Enhancing Source-Based Clone Detection Using Intermediate Representation

Gehan M. K. Selim
School of Computing, Queens University
Kingston, Ontario, Canada, K7L3N6
gehan@cs.queensu.ca

King Chun Foo, Ying Zou
Department of Electrical and Computer Engineering,
Queens University,
Kingston, Ontario, Canada, K7L3N6
3kcdf@queensu.ca, ying.zou@queensu.ca

Abstract— **Detecting software clones in large scale projects helps improve the maintainability of large code bases. The source code representation (e.g., Java or C files) of a software system has traditionally been used for clone detection. In this paper, we propose a technique that transforms the source code to an intermediate representation, and then reuses established source-based clone detection techniques to detect clones in the intermediate representation. The clones are mapped back to the source code and are used to augment the results reported by source-based clone detection. We demonstrate the performance of our new technique using systems from the Bellon clone evaluation benchmark. The result shows that our technique can detect Type 3 clones. Our technique has higher recall with minimal drop in precision using Bellon corpus. By examining the complete clone groups, our technique has higher precision than the standalone string based and token based clone detectors.**

Keywords- software clones, intermediate representation, token based clone detection tools, string based clone detection tools.

I. INTRODUCTION

Code clones are sets of syntactically or semantically similar code segments residing at different locations in the source code. Code clones are generally considered to be a contributing factor that leads to software maintenance problems [2]. The primary concern is that programmers often reuse existing solutions by copying and pasting code lines with little or no modification. This reuse scheme is error prone. For example, 34 out of 35 errors in a single file under the Linux drivers/i2o directory were caused by copying and pasting activities [7]. To ensure a thorough correction of errors upon discovery, programmers must apply their fixes to all duplicated code segments.

The level of code similarity ranges from identical code segments (Type 1), code segments with slight renaming (Type 2), and slightly renamed segments with additional or removed intermediate lines (Type 3) [16] [20]. The detection of these types of clones varies based on the techniques used for detecting them, with more complex techniques being able to detect Type 3 clones but require additional processing power. For example, string based techniques [11] [28] can recognize Type 1 clones. Token based techniques can identify Type 1 and 2 clones. Both string based and token based techniques can perform very fast detection. However, most string based or token based techniques fail to directly detect Type 3 clones without any post-processing steps to combine Type 1 and Type 2 clones to form Type 3 clones [21][23]. Generally, Type 3 clones can be detected with Abstract Syntax Tree (AST) based [9] and Program Dependency Graph (PDG) based techniques [14] [18]. Such techniques may have higher computation requirements than the string based and token based techniques. An ideal detection technique would perform as fast as string based or token based techniques while being able to detect Type 3 clones. Koschke et al. [15] propose a clone detection technique that uses abstract suffix trees of the source code to find Type 3 clones in linear time and space.

Instead of detecting clones on the source code represented in AST or PDG, we believe that we can uncover Type 3 clones by analyzing the intermediate representation (e.g., three address instructions) using string based or token based techniques. These clones could be then added to the Type 1 and Type 2 clones which are much easier to detect. Our intuition is derived from the fact that compilers tend to simplify and normalize many high level constructs into the same low level constructs. For example, we would expect that 'for' and 'while' loops to be mapped to the same intermediate representation (e.g., a goto statement). We would expect a compiler to perform consistent statement reordering when emitting the intermediate representation. Such re-ordering would reduce high variation in code segments.

The main contributions of our work are as follows:

1. We propose a hybrid clone detection technique which can detect Type 3 clones by combining source code and intermediate code detected clones. The intermediate code clones are mapped back to the source code.
2. We demonstrate our technique on the Java programming language. We use the Soot framework [24] to perform the intermediate code analysis. We use existing string based and token based clone detectors (i.e., CCFinder [25] and Simian [28]) to perform the clone detection.
3. We quantitatively and qualitatively study the performance of our technique using the Bellon clone evaluation benchmark [6]. Our study shows that our technique has higher recall for three projects compared to other source-based techniques using Bellon corpus. Our technique has higher precision than the standalone string or token based clone detectors by checking all clone groups.

The rest of this paper is organized as follows. Section II gives an overview of the Soot framework and its intermediate representation of a Java application. The intermediate representation is called Jimple. Our proposed hybrid technique analyzes the Jimple code for clones using traditional string based and token based clone detectors.

Section III discusses our proposed clone detection technique. Section IV describes the results of our case study, and explores threats to validity. Section V covers related research work in the field of clone detection. Finally, Section VI concludes the paper and explores the future work.

II. Overview of Jimple Representation

We demonstrate our clone detection technique on the Java programming language. To study the intermediate representation of a Java application, we use the Soot framework [28]. The Soot framework uses the Jimple language to produce a unified format for representing Java source code. Jimple is an alternative representation to the stack based Java binary code. Jimple serves as an abstraction layer on top of the binary code. Jimple dramatically reduces the number of operations needed to represent the Java binary code. We believe that this limited number of operations leads to a reduced dissimilarity in cloned code segments and helps us locate clone instances with complex variations. Similar to binary code, Jimple uses unconditional transfer of control to represent all Java control structure (e.g., if-else, for-loop). In other words, all control structures are translated to a combination of 'goto' statements, program labels, and 'nop' statements. The Soot framework generates line tags that preserve the mapping of Jimple lines back to their corresponding Java lines. Details about the Soot framework and its representations can be found in [24].

Java	Jimple

1 static int factorial (int x){
2 int result = 1;
3 int i = 2;
4 while (i <= x) {
5 result *= i;
6 i++;
7 }
8 return result;
9 }
	1 static int factorial (int) {
	2 int x, result, i, temp$0, temp$1, temp$2, temp$3;
	3 x := @parameter0: int; /*1*/
	4 result = 1; /*2*/
	5 i = 2; /*3*/
	6
	7 label0:
	8 nop; /*3*/
	9 if i <= x goto label1; /*4*/
	10 goto label2;/*4*/
	11
	12 label1:
	13 nop; /*4*/
	14 temp$0 = result; /*4*/
	15 temp$1 = temp$0 * i; /*4*/
	16 result = temp$1; /*5*/
	17 temp$2 = i; /*5*/
	18 temp$3 = temp$2 + 1; /*6*/
	19 i = temp$3; /*6*/
	20 goto label0; /*4*/
	21
	22 label2:
	23 nop; /*4*/
	24 return result; /*8*/
	25 }

Figure 1 . A sample Java code and its corresponding Jimple code

Figure 1 shows a sample Java code and its corresponding Jimple code. A simple Java method that calculates the factorial of a given integer is listed on the left; the method is translated into Jimple on the right. As shown in Figure 1, the Java line tags generated by Soot are captured as comments and listed beside each Jimple code line. These generated tags help automatically map the Jimple code lines to the originating Java code lines.

