organized as follows: In Section II, we present the studies related to PRECINCT. Then, in Section III, we present the PRECINCT approach. The evaluation of PRECINCT is the subject of Section IV, followed by threats to validity. Finally, we conclude the paper in Section VI.

II. Related Work

Clone detection is an important and difficult task. Throughout the years, researchers and practitioners have

[1]https://git-scm.com/

developed a considerable number of methods and tools to detect efficiently source code clones. In this section, we first describe some classical techniques for code clone detection and then present noticeable local- and remote-based approaches.

A. Techniques

Tree-matching and metric-based methods are two sub-categories of syntactic analysis for clone detection. Syntactic analyses consist of building abstract syntax trees (AST) and analyze them with a set of dedicated metrics or searching for identical sub-trees. Many existing AST-based approaches rely on sub-tree comparison to detect clone, including the work of Baxter et al.[27], Wahleret et al. [28], and the work of Jian et al. with Deckard [29]. An AST-based approach compares metrics computed on the AST, rather than the code itself, to identify clones [30], [31].

Text-based techniques use the code and compare sequences of code blocks to each other to identify potential clones. Johnson was perhaps the first one to use fingerprints to detect clones [32], [33]. Blocks of code are hashed, producing fingerprints that can be compared. If two blocks share the same fingerprint, they are considered as clones. Manber et al. [34] and Ducasse et al. [35] refined the fingerprint technique by using leading keywords and dot-plots.

Another approach for detecting clones is to use static analysis and to leverage the semantics of a program to improve the detection. These techniques rely on program dependency graphs, where nodes are statements and edges are dependencies. Then, the problem of finding clones is reduced to the problem of finding identical sub-groups in the program dependency graph. Examples of existing techniques that fall into this category are the ones presented by Krinke et al.[36] and Gabel et al. [37].

Many clone detection tools resort to lexical approaches for clone detection. Here, the code is transformed into a series of tokens. If sub-series repeat themselves, it means that a potential clone is in the code. Some popular tools that use this technique include Dup [1], CCFinder [38], and CP-Miner [6].

In 2010, Hummel *et al.* proposed an approach that is both incremental and scalable using index-based clone detection [39]. Incremental clone detection is a technique where only the changes from one version to another are analysed. Thus, the computational time is greatly reduced. Using more than 100 machines in a cluster, they managed to drop the computation time of Type 1 and 2 to less than a second while comparing a new version. The time required to find all the clones on a 73 MLOC system was 36 minute. We reach similar performances, for one revision, using a single machine. While being extremely fast and reliable, Hummel *et al.*'s approach required an industrial cluster to achieve such performance. In our opinion, it is unlikely that standard practitioners have access to such computational power. Moreover, the authors' approach only targets Type 1 and 2 clones. Higo *et al.* proposed an incremental clone detection approach based on program dependency graphs (PDG) [40]. Using PDG is arguably more complex than text comparison and allows the detection of clone structures that are scattered in the program. They were able to analyze 5,903 revisions in 15 hours in Apache Ant.

B. Remote Implementation

Yuki *et al* conducted one of the few studies on the application of clone management to industrial systems [9].They implemented a tool named Clone Notifier at NEC with the help of experienced practitioners. They specifically focus on clone insertion notification, very much like PRECINCT. Unlike PRECINCT, their approach uses a remote approach in which the changes are committed (i.e., they reach the central repository, and anyone can pull them into they machines) and a central server analyses the changes. If the committed changes contain newly inserted clones, then an email notification is sent. We first argue that detecting clones in a remote fashion can be damaging for the project as a clone can be synchronized by other team members, which can lead to challenging merges when the clones are removed. Secondly, the authors did not report any performance measurements and the longer it takes for the notification to be sent to the developer, the harder it can be to reconstruct the mind-map required for clone removal. PRECINCT, however, is able to perform an incremental online detection in few seconds.

Zhang et al proposed CCEvents (Code Cloning Events) [10]. Their approach monitors code repository continuously and allows stockholders to use a domain specific language called CCEML to specify which notifications they wish to receive.

In addition, many commercial tools now include clone detection as part of continuous integration. Codeclimate,[2] Codacy,[3] Scrutinizer[4] and Coveralls[5] are some noticeable examples. These tools will perform various tasks such as executing unit test suite, computing quality metrics and, performing a clone detection and provide a report by email. The techniques they use for clone detection is not known.