As shown in line 2, Jimple code declares all variables, such as local variables (e.g., x and result) and temporary variables (e.g., temp$0 and temp$1). In lines 3 to 5, the Jimple code initializes the function argument (i.e., x:=@parameter0:int) and variables. Lines 7 to 10 test the execution condition for the while statement as mapped to the Java code. Lines 12 to 20 correspond to the body of the while-loop (i.e., lines 4 to 6 in Java code). To complete the Java while-loop, a 'goto' statement in line 20 of Jimple code re-directs the execution to line 7 to test the conditions for the next iteration. Line 24 corresponds to the Java code that returns the result following the execution of the while statement (i.e., line 8 in Java).

III. Proposed Technique

Figure 2 gives an overview of our hybrid clone detection technique. The technique produces a list of cloned code segments by combining source-code clones and intermediate-code clones that are mapped to the corresponding source code.

Our work strives for detecting more complex clones (e.g., Type 3 and gapped clones) using traditional, high performance source-code clone detection tools (i.e., string and token based clone detectors). To use such tools to detect clones in the intermediate code, the Soot framework automatically generates Jimple code corresponding to the source code. The Soot framework can generate the Jimple code from the binary (i.e., class) or the source file. We use the source file to avoid transforming the external library code and binary code annotations from binary code to the Jimple code. The Jimple files are named with the extension of ".Jimple" by the Soot framework. While there exist no tools to detect clones in Jimple code, Jimple has a very similar structure to Java. We force the source-based clone detectors to recognize the Jimple code as Java code. For example, we force Simian (i.e., a string based clone detector) to recognize the Jimple code as Java code by configuring the language option to "Java". To use CCFinder (i.e., a token based clone detector), we rename each Jimple file with the extension of ".Java". We refer to this step as the 'Java-fication' step as shown in Figure 2. The content of the Jimple code remains intact during the Java-fiction step. Traditional source-based clone detection tools are then executed on the Java-like Jimple code. The Soot framework automatically embeds the corresponding line number of the Java code for each line of Jimple code. The detected clones in Jimple are automatically mapped back to the corresponding source code. The mapped-Jimple clones and the source-based clones are then merged to produce the final results of our technique.

We discuss below our approach to merge the mapped-Jimple clones and the source-based clones. We also discuss the impact of our technique on the detection of Type 3 clones.

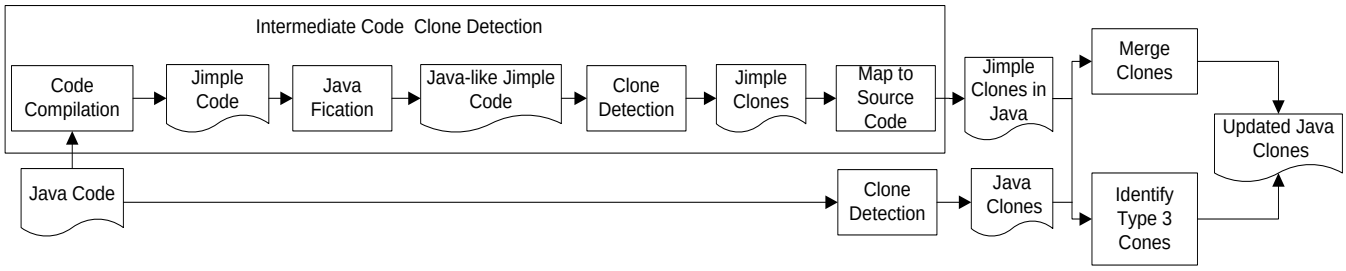

Figure 2. An overview of our proposed technique

A. Merging Jimple Clones and Java Source Clones

We consider three cases for merging Java clone groups and Jimple clone groups: complete merge, partial merge and no merge. The merging process is applied directly to the outputs of clone detectors.

Complete merging of clone groups: For a Jimple clone group to be merged to a Java clone group, there must be a one-to-one mapping between the clone instances from the Jimple and Java clone groups, such that the Java code lines mapped from a Jimple clone instance are a subset of the Java code lines of a Java clone instance. More specifically, a Jimple clone instance can be mapped to a Java clone instance when a Jimple clone instance contains consecutive statements without go to statements. If a Jimple clone instance contains code that corresponds to a Java control statement (e.g., if-else and for statement), the Jimple clone instance would cover a continuous sequence of Java statements before the subsequent control statement. For example in Figure 1, a Jimple clone instance which covers lines 4 to 20 can be mapped to a Java clone instance that covers from lines 2 to 6. In case of complete merging, the Java and Jimple clone instances reveal the same information. Hence, we discard the Jimple clone instances.

Partial Merging of Clone Groups: Instead of encapsulating another clone instance completely, Java and Jimple clone instances can overlap. To group these two types of clone instances, the two overlapping clone instances must cover an overlapping threshold (e.g., 70%) of common lines of Java code in each group of clone instances. Hence, we are confident that the majority of the Java code is common in the original Java clone instances and the segments mapped from Jimple clone instances. We evaluate the overlap between the Java and Jimple clone instances using equation (1). Different overlapping threshold values are experimented in this study.

$$Ovelap = \frac{Matching\ Consecutive\ \ Lines\ Between\ 2\ Clone\ Instances}{Total\ Lines\ Of\ \ Code\ Of\ \ Both\ Instances} \quad (1)$$

TABLE I.
EXAMPLES OF PARTIAL MAPPING BETWEEN JIMPLE AND JAVA CODE

Features	Java Code	Jimple Code
Expression	i++; i = i + 1; ++i;	temp = i + 1; i=temp;
Method invocation	s = str.toString().trim()	temp = str.toString(); s = temp.trim();
Control structure	do-while, for-loop, while-loop, for-each-loop, if-statement, condition	goto label

In case of partial merging, we place the Java and Jimple clone instances in the same clone group. In our study, we found that partial mappings between the Jimple and Java clone instances are caused by different coding styles: expression shorthand, nested method invocation and semantically equivalent control statements. Table I shows some examples of coding styles adopted by developers according to personal preferences. As shown in Table I, Java supports three different forms for incrementing an integer. Similar code segments with minor differences in coding styles are quantified into Type 3 clones. It is challenging for string based and token based clone detectors to identify Type 3 clones. Jimple unifies Java source code by expanding expression shorthand and nested method calls into a unified form of three-address instructions. For example shown in Table I, the three forms of incrementing an integer in Java are converted to a unified form in Jimple. Nested method invocation can be used in Java code shown in the second row of Table I. The Jimple code expands the method invocations into multiple individual method call statements.