Remote approaches do not hinder the productivity of developers, but we argue that the detection intervenes too late in the development process as the clones have reached the central repository and can be pulled by other members of the organization. Consequently, they can use these clones in their code and make their removal even more challenging. Finally, notifications from remote approaches arrive too late as developers will likely have move to another task.

[2]https://codeclimate.com/
[3]https://codacy.com/
[4]https://scrutinizer-ci.com/
[5]https://coveralls.io/

C. *Local Implementation*

1) IDE: Gode and Koschke [41] proposed an incremental clone detector that relies on the results of analysis from past versions of a system to only analyze the new changes. Their clone detector takes the form of an IDE plugin that alerts developers as soon as a clone is inserted into the program.

Zibran and Roy proposed another IDE-based clone management system to detect and refactor near-miss clones for Eclipse [7], [42]. Their approach uses a k-difference hybrid suffix tree algorithm and is able to detect clones in real-time and propsoe a semi-automated refactoring process.

Robert el al proposed another IDE plugin for Eclipse called CloneDR based on ASTs that introduced novel visualization for clone detection such as scatter-plots [43].

As we stated in the introduction, IDE-based methods tend to issue many warnings to developers that may interrupt their work, hence hindering their productivity [11]. Indeed, Latoza et al found that it exists six different reasons to duplicate code ((a) separate developers implement same functionality, (b) copy and paste of example code, (c) design decision distributed over multiple methods, (d) copy of other team's code base, (e) branch maintained separately, (f) reimplementation by same developer in different language) and developers are aware that they are creating clones in five out of the six situations (b, c, d, e, and f) [44]. Consequently, warnings provided by IDE-based local detection are truly useful in the unlikely case of two developers re-implementing the same functionality.

PRECINCT operates at commit-time, and hence we believe it does not hinder the productivity of developers while being well-integrated with their workflow.

2) External Tools: Tung *et al* proposed an advanced clone-aware source code management system called Clever. Their approach uses abstract syntax trees to detect, update, and manage clones. While efficient, their approach does not prevent the introduction of clones, and it is not incremental. Developers have to run a project-wide detection for each commit of a version of the program. The same teams [45] conducted follow-up study by making Clever incremental. Their new tool, JSync, is an incremental clone detector that will only perform the detection of clones on the new changes. While the new version of their work is incremental, it is different from PRECINCT is the sense that their approach relies on developers to run the approach manually to detect clones, potentially leading to a loss of productivity as described in the previous section.

Niko *el al.* proposed techniques revolving around hashing to obtain a quick answer while detecting Type 1, Type 2, and Type 3 clones in Squeaksource [46]. While their approach works on a single system (i.e., detecting clones on one version of one system), they found that more than 14% of all clones are copied from project to project, stressing the need for fast and scalable approaches for clone detection to detect clone across a large number of projects. On the performance side, Niko *el al.* were able to perform clone detection on 74,026 classes in 14:45 hours (4,747 class per hour) with a 8 core Xeon at 2.3 GHz with 16 GB of RAM. While these results are promising, especially because the approach detects clones across projects and versions, the computing power required is still considerable.

Similarly, Saini *et al* and Sajnani *et al* proposed an approach, called SourcererCC [8], [47]. SourcererCC targets fast clone detection on developers' workstation (12 GB RAM). SourcererCC is a token-based clone detector that uses an optimized inverted-index. It was tested on 25K projects cumulating 250 MLOC. The technique achieves a precision of 86% and a recall of 86%-100% for clones of Type 1, 2 and 3.

Toomey *el al.* also proposed an efficient token based approach for detecting clones called ctcompare [48]. Their tokenization is, however, different than most approaches as they used lexical analysis to produce sequences of tokens that can be transformed into token tuples. ctcompare is accurate, scalable and fast but does not detect Type 3 clones.