Unmerged Clone Groups: If the overlapped lines of Java code between the Jimple clone instances and Java clone instances are less than the threshold value (e.g., 70%), the Java and Jimple clone instances remain unmapped. The unmapped Jimple clone instances provide the opportunity to identify Type 3 clones in Java code.

B. Detecting Type 3 Clones

Identifying Gapped Type 3 Clones: A Jimple clone instance may contain Java code fragment that overlaps with two Java clone instances from different clone groups. But the overlapping percentages in both cases are less than the thresholds. Essentially, the Jimple clone instance when converted to Java connects the Java clone instances of different groups to form a larger sized clone instance. This is particularly important when the smaller sized clone instances have dependencies. By connecting these smaller sized clone instances, developers can get a more complete picture of the computation in the clone instances which in turn help with the maintenance effort.

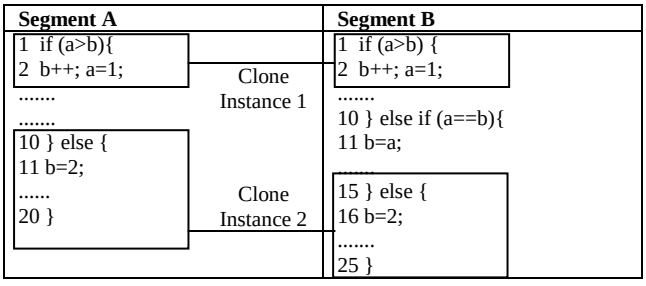

Figure 3 An Example of a Type 3 gapped clone instance

It is possible that Java clone instances from two different clone groups always appear in close proximity to each other. These Java clone instances, however, are separated by non-cloned Java code segments, such as additional conditional

branches for error checking, and extra computation. For example shown in Figure 3, both Java clone instances in segment A are identical to the Java clone instances in segment B except segment B contains an extra else-if branch between Lines 10 and 14. Such clone instances of two clone groups are separated. The clone detectors would detect each similar portion as a separate clone instance given that the portion is of sufficient length. We refer to such instances as "gapped clones" where the instances from different clone groups are separated by non-cloned code segments [20]. The gapped clones are considered as Type 3 clones and may not be detected by existing token based or string based detectors dependent on the size of the gap. However, in some cases, the Jimple code corresponding to the separated Java clone instances is reordered and placed next to each other by Soot. The non-cloned segments are moved to the end of the two clone instances. Therefore, the corresponding Jimple clone instances are contiguous code. We infer gapped Java clones by mapping Jimple clones to two separated Java clone instances from different clone groups to form a larger Java clone instance. We evaluate the similarity of the new Java clone instances using equation (2) to control the number of non-cloned lines between both Java clone instances. Different similarity threshold values are experimented in the case study.

$$Similarity = \frac{Matching\ Cloned\ Lines\ Between\ Two\ Clone\ Instances}{Total\ Lines\ Of\ Code\ Of\ Both\ Instances} \quad (2)$$

Identifying Non-Gapped Type 3 Clones: Unmapped Jimple clone instances provide an opportunity for discovering new Java clone instances that are missed by a source-based clone detector. For each unmapped Jimple clone instance, if the length of the mapped Java code segments is sufficiently large; above the threshold set by a clone detector, a new Java clone instance is created from the mapping. A new Java clone group would contain all instances of the original Jimple clone group in their mapped Java source code.

Similar to the cases where a Jimple clone instance is partially mapped to a Java clone instance, clone detectors fail to recognize the new Java clone instances mapped from the Jimple code due to the choice of control statements and coding styles for writing the same computation in the code fragments. For example, switch statements in one clone instance can be represented using a series of if-else-if conditions in another clone instance; For-loop statements in one clone instance can be expressed using the equivalent while statement. Jimple translates the different variations of a control statement into a common form using goto statements and labels. As a result, a string based or token based clone detector can capture similar Jimple segments that are written in different coding styles when translated back to Java. This enables the string based or token based clone detector to detect Type 3 clones which are clones produced due to minor differences in coding styles [20].

IV. CASE STUDY

We perform a case study to evaluate our new technique. The goals of the case study are to:

- Qualitatively compare the performance (i.e., precision and recall) of our technique with source-based clone detection techniques. We expect that our technique would increase the recall. However we must ensure that the improvement in recall is not associated with a drastic drop in precision.
- Examine the types of clone instances reported by our technique.

In the following subsections, we discuss the experiment setup and the evaluation results.

A. Experiment Setup

Bellon Benchmark: The Bellon setup is an experimental setup suggested by Bellon et al. [6] as a means of standardizing the evaluation of clone detectors. Different clone detectors are evaluated using eight Java and C systems. Such clone detectors have different capability to detect Type 1, Type 2 and Type 3 clones. Six leading clone-detection researchers test their clone detectors on the subject systems. Due to the time needed to manually verify the results, only 2% of the clone groups reported by the six clone detectors are randomly selected and evaluated by Bellon. Evaluation is done incrementally where 1% of the reported clone groups are 'oracled' for evaluation, then another 1% is tested. Clone groups validated by Bellon as correctly identified clone groups are used to build a reference corpus. Each reported clone group is referred to as a 'candidate', and each correctly identified clone group is referred to as a 'reference'. Further details of the original setup are provided in [6].

Subject Systems: We select three of the four Java systems used in the Bellon benchmark. Table II summarizes the characteristics of these three systems.

- Eteria IRC Client (EIRC) is an Internet relay chat client program.
- Secure Practical Universal Lecture Evaluator (Spule) automates the evaluation of lecture polls.
- Netbeans-Javadoc is a documentation tool provided by Netbeans for viewing and generating documentation of Java projects.

After compiling using the Soot framework, the Jimple code for the systems is automatically generated. As shown in Table II, the number of the Jimple code files is more than their corresponding Java files since the Soot framework generates separate Jimple files for inner classes. We limit our study to these three systems since the Soot framework was not able to process the remaining system due to the lengthy folder structure of j2sdk1.4.0-Javax-swing.