PRECINCT aims to prevent clone insertion at commit-time. This way, software developers do not have to resort to external tools or IDE plugins to remove clones after they are inserted. Our approach notifies software developers of possible clones as they commit their code. It is our opinion that clone detection should be part of the "Just-In-Time Quality Assurance" [49] movement where defect prediction models identify risky changes as soon as they are committed. This allows for less time-consuming approaches where developers can analyse risky changes while they are still fresh in their minds rather than being assigned a list of risky packages/class to review by the quality assurance team. Zibran recently proposed an infrastructure for integrating clone management that covers clone detection and refactoring on developers' workstations supported by a remote server infrastructure [50]. It could, if implemented, fit in the category of "Just-In-Time Clone Detection." This work is close to ours in the sense that the authors propose an incremental clone detection that integrates well with the workflow of developers. We argue, however, that our approach is simpler to implement and uses directly the code versioning tool rather than yet another plugin in the developer's IDE. Finally, our approach has been implemented and tested, while the approach presented by Zibran *et al.* is only conceptual.

III. The PRECINCT Approach

The PRECINCT approach is composed of six steps. The first and last steps are typical steps that a developer would do when committing code. The first step is the commit step where developers send their latest changes 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 regarding the files that have

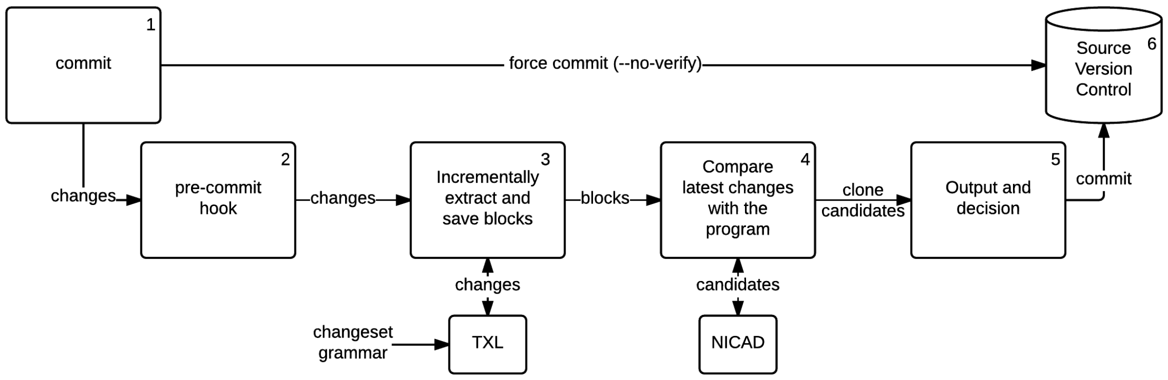

Fig. 1: Overview of the PRECINCT approach.

been modified, more specifically, the lines that have been modified. The modified lines of the files are sent to TXL [51] for block extraction. Then, the blocks are compared to previously extracted blocks to identify candidate clones using the comparison engine of NICAD [52]. We chose NICAD engine because it has been shown to provide high accuracy [52]. The tool is also readily available, easy to use, customizable, and works with TXL. Note, however, that PRECINCT can also work with other engines for comparing code fragments. Finally, the output of NICAD is further refined and presented to the user for decision. 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 critical 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 such as compliance with the coding rules or automatic run of unit test suites.

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. Depending on the exit status of the hook, the commit will be aborted and not pushed to the central repository. Also, developers can choose to ignore the pre-hook. In Git, for example, they will need to use the command git commit –no−verify instead of git commit. This can be useful in case of an urgent need for fixing a bug where the code has to reach the central repository as quickly as possible. Developers can do things like check for code style, 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. 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.

C. Extract and Save Blocks

A block is a set of consecutive lines of code that will be compared to all other blocks to identify clones. To achieve this critical part of PRECINCT, we rely on TXL [51], which is a first-order functional programming over linear term rewriting, developed by Cordy et al.[51]. 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 1 — extracted from the official documentation[6] — shows a grammar matching an *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.

Listing 1: Txl Sample

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

define else_statement
```

[6]http://txl.ca

```
    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 to output it. Also, TXL supports what the creators call Agile Parsing [53], which allows developers to redefine the 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, we only process, in an incremental manner, the latest modification of the source code instead of the source code as a whole.

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

Algorithm 1 presents an overview of the "extract" and "save" blocks operations of PRECINCT. 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 the right to have a complete block.