TABLE II
OVERVIEW OF SUBJECT SYSTEMS

System	KLOC in Java	# Java Files	# Jimple Files
EIRC	11	65	79
Spule	13	58	150
Netbeans-Javadoc	19	101	207

Clone Detectors: We experimented with Simian [28] and CCFinder [12] [25] as our choice of source-based clone detectors used by our technique. CCFinder is a token based clone detector which detects Type 1 and Type 2 clones.

Simian uses a string based technique which normalizes program identifiers before clone detection. Simian can analyze code in any format since it is string based. However, CCFinder does not support clone detection on Jimple format. Hence, it is the rationale for our Jimple 'Java-fication' step in our technique shown in Figure 2.

We used the default thresholds set by the clone detectors to categorize similar code blocks as clones in both Java code and Jimple code. Specifically, Simian uses a threshold of 6 lines to treat similar code blocks as clones, and CCFinder has a threshold of 50 tokens to categorize code blocks as clones. We configure the CCFinder tool to produce non-overlapping clones. This option ensures that CCFinder does not report clone groups that are due to the shifting of the same code segments by a few lines.

Selection of Overlapping Thresholds: To find the number of clone groups detected in common from Jimple and Java, we counted the number of 'completely' or 'partially' merged clone groups between the two code representations (described in Section III.A). For detecting partially mapped clone groups, we experimented with different overlapping thresholds between the clones mapped from Jimple and the clones from Java. Table III shows the thresholds and the corresponding statistics. We tried 70%, 80% and 90% to determine when to perform partial merging of clone groups. For 80% overlap, we got 3 to 5 fewer clone groups being merged than those merged at 70%. For 90% overlap, we got 4 to 7 fewer clone groups being merged than those merged at 70%. As the overlapping threshold increases, the clone groups to be merged decreases since we request a higher overlap. However, the decrease in the number of clone groups is not significant. We choose 70% as the overlapping threshold to generate our results for the Bellon performance evaluation.

TABLE III
PARTIAL MERGING RESULTS FOR DIFFERENT OVERLAPPING THRESHOLDS

System	# Common Clone Groups from Jimple and Java Using Simian			# Common Clone Groups from Jimple and Java Using CCFinder		
	Overlapping Thresholds			Overlapping Thresholds		
	70%	*80%*	*90%*	*70%*	*80%*	*90%*
EIRC	33	30	29	103	99	98
Spule	79	75	75	200	197	196
Javadoc	67	62	60	259	255	252

Selection of Similarity Thresholds: To find the number of gapped clones detected from the Jimple representation, we try several similarity thresholds (described in Section III.B). Table IV shows the number of gapped clones obtained for the different thresholds. We tried 90%, 80% and 70% as similarity thresholds to determine when to combine smaller Java clones into a bigger gapped Java clone. For 80% similarity, we got 3 to 4 more gapped Java clone instances than those obtained when using 90%. For 70% similarity, we got 4 to 8 more gapped Java clone instances than those obtained when using 90% similarity. As the similarity threshold decreases, the gapped Java clone instances increases since we request a less degree of similarity and hence less restricting conditions. The increase in the number of gapped Java clone instances is not significant. We choose 90% as the similarity threshold to generate our results for the Bellon performance evaluation.

TABLE IV
GAPPED CLONE COUNT FOR DIFFERENT SIMILARITY THRESHOLDS

System	# Gapped Clone Instances from Jimple using Simian			# Gapped Clone Instances from Jimple using CCFinder		
	Similarity Thresholds			Similarity Thresholds		
	90%	*80%*	*70%*	*90%*	*80%*	*70%*
EIRC	201	204	209	300	304	307
Spule	166	170	170	500	504	508
Javadoc	522	525	530	570	573	577

B. Performance Evaluation

Evaluation Metrics: We compare the performance of our technique to Simian alone and to CCFinder alone. We also compare our performance to the performance of CloneDR [5], an AST based clone detector, and CLAN [19], a metric based clone detector. CloneDR and CLAN can detect Type 3 clones. The results generated from both clone detectors are available in the Bellon benchmark.

We measure the performance using recall and precision which are calculated as shown in equations (3) and (4). Recall is the number of reference clone groups detected by our technique relative to all of the reference clone groups available in the benchmark. Precision is the number of reference clone groups detected by our technique relative to all the candidate clone groups detected by our technique.

Bellon et al. note that the precision metric would result in the minimum possible precision of the clone detector under assessment, since the number of candidates produced by a clone detector can be very large relative to the number of the references provided in the benchmark. Hence Bellon et al. suggest measuring the ratio of rejected candidates; shown in equation (5). For the rejected ratio, we find how many candidates are seen by Bellon; whether they were rejected or accepted as clones, and from those how many are 'oracled'. The lower the rejected ratio, the better is the performance since rejected is the inverse of the theoretical precision if Bellon had manually examined or oracled all of the generated clone groups, not just 2% of them.

$$Recall = \frac{|Detected \quad References|}{|References|} \tag{3}$$

$$Precision = \frac{|Detected \quad References|}{|Candidates|} \tag{4}$$

$$Rejected = \frac{|Rejected \quad Candidates|}{|Oracled \quad Candidates|} \tag{5}$$

We also carry out a manual evaluation of all the clone groups generated by our technique and those generated by Simian and CCFinder. In the manual evaluation, we verify the validity of the clone groups and classify the types of clone groups. Manual evaluation was necessary since the

rejected and precision values available in the Bellon corpus are derived from the candidates examined by Bellon and hence can be considered as an incomplete evaluation of our technique. It took the authors over 8 days to perform the manual evaluation. The precision of manually verifying all clone groups is defined in equation (6):

$$Precision = \frac{\left| True\ Clone\ Groups \right|}{\left| Candidate\ Clone\ Groups \right|} \quad (6)$$

Results: The normalized Java code of the test systems [26] is given as input to our technique which uses Simian and CCFinder alternatively as the intermediate clone detector. The systems are also passed to Simian and CCFinder as standalone clone detectors. For each of these cases, recall, precision and rejected values are calculated according to equations (3), (4) and (5) with reference to the benchmark. The results obtained are shown in Table V, Table VI and Table VII. We also compare the precision, recall and rejected values of our technique to those of other clone detectors that detect Type 3 clones in Table VIII.