Listing 2: Changeset c4016c of monit

```
 1  @@ -315,36 +315,6 @@
 2  int initprocesstree_sysdep
 3     (ProcessTree_T **reference) {
 4        mach_port_deallocate(mytask, task);
 5     }
 6  }
 7  - if (task_for_pid(mytask, pt[i].pid,
 8  -  &task) == KERN_SUCCESS) {
 9  -     mach_msg_type_number_t    count;
10  -     task_basic_info_data_t    taskinfo;
```

As depicted by Listing 2, 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 to have syntactically correct blocks. We do so by checking the block's beginning and ending (using { and }) in C for example. Then, we send these expanded changesets to TXL for block extraction and formalization.

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.

We believe that deleted blocks have been removed 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 made by the developer and produces an up to date block representation of the system at hand in an incremental way. The blocks are analyzed in the next step to discover potential clones.

```
   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 ← expand_left(cs);
13     else if cs is unbalanced left then
14         cs ← expand_right(cs);
15     end
17     return txl_extract_blocks(cs);
```

Algorithm 1: Overview of the Extract Blocks Operation

D. Compare Extracted Blocks

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 comparisons) would require a complete program to work, a program that compiles. In the relatively wide-range of tools and techniques that exist to detect clones by considering code as text [2], [32]–[34], [54], [55], we selected NICAD as the main text-based

TABLE I: Pretty-Printing Example

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

method for comparing clones [52] for several reasons. First, NICAD is built on top of TXL, which we also used in the previous step. Second, NICAD can detect Type 1, 2 and 3 clones.

NICAD works in three phases: *Extraction*, *Comparison* and *Reporting*. During the *Extraction* phase 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, which is not the case for changesets. We replaced NICAD's *Extraction* phase with our scripts, described in the previous section.

In the *Comparison* phase, extracted blocks are transformed, clustered and compared 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, which renders the comparison of code fragments easier. Furthermore, in the pretty-printing process, statements can be broken down into several lines. Table I [56] 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 two as a clone pair because only the initialization of *i* is changed. This specific example would not have been marked as a clone by other tools we tested such as Duploc [35]. 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 [57]. Then, a percentage of unique statements can be computed and, depend 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 amount of data (an issue that many clone detection techniques face), we implemented our reporting system, which is also well embedded with 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 to detect potential clones. Moreover, the clone detection in PRECINCT is less intensive than NICAD because we only compare the latest changes with the rest of the program instead of comparing all the blocks with each other.

TABLE II: List of Target Systems in Terms of Files and Kilo Line of Code (KLOC) at current version and Language

SUT	Revisions	Files	KLoC	Language
Monit	826	264	107	C
Jhotdraw	735	1984	44	Java
dnsjava	1637	233	47	Java

E. Output and Decision

In this final step, we report the result of the clone detection at commit time on the latest changes made by the developer. The process is straightforward. Every change made by the developer goes through the previous steps and is checked for the introduction of potential clones. 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. In comparison to this straightforward and interactive output, NICAD outputs each and every detail of the detection result such as the total number of potential clones, the total number of lines, the total number of unique line text chars, the total number of unique lines, and so on. We think that so many details might make it hard for developers to react to these results. A problem that was also raised by Johnson et al. [20] when examining bug detection tools. Then the potential clones are stored in XML files that can be viewed using an Internet browser or a text editor.

(I) Inspect will cause a diff-like visualization of the suspected clones 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.

IV. Case Study

In this section, we show the effectiveness of PRECINCT for detecting clones at commit time in three open source systems.

The aim of the case study is to answer the following question: *Can we detect clones at commit time, i.e., before they are inserted into the final code, and if so, what would be the accuracy?*

A. Target Systems

Table II shows the systems used in this study and their characteristics in terms of the number files they contain and the size in KLoC (Kilo Lines of Code). We also include the number of revisions used for each system and the programming language in which the system is written.

Monit[7] is a small open source utility for managing and monitoring Unix systems. Monit is used to conduct

[7]https://mmonit.com/monit/

automatic maintenance and repair and supports the ability to identify causal actions to detect errors. This system is written in C and composed of 826 revisions, 264 files, and the latest version has 107 KLoC. We have chosen Monit as a target system because it was one of the systems NICAD was tested on.

JHotDraw[8] is a Java GUI framework for technical and structured graphics. It has been developed as a "design exercise". Its design is largely based on the use of design patterns. JHotDraw is composed of 735 revisions, 1984 files, and the latest revision has 44 Kloc. It is written in Java, and it is often used by researchers as a test bench. JHotDraw was also used by NICAD's developers to evaluate their approach.

Dnsjava[9] is a tool for implementing the DNS (Domain Name Service) mechanisms in Java. This tool can be used for queries, zone transfers, and dynamic updates. It is not as large as the other two, but it still makes an interesting case subject because it has been well maintained for the past decade. Also, this tool is used in many other popular tools such as Aspirin, Muffin and Scarab. Dnsjava is composed of 1637 revisions, 233 files; the latest revision contains 47 Kloc. We have chosen this system because we are familiar with it as we used it before [58].

B. Process

As our approach relies on commit pre-hooks to detect possible clones during the development process (more particularly at commit time), we had to find a way to *replay* past commits. To do so, we *cloned* our test subjects, and then created a new branch called *PRECINCT_EXT*. When created, this branch is reinitialized at the initial state of the project (the first commit), and each commit can be replayed as they have originally been. At each commit, we store the time taken for PRECINCT to run as well as the number of detected clone pairs. We also compute the size of the output in terms of the number of lines of text output by our method. The aim is to reduce the output size to help software developers interpret the results.

To validate the results obtained by PRECINCT, we needed to use a reliable clone detection approach to extract clones from the target systems and use these clones as a baseline for comparison. For this, we turned to NICAD because of its popularity, high accuracy, and availability [52]. This means, we run NICAD on the revisions of the system to obtain the clones then we used NICAD clones as a baseline for comparing the results achieved by PRECINCT.

It may appear strange that we are using NICAD to validate our approach, knowing that our approach uses NICAD's code comparison engine. In fact, what we are assessing here is the ability for PRECINCT to detect clones at commit time using changesets. The major part of PRECINCT is the capacity to intercept code changes and build working code blocks that are fed to a code fragment comparison engine (in our case NICAD's engine). PRECINCT can be based on any other code comparison engine.

We show the result of detecting Type 3 clones with a maximum line difference of 30% as discussed in Table III. As discussed in the introductory section, we chose to report on Type 3 clones because they are more challenging to detect than Type 1 and 2. PRECINCT detects Type 1 and 2 too, so does NICAD. For the time being, PRECINCT is not designed to detect Type 4 clones. These clones use different implementations. Detecting Type 4 clones is part of future work.

We assess the performance of PRECINCT in terms of precision, recall, and F_1-measure by using NICAD's results as ground truth. They are computed using TP (true positives), FP (false positives), FN (false negatives), which are defined as follows:

- TP: is the number of clones that were properly detected by PRECINCT (again, using NICAD's results as baseline)
- FP: is the number of non-clones that were classified by PRECINCT as clones
- FN: is the number of clones that were not detected by PRECINCT
- Precision: TP / (TP + FP)
- Recall: TP / (TP + FN)
- F1-measure: 2.(precision.recall)/(precision+recall)

C. Results

Table III summarizes PRECINCT's results in terms of precision, recall, F_1-measure, execution time and output reduction for our three subject systems: Monit, JHotDraw, and Dnsjava. The first version of Monit contains 85 clone pairs, and this number stays stable until Revision 100. From Revision 100 to 472 the detected clone pairs vary between 68 and 88 before reaching 219 at Revision 473. The number of clone pairs goes down to 122 at Revision 491 and decreases to 128 in the last revision. PRECINCT was able to detect 96.1% (123/128) of the clone pairs that are detected by NICAD with a 100% recall. It took in average around 1 second for PRECINCT to execute on a Debian 8 system with Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz, 8Gb of DDR3 memory. It is also worth mentioning that the computer we used is equipped with SSD (Solid State Drive). This impacts the running time as clone detection is a file intensive operation. Finally, the PRECINCT was able to output 88.3% fewer lines than NICAD.

JHotDraw starts with 196 clone pairs at Revision 1 and reaches a pick of 2048 at Revision 180. The number of clones continues to go up until Revisions 685 and 686 where the number of pairs is 1229 before picking at 6538 and more from Revisions 687 to 721. PRECINCT was able

[8]http://www.jhotdraw.org/
[9]http://www.dnsjava.org/

TABLE III: Overview of PRECINCT's results in terms of precision, recall, F_1-measure, execution time and output reduction.