Results of Recall (Table V, Table VIII): Our technique using Simian or CCFinder gives higher recall than that of Simian or CCFinder when used as a standalone clone detector. The high recall is mainly because our technique detects additional correctly identified clone groups present in the corpus when Jimple code is used for clone detection. The results also show that our technique using CCFinder improves more on the performance of CCFinder than our technique using Simian improves over Simian. This can be seen from the improved recall which reflects additional number of detected reference clones of our technique. Thus we can conclude that our technique improves over the performance of CCFinder but does not significantly improve over the performance of Simian. From Table VIII, we can also see that in general our technique when used with Simian or CCFinder achieves much higher recall than the recall achieved by CLAN or CloneDr. Only in the case of Spule, CloneDR achieves a higher recall than our technique when used with Simian.

TABLE V
RECALL USING THE BELLON REFERENCE CORPUS

Recall	Our Technique using Simian	Simian	Our Technique using CCFinder	CCFinder
EIRC	0.68	0.6	0.8	0.35
Spule	0.66	0.66	0.94	0.7
Javadoc	0.78	0.76	0.89	0.76

TABLE VI
PRECISION USING THE BELLON REFERENCE CORPUS

Precision	Our Technique using Simian	Simian	Our Technique using CCFinder	CCFinder
EIRC	0.39	0.5	0.35	0.36
Spule	0.84	0.88	0.82	0.90
Javadoc	0.37	0.44	0.28	0.62

TABLE VII
REJECTED VALUES USING THE BELLON REFERENCE CORPUS

Rejected	Our Technique using Simian	Simian	Our Technique using CCFinder	CCFinder
EIRC	0.43	0.44	0.30	0.3
Spule	0.24	0.24	0.09	0.22
Javadoc	0.35	0.33	0.32	0.19

TABLE VIII
COMPARISON OF CLAN AND CLONEDR TO OUR TECHNIQUE

		EIRC	**Spule**	**Javadoc**
Precision	CLAN	0.33	0.37	0.16
	CloneDR	0.34	0.32	0.27
	Our Technique using Simian	0.39	0.84	0.37
	Our Technique using CCFinder	0.35	0.82	0.28
Recall	CLAN	0.18	0.59	0.24
	CloneDR	0.28	0.68	0.16
	Our Technique using Simian	0.68	0.66	0.78
	Our Technique using CCFinder	0.8	0.94	0.89
Reject	CLAN	0.13	0.2	0
	CloneDR	0.15	0.24	0
	Our Technique using Simian	0.43	0.24	0.35
	Our Technique using CCFinder	0.30	0.09	0.32

Results of Precision (Table VI, Table VIII): The precision of our technique is lower than that of the standalone clone detectors since additional candidate clone groups are returned. In addition, since the precision is calculated relative to the corpus, some of the additional and correctly identified clone groups might not necessarily exist in the Bellon corpus. The Bellon corpus was built by subjective evaluation of clones, and only 2% of the available clone groups were evaluated by Bellon as being correctly identified clones or not. Hence, the corpus contains a sample of the available clones in the test systems but not all of them. From Table VIII, we can see that our technique when used with Simian or CCFinder achieves much higher precision than that of CloneDR or CLAN. This can be attributed both to using Jimple for capturing additional clone groups and to configuring CCFinder to decrease the number of candidates returned by our technique when using CCFinder. We do note that our drop in precision is not a drastic drop except for Netbeans-Javadoc system.

Results of Rejected Ratio (Table VII, Table VIII): According to Table VII our technique using CCFinder outperforms CCFinder when tested on Spule. Our technique using Simian also outperforms Simian when tested on EIRC. For the remaining cases, CCFinder and Simian outperform our technique. The low performance of our techniques is due to the fact that many clone groups generated by our technique are not in the Bellon benchmark. From Table VIII, we also see that our technique usually produces higher rejected values than those of CLAN and CloneDR. However, we found that this is because when calculating the rejected ratio, the number of oracle candidates (i.e., denominator of the rejected ratio) can be very low and hence the probability of having candidates rejected from a small number is high.

TABLE IX
PRECISION BASED ON MANUAL EVALUATION

Precision	Simian	Our Technique using Simian	CCFinder	Our Technique using CCFinder
EIRC	0.74	0.79	0. 69	0.72
Spule	0.82	0.87	0.79	0.83
Javadoc	0.8	0.84	0.76	0.79

TABLE X
DISTRIBUTION OF CLONE TYPES

System	**Clone Type**	**Simian**	**Our Technique using Simian**	**CCFinder**	**Our Technique using CCFinder**
EIRC	1	44%	36.5%	28.8%	29.9%
	2	30%	32.5%	37%	35%
	3	0	10%	3.2%	7.1%
	Not clone	26%	21%	31%	28%
Spule	1	43%	46.9%	35.5%	30.2%
	2	39%	32%	40.5%	45.8%
	3	0%	8.1%	3%	7%
	Not clone	18%	13%	21%	17%
Javadoc	1	42.2%	47.9%	30.8%	32.7%
	2	37.8%	30.1%	41.2%	39.3%
	3	0%	6%	4%	7%
	Not clone	20%	16%	24%	21%

Results of Precision for all Clone Groups (Table IX, Table X): Since the Bellon corpus contains 2% of the actual clone groups available in the systems, the precision measured reflects the precision of our technique in comparison to the available fraction of clone groups in the Bellon corpus. We manually verified the generated clone groups by examining their validity and the precision of the detection from our point of view.

Table **IX** IX shows the results of the precision based on our manual evaluation. Our technique generates higher precision than that of the standalone clone detectors. In general, each Java line maps to one or more Jimple lines. In many cases the Jimple file could be twice as large in LOC as its corresponding Java file. This in turn implies the possibility of more clone groups. The number of correctly identified clone groups has also increased significantly with our technique. Table X shows that our technique detects a more Type 1, Type 2 and Type 3 clones than those detected by the standalone clone detectors. The percentage of false clones detected by our technique has decreased.

TABLE XI
COMMON AND GAPPED CLONE GROUPS USING SIMIAN

System	# Clone Groups from Java	# Clone Groups from Jimple	# New Clone Groups from Jimple	# Common Clone Groups from Jimple and Java (%)	# Gapped Clones in Jimple
EIRC	72	90	57	33 (36.7%)	201
Spule	123	160	81	79 (49.4%)	166
Javadoc	110	143	76	67 (46.9%)	522

TABLE XII
COMMON AND GAPPED CLONE GROUPS USING CCFINDER

System	# Clone Groups from Java	# Clone Groups from Jimple	# New Clone Groups from Jimple	# Common Clone Groups from Jimple and Java (%)	# Gapped Clones in Jimple
EIRC	150	197	94	103 (52.3%)	300
Spule	377	400	100	200 (50%)	500
Javadoc	302	330	71	259 (78.5%)	570

C. Discussion of Results

We structure our discussion along the following 3 questions:

Q1: Can clone groups detected from Jimple code be mapped to clone groups detected from Java?