	Detected	Precision	Recall	F1-measure	NICAD's Average Execution Time	PRECINCT's Average Execution Time	Overall Output Reduction
Monit	123	96.1%	100%	98%	2.2s	0.9s	88.3%
JHotDraw	6490	98.3%	100%	99.1%	5.1s	1.7s	70.1%
DnsJava	226	82.8%	100%	90.6%	1.8s	1.1s	88.6%
Total	6839	97.7%	100%	98.8%	3s	1.2s	83.4%

to detect 98.3% of the clone pairs detected by NICAD (6490/6599) with 100% recall while executing on average in 1.7 seconds (compared to 5.1 seconds for NICAD). With JHotDraw, we can clearly see the advantages of incremental approaches. Indeed, the execution time of PRECINCT is loosely impacted by the number of files inside the system as the blocks are constructed incrementally. Also, we only compare the latest change to the remaining of the program and not all the blocks to each other as NICAD. We also were able to reduce by 70.1% the number of lines output by NICAD.

Finally, for Dnsjava, the number of clone pairs starts high with 258 clones and goes up until Revision 70 where it reaches 165. Another quick drop is observed at Revision 239 where we found only 25 clone pairs. The number of clone pairs stays stable until Revision 1030 where it reaches 273. PRECINCT was able to detect 82.8% of the clone pairs detected by NICAD (226/273) with 100% recall while executing on average in 1.1 seconds while NICAD took 3 seconds in average. PRECINCT outputs 83.4% fewer lines of code than NICAD.

Overall, PRECINCT prevented 97.7% of the 7000 clones (in all systems) to reach the central source code repository while executing more than twice as fast as NICAD (1.2 seconds compared to 3 seconds in average)and reducing the output in terms of lines of text output the developers by 83.4% in average. Note here that we have not evaluated the quality of the output of PRECINCT compared to NICAD's output. We need to conduct user studies for this. We are, however, confident, based on our experience trying many clone detection tools, that a simpler and a more interactive way to present the results of a clone detection tool is warranted. PRECINCT aims to do just that.

The difference in execution time between NICAD and PRECINCT stems from the fact that, unlike PRECINCT, NICAD is not an incremental approach. For each revision, NICAD has to extract all the code blocks and then compares all the pairs with each other. On the other hand, PRECINCT only extracts blocks when they are modified and only compares what has been changed with the rest of the program.

The difference in precision between NICAD and PRECINCT (2.3%) can be explained by the fact that sometimes developers commit code that does not compile. Such commits will still count as a revision, but TXL fails to extract blocks that do not comply with the target language syntax. While NICAD also fails in such a case, the disadvantage of PRECINCT comes from the fact that the failed block is saved and used as a reference until it is changed by a correct one in another commit.

V. Threats to Validity

The selection of target systems is one of the common threats to validity for approaches aiming to improve the analysis of software systems. It is possible that the selected programs share common properties that we are not aware of and therefore, invalidate our results. However, the systems analyzed by PRECINCT are the same as the ones used in similar studies. Moreover, the systems vary in terms of purpose, size, and history.

Also, we see a threat to validity that stems from the fact that we only used open source systems. The results may not be generalizable to industrial systems. We intend to undertake these studies in future work.

The programs we used in this study are all based on the Java, C, and Python programming languages. This can limit the generalization of the results. However, similar to Java, C, Python, if one writes a TXL grammar for a new language — which can be a relatively hard work — then PRECINCT can work since PRECINCT relies on TXL.

Finally, we use NICAD as the code comparison engine. The accuracy of NICAD affects the accuracy of PRECINCT. This said, since NICAD has been tested on large systems, we are confident that it is a suitable engine for comparing code using TXL. Also, there is nothing that prevents us from using other code comparisons engines, if need be.

In conclusion, internal and external validity have both been minimized by choosing a set of three different systems, using input data that can be found in any programming languages and version systems (commit and changesets).

VI. Conclusion

We presented PRECINCT (PREventing Clones INsertion at Commit Time), an incremental approach for preventing clone insertion at commit time that combines efficient block extraction and clone detection and integrate itself seamlessly into the day-to-day workflow of developers. PRECINCT takes advantage of TXL and NICAD to create a clone detection tool approach that runs automatically before each commit in 1.2 seconds with a 97.7% precision and a 100% recall (when using NICAD results as a baseline).

Our approach also assesses two major factors that contribute to the slow adoption of clone detection tools: a large number of data output by clone detection methods, and smooth integration with the task flow of the developers. PRECINCT can reduce the number of lines output by a traditional clone detection tool such as NICAD by 83.4% while keeping all the necessary information that allow developers to decide whether the detected clone is, in fact, a clone. Also, our approach is seamlessly integrated with the developers' workflow using pre-commit hooks, which are part any version control systems.

To build on this work, we need to experiment with additional (and larger) systems with the dual aim to (a) improve and fine-tune the approach, and (b) assess the scalability of our approach when applied to even larger (and proprietary) systems. Also, we want to improve PRECINCT to support Type 4 clones.

References

[1] B. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of 2nd working conference on reverse engineering*, 1995, pp. 86–95.

[2] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in *Proceedings. ieee international conference on software maintenance*, 1999, pp. 109–118.

[3] M. Kim, V. Sazawal, and D. Notkin, "An empirical study of code clone genealogies," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 187–196, Sep. 2005.

[4] C. Kapser and M. Godfrey, "Cloning Considered Harmful Considered Harmful," in *13th working conference on reverse engineering*, 2006, pp. 19–28.

[5] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *IEEE 31st international conference on software engineering*, 2009, pp. 485–495.

[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.

[7] M. F. Zibran and C. K. Roy, "IDE-based real-time focused search for near-miss clones," in *Proceedings of the 27th annual acm symposium on applied computing*, 2012, pp. 1235–1242.

[8] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: scaling code clone detection to big-code," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 1157–1168.

[9] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, "Industrial application of clone change management system," in *Proceedings of the 6th international workshop on software clones*, 2012, pp. 67–71.

[10] G. Zhang, X. Peng, Z. Xing, Shihai Jiang, Hai Wang, and W. Zhao, "Towards contextual and on-demand code clone management by continuous monitoring," in *28th ieee/acm international conference on automated software engineering*, 2013, pp. 497–507.

[11] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[12] T. J. Robertson, S. Prabhakararao, M. Burnett, C. Cook, J. R. Ruthruff, L. Beckwith, and A. Phalgune, "Impact of interruption style on end-user debugging," in *Proceedings of the sigchi conference on human factors in computing systems*, 2004, pp. 287–294.

[13] T. J. Robertson, J. Lawrance, and M. Burnett, "Impact of high-intensity negotiated-style interruptions on end-user debugging," *Journal of Visual Languages and Computing*, vol. 17, no. 2, pp. 187–202, 2006.

[14] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, and C. Cook, "Tinkering and gender in end-user programmers' debugging," in *Proceedings of the sigchi conference on human factors in computing systems - chi '06*, 2006, pp. 231–240.

[15] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr., "Does bug prediction support human developers? findings from a google case study," in *International conference on software engineering*, 2013, pp. 372–381.

[16] S. L. Foss and G. C. Murphy, "Do developers respond to code stability warnings?" in *Proceedings of the 25th annual international conference on computer science and software engineering*, 2015, pp. 162–170.

[17] L. Layman, L. Williams, and R. S. Amant, "Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools," in *First international symposium on empirical software engineering and measurement (esem 2007)*, 2007, pp. 176–185.

[18] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th acm sigplan-sigsoft workshop on program analysis for software tools and engineering - paste '07*, 2007, pp. 1–8.

[19] N. Ayewah and W. Pugh, "A report on a survey and study of static analysis users," in *Proceedings of the 2008 workshop on defects in large software systems - defects '08*, 2008, pp. 1–5.

[20] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *35th international conference on software engineering (icse)*, 2013, pp. 672–681.

[21] D. A. Norman, *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books, 2013, p. 347.

[22] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, Dec. 2004.

[23] N. Lopez and A. van der Hoek, "The code orb: supporting contextualized coding via at-a-glance views," in *Proceeding of the 33rd international conference on software engineering*, 2011, pp. 824–827.

[24] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, Sep. 2007.

[25] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," in *Proceedings of the fourth working conference on reverse engineering*, 1997, pp. 44–54.

[26] C. Kapser and M. Godfrey, "Aiding comprehension of cloning through categorization," in *Proceedings. 7th international workshop on principles of software evolution, 2004.*, 2004, pp. 85–94.