We compare the clone groups detected from Jimple and Java code to investigate if both code representations can reveal similar cloning information. Table XI compares the clone groups detected by Simian from each code representation without merging the clone results. Similarly, Table XII shows the clone groups detected by CCFinder from each code representation without merging. The second column (e.g., # Clone groups from Java) in Table XI and Table XII lists the number of clone groups detected by the corresponding clone detector from Java code only. The third column (e.g., # Clone Groups from Jimple) shows the number of clone groups detected by the corresponding clone detector from the Jimple code only. The fourth column (e.g., # New Clone Groups from Jimple) shows the number of new clone groups detected by a clone detector from Jimple code, but not from the Java code. The fifth column (e.g., # Common Clone Groups from Jimple and Java(%)) lists the number and the percentage of Jimple clone groups detected in common with Java clone groups.

From Table XI and Table XII, a considerable number of clone groups are detected in common by Simian when used with Jimple and Java code. The percentages of common clone groups are 36.7%, 49.4% and 46.9% respectively for EIRC, Spule and Netbeans-Javadoc. Similarly, we examine the clone groups detected by CCFinder on both representations. Higher percentages of common clone groups are obtained: the: 52.3% for EIRC, 75% for Spule and 78.5% for Netbeans-Javadoc.

We check the clone instances detected in Java code but not in the corresponding Jimple code. We found that the Java clone instances are identical in the Java code but the corresponding Jimple segments use different programming keywords to distinguish various types of methods being invoked. For example, Jimple uses the keyword "interfaceinvoke" to represent calls to methods declared in interfaces; "virtualinvoke" to represent dynamically dispatched calls; and "staticinvoke" to represent calls to static methods. As a result, a clone detector on Jimple code would fail on cases where a developer uses different types of method calls on the cloned segments.

As shown in Table XI and Table XII, string or token based clone detectors on either Java code or Jimple code detects common clone groups. However, each code

representation detects additional clone groups that are not detected by the other representation. This implies that combining the output of clone detection on Java and Jimple code covers more clone groups than simply using one code representation for clone detection.

> There is a maximum of 49.4% correspondence between clone groups detected from Jimple and Java code when using Simian. Similarly, there is a maximum of 78.5% correspondence when using CCFinder.

Q2: Can clones from Jimple code be used as connector to join Java code clones that are separated by irrelevant code lines?

We investigate if our technique detects gapped clones initially not captured in Java code. The sixth column (e.g., # Gapped Clones in Jimple) in Table XI and Table XII show that our technique can use Jimple code to detect larger clone instances that are formed from smaller clone instances detected on Java code. CCFinder detects more gapped clone instances from Jimple code than Simian. Looking at the column "#Gapped Clones in Jimple" in Table XI and Table XII, additional 99, 334 and 48 gapped clone instances respectively for the three systems are detected by CCFinder. Such gapped clones are not detected using only Simian or CCFinder on the Java code. In the case of gapped clones, we express them in terms of clone instances not clone groups, because it is possible that not all instances in a clone group are gapped clones.

For example, Figure 4 shows a pair of Type 3 gapped Java clone instances and their corresponding Jimple code. The gap marked in bold in Java Clone instance 2 is a 'case' statement and two variable definitions added between the instances of the two clone groups. The corresponding Jimple code (pointed by the dashed arrow) is also marked in bold. The Jimple code corresponding to the two different Java clone instances are clone instances of the same clone group. Such a gapped clone can be detected by our technique since the gap in instance 2 is moved to the end of two clone groups after the Soot framework compiles the Java code to generate intermediate code. Therefore, the Jimple code corresponding to the two Java clone instances become identical consecutive code blocks. Such Jimple clone instances when mapped back to Java code span the additional lines. Our technique detects it as a gapped clone. Other clone detectors such as Gemini [23] can detect gapped clones by visualizing the clone groups that are separated by a few lines of code. A user can interactively choose to merge the smaller clones to form a larger clone. Without such post-processing steps, our technique automatically detects gapped clones of varying gap size. As discussed in Section III.B, a similarity threshold can be specified to control the size of the 'gap'. Our technique detects gapped clones with an average gap size of 6 lines of code. Comparing the output of our technique with the output of Simian or CCFinder, small Type 1 and Type 2 clone instances detected by Simian or CCFinder are recognized by our technique as larger, gapped clones.

> Our technique detects a large number of gapped clone instances (up to 570) in the subject systems that are not detected by Simian or CCFinder.

Q3: Can we find new clone groups that can be only detected in the Jimple code?

The major drive behind our technique is to detect Type 3 clones using string and token based clone detectors. We investigate whether the Jimple clone detection finds new clones and the distribution of the three types of clones. Table X demonstrates that our technique locates additional clone groups of different types by combining Jimple-level detection and Java-level detection. While maintaining the simplicity of string and token based clone detectors, our technique is capable of detecting Type 3 clones (i.e., Type 3 gapped clones and Type 3 non-gapped clones). The Jimple-level detection using Simian detects 10%, 8.1% and 6% more Type 3 clones in EIRC, Spule and Netbeans-Javadoc than Simian does alone. Similarly, our Jimple-level detection using CCFinder detects 3.9%, 4% and 3% more Type 3 clones in EIRC, Spule and Netbeans-Javadoc than CCFinder does.

To get an idea of the proportion of the Type 3 non-gapped clone among the Type 3 clones, we randomly select 10 clone groups classified as Type 3 from each subject system. From the selected clone groups, we count the number of gapped and non-gapped clone instances. Table XIII shows the number of non-gapped clone instances and the number of gapped clone instances. From the randomly selected sample of Type 3 clone groups, our technique using Simian or CCFinder detects more Type 3 non-gapped clones than Type 3 gapped clones. Figure 5 shows an example of a non-gapped Type 3 clone detected by our technique. The two clone instances use different control structures to achieve the same functionality. Figure 5 shows the corresponding Jimple code for each Java code. Since their Jimple code normalizes the differences in the types of loop statements used, this Type 3 non-gapped clone is detected by our technique.

> Our technique is able to detect Type 3 clones due to the use of Jimple-level clone detection.