[27] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the ieee international conference on software maintenance.*, 1998, pp. 368–377.

[28] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in

Fourth ieee international workshop on source code analysis and manipulation, 2004, pp. 128–135.

[29] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *29th international conference on software engineering*, 2007, pp. 96–105.

[30] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Lague, "Extending software quality assessment techniques to Java systems," in *Proceedings seventh international workshop on program comprehension*, 1999, pp. 49–56.

[31] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Partial redesign of Java software systems based on clone analysis," in *Sixth working conference on reverse engineering*, 1999, pp. 326–336.

[32] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the 1993 conference of the centre for advanced studies on collaborative research: Software engineering-volume 1*, 1993, pp. 171–183.

[33] J. H. Johnson, "Visualizing textual redundancy in legacy source," in *Proceedings of the 1994 conference of the centre for advanced studies on collaborative research*, 1994, p. 32.

[34] U. Manber, "Finding similar files in a large file system," in *Usenix winter*, 1994, pp. 1–10.

[35] 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).*, 1999, pp. 109–118.

[36] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Eighth ieee working conference on reverse engineering*, 2001, pp. 301–309.

[37] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 13th international conference on software engineering - icse '08*, 2008, pp. 321–330.

[38] 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.

[39] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *IEEE international conference on software maintenance*, 2010, pp. 1–9.

[40] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto, "Incremental Code Clone Detection: A PDG-based Approach," in *18th working conference on reverse engineering*, 2011, pp. 3–12.

[41] N. Göde and R. Koschke, "Incremental Clone Detection," in *13th european conference on software maintenance and reengineering*, 2009, pp. 219–228.

[42] M. F. Zibran and C. K. Roy, "Towards flexible code clone detection, management, and refactoring in IDE," in *Proceeding of the 5th international workshop on software clones*, 2011, pp. 75–76.

[43] R. Tairas, J. Gray, and I. Baxter, "Visualization of clone detection results," in *Proceedings of the 2006 oopsla workshop on eclipse technology eXchange*, 2006, pp. 50–54.

[44] T. D. Latoza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceeding of the 28th international conference on software engineering - icse '06*, 2006, pp. 492–501.

[45] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone Management for Evolving Software," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1008–1026, Sep. 2012.

[46] S. Niko, L. Mircea, and R. Romain, "On how often code is cloned across repositories," in *Proceedings of the 34th international conference on software engineering*, 2012, pp. 1289–1292.

[47] V. Saini, H. Sajnani, J. Kim, and C. Lopes, "SourcererCC and SourcererCC-I: : Tools to Detect Clones in Batch mode and During Software Development," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 597–600.

[48] W. Toomey, "Ctcompare: Code clone detection using hashed token sequences," in *6th international workshop on software clones (iwsc)*, 2012, pp. 92–93.

[49] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[50] M. F. Zibran, "Towards Implementation of an Integrated Clone Management Infrastructure," in *23rd international conference on software analysis, evolution, and reengineering*, 2016, pp. 60–61.

[51] 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*, 2006, pp. 1–11.

[52] J. R. Cordy and C. K. Roy, "The NiCad Clone Detector," in *2011 ieee 19th international conference on program comprehension*, 2011, pp. 219–220.

[53] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider, "Agile Parsing in TXL," in *Proceedings of ieee international conference on automated software engineering*, 2003, vol. 10, pp. 311–336.

[54] A. Marcus and J. Maletic, "Identification of high-level concept clones in source code," in *Proceedings 16th annual international conference on automated software engineering*, 2001, pp. 107–114.

[55] 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*, 2005, pp. 63–71.

[56] CHANCHAL K. ROY, "Detection and Analysis of Near-Miss Software Clones," PhD thesis, Queen's University, 2009.

[57] 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.

[58] M. Nayrolles, A. Hamou-Lhadj, T. Sofiene, and A. Larsson, "JCHARMING : A Bug Reproduction Approach Using Crash Traces and Directed Model Checking," in *Proceedings of the 22nd international conference on software analysis, evolution, and reengineering*, 2015, pp. 101–110.

[59] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "A bug reproduction approach based on directed model checking and crash traces," *Journal of Software: Evolution and Process*, 2016.