TABLE XIII
SAMPLED NUMBER OF NON-GAPPED AND GAPPED CLONE INSTANCES

System	Our Technique using Simian			Our Technique using CCFinder		
	#Type 3 Clones	*Non-Gapped*	*Gapped*	*#Type 3 Clones*	*Non-Gapped*	*Gapped*
EIRC	35	22	13	38	22	16
Spule	40	25	15	38	29	9
Javadoc	49	41	8	47	30	17

D. Threats to Validity

Internal Threats: The Bellon reference corpus was manually built by Bellon using only 2% of all the clones suggested by the 6 clone detectors that participated in the study. Hence, evaluation using the Bellon corpus might not provide an accurate precision estimate. Moreover, the Bellon corpus breaks a few gapped clones into separate groups. Therefore, many gapped clones detected by our technique

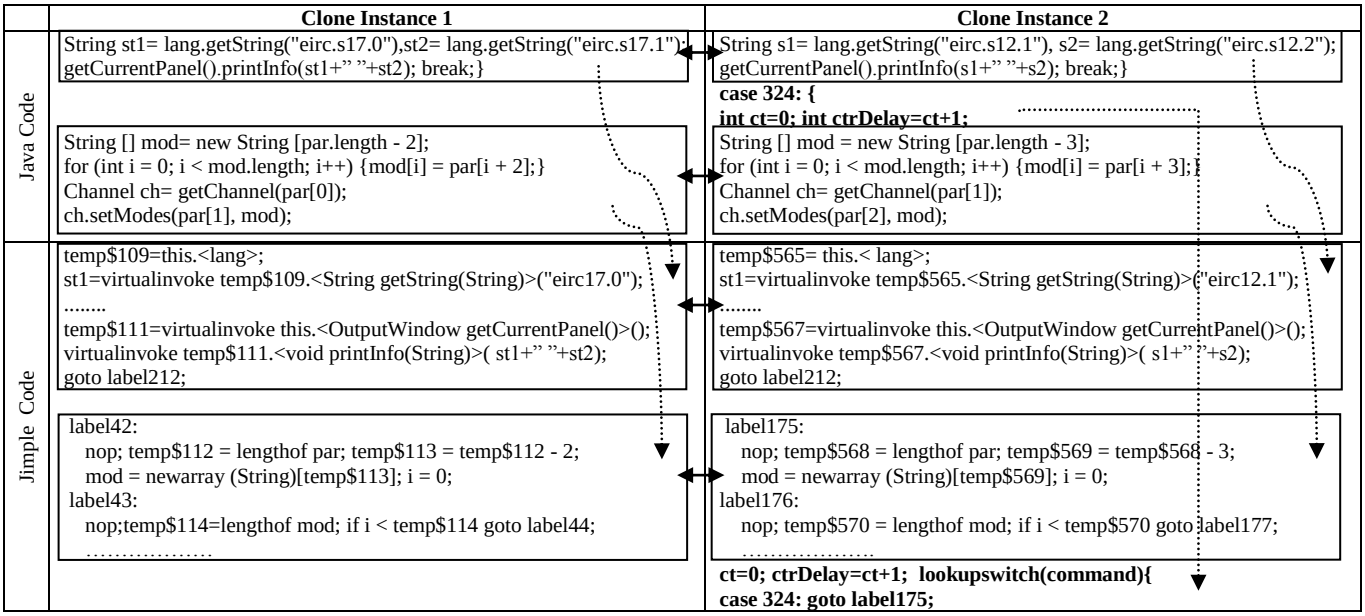

Figure 4. An example of a Type 3 gapped clone detected by our technique from EIRC

	Clone Instance 1	Clone Instance 2
Java Code	for (int j=i; j<len; j++) if (Character.isSpaceChar(s.charAt(j))) return(j);	while (s.length()>i && Character.isSpaceChar(s.charAt(i))) i++;

Jimple Code	label1:
	nop; temp$1 = virtualinvoke s.< char charAt(int)>(j);
	temp$2 = staticinvoke < boolean isSpaceChar(char)>(temp$1);
	if temp$2 == 0 goto label3; goto label2; label2: nop; return j
		label17:
		nop; temp$89 = virtualinvoke s.< char charAt(int)>(i);
		temp$90= staticinvoke<boolean isSpaceChar(char)>(temp$89);
		if temp$90 == 0 goto label19; goto label18; label18: nop;

Figure 5. An example of a Type 3 non-gapped clone from EIRC detected by our technique

are considered spurious clones when using the Bellon corpus for evaluation. The selection of a subgroup of the entire set of clone groups to be part of the Bellon corpus presents an internal threat to the validity of the study.

External Threats: Manual evaluation of the clones is time consuming. It took us over one week to manually evaluate the results generated from the three subject systems using four different settings (i.e., our technique using Simian and CCFinder and the two standalone tools). In the future, we plan to investigate the performance of our technique on larger systems. More systems in the Bellon benchmark are available for evaluation. However, we did have problems in running Soot on the eclipse plug-ins due to their hierarchical folder structure.

Manual evaluation is carried out to validate the actual precision of the standalone clone detectors and the precision of our technique. However, this evaluation was done by a single author and was not verified. Therefore, additional raters are needed to substantiate the manual evaluation of the clones. However, prior work shows that determining whether a code segment is a clone or not is rather subjective and it is hard to find consensus [13].

V. Related Work

Many studies have been recently directed to the clone detection problem. Koschke [16] provides a survey of the clone detection problem. The study covers the classification of clones, recent research work on cloning and available clone detectors. Roy and Cordy [20] provide a survey of the state of the art in clone detection techniques. In this Section, we compare our work with byte code clone detection and source code clone detection.

Byte Code Clone Detection

Saebjornsen et al. [22] propose a clone detection algorithm for disassembled, binary executables. The study uses exact clone matching, inexact clone matching and employs hash tables to detect clones. The algorithm is tested on Windows XP executables and is found to be scalable on large systems and produces few spurious clones. Baker and Manber [1, 2, 3, 4] use Siff, Dup and Diff on disassembled byte code. The study discusses the required pre-processing and adaptation of the clone detectors to handle Java byte code. Experiments showed that the approach is effective in detecting clones from byte code which can be easily mapped back to source code. Davis [27] uses Java class files to detect clones by matching segments of p-code (i.e., packed code).

The aforementioned approaches detect clones on byte code only. The results are not intended to be understood by developers. Our technique uses Jimple representation, combines the clones generated from intermediate code and Java code to improve clone groups generated from Java source code. By converting clones in intermediate code to the source code, our technique makes it easier for developers to interpret the results and improve the code.

Source Code Clone Detection

String based approaches analyze source code as a sequence of text lines. Two subsets of the input are grouped as a clone group if they share a pre-defined number of lines between them. For example, Simian [28] as a string based clone detector which can detect clones in different programming languages. Simian is sensitive to alterations in

code format. Few details of the inner workings of Simian and the used approach have been disclosed. Ducasse et al. [8] propose a string based technique that normalizes the code to minimize sensitivity of the detection process to minor changes in the code. The technique can achieve high recall and average precision.

CCFinder [12] is a token based clone detector which uses a lexical analyzer to convert the code into a token sequence on which rule based transformation is applied. CCFinder can detect clones on different programming languages. Both string based and token based clone detectors fail to directly detect Type 3 clones. Our technique uses the string based and token based clone detectors to identify Type 3 clones by applying the clone detectors on intermediate code. Other techniques have been proposed. For example, Jia et al. [10] propose the "KClone" technique which uses lexical analysis on the source code to detect Type 1 and Type 2 clones. Li et al. [17] propose a clone detection technique that detects Type 3 clones using data mining techniques. Similar to our work, Clone Miner [30], Deckard [31] and semantic clone detector [32] can detect gapped and reordered clones. Different from the aforementioned work, our technique enhances existing string and token based clone detectors to detect Type 3 clones without extra computational complexity.

VI. Conclusion

In this paper, we present a hybrid clone detection technique. The technique complements string or token-based clone detectors to detect Type 3 clones by leveraging the intermediate representation. Using systems from the Bellon benchmark and through a manual quantitative and qualitative evaluation, we show that our technique is able to detect Type 3 clones. The recall of our technique is higher than source-based clone detectors with minimal drop in the precision using Bellon corpus which has incomplete clone groups. By analyzing all the clone groups, our technique has slightly higher precision than the standalone string and token based clone detectors. In the future, we plan to apply our technique on larger systems, and evaluate the time performance and scalability of our technique.

References

[1] B. S. Baker, "A Program for Identifying Duplicated Code", Computing Science and Statistics, 1992, 24:49-57.

[2] B. S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems", WCRE 1995, pp. 86-95.

[3] B. S. Baker, "Parameterized diff," ACM-SIAM symposium on Discrete algorithms, 1999, p.854-855.

[4] B. S. Baker, U. Manber, "Deducing similarities in Java sources from bytecodes" Usenix Annual Technical Conference, 1998, pp. 179-190.

[5] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, L. Bier, "Clone Detection Using Abstract Syntax Trees", Intl. Conference on Software Maintenance 1998, Vol. 98, pp. 368-377.

[6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, "Comparison and Evaluation of Clone Detection Tools", IEEE TSE, September 2007, 33 (9), pp.577-591.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D.R. Engler, "An Empirical Study of Operating System Errors," Symp. Operating Systems Principles, pp. 73-88, 2001.

[8] S. Ducasse, O. Nierstrasz, M. Rieger, "On the Effectiveness of Clone Detection by String Matching", Journal of Software Maintenance and Evolution: Research and Practice, 2006, 18:37–58.

[9] W. S. Evans, C. W. Fraser, F. Ma, "Clone Detection via Structural Abstraction", Software Quality Journal, 2009, Vol. 17, pp. 309-330.

[10] Y. Jia, D. Binkley, M. Harman, J. Krinke, M. Matsushita, "KClone: A Proposed Approach to Fast Precise Code Clone Detection", Intl. Workshop on Software Clones, 2009

[11] J. Johnson. "Substring Matching for Clone Detection and Change Tracking," ICSM, September 1994, pp. 120-126.

[12] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: A Multilinguistic Token based Code Clone Detection System for Large Scale Source Code", TSE, July 2002, 28 (7), pp. 654-670.

[13] C. J. Kapser, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. V. Rysselberghe, P. Weißgerber, "Subjectivity in Clone Judgement: Can We Ever Agree?", Dagstuhl Seminar 06301, 2007.

[14] R. Komondoor, S. Horwitz. "Using Slicing to Identify Duplication in Source Code," Intl. Symposium on Static Analysis, July 2001, Vol. LNCS 2126, pp. 40-56.

[15] R. Koschke, R. Falke, P. Frenzel,"Clone Detection Using Abstract Syntax Suffix Trees", WCRE, 2006, pp.253-262.

[16] R. Koschke, "Survey of Research on Software Clones: Duplication, Redundancy, and Similarity in Software", Dagstuhl Seminar 06301, 2006.

[17] Z. Li, S. Lu, S. Myagmar, Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code", TSE, March 2006, Vol. 32(3): 176-192.

[18] C. Liu, C. Chen, J. Han, P. S. Yu, "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis" Conf. on Knowledge Discovery and Data Mining, 2006, pp. 872-881.

[19] E. Merlo, "Using Metrics-Based Spectral Similarity", Dagstuhl Seminar Proceedings 06301, 2007.

[20] C. K. Roy, J. R. Cordy, "A Survey on Software Clone Detection Research", Queens University, Kingston, ON, Canada, Technical Report No. 2007-541, 2007.

[21] C. K. Roy, J. R. Cordy, R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", Science of Computer Programming (2009), 74(7), Elsevier, 470-495

[22] A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan, Z. Su, "Detecting Code Clones in Binary Executables", Int. Symposium on Software Testing and Analysis, 2009.

[23] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue "On Detection of Gapped Code Clones using Gap Locations", APSEC, 2002.

[24] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, V. Sundaresan, "Soot: a Java Bytecode Optimization Framework", Center for Advanced Studies Conference 1999.

[25] CCFinder Website [Online]. Available: http://www.ccfinder.net/, last accessed in December 2009.

[26] Detection of Software Clones [Online]. Available: http://www.bauhaus-stuttgart.de/clones/, last accessed in Dec. 2009.

[27] JCD Java Clone Detector homepage, I. Davis [Online]. Available: http://www.swag.uwaterloo.ca/jcd/, last accessed in December 2009.

[28] Simian homepage [Online]. Available: http://www.redhillconsulting.com.au/products/simian/, last accessed in December 2009.

[29] Soot. Available: http://www.sable.mcgill.ca/soot/, last accessed in December 2009.

[30] H. A. Basit, S. Jarzabek, "Detecting Higher-level Similarity Patterns in Programs, ESEC and ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2005

[31] L. Jiang, G. Misherghi, Z. Su, S. Glondu, "DECKARD: Scalable and Accurate Tree-based Detection of Code Clones", ICSE 2007.

[32] M. Gabel, L. Jiang, Z. Su, "Scalable Detection of Semantic Clones", ICSE 2008.