Java Bytecode Clone Detection via Relaxation on Code Fingerprint and Semantic Web Reasoning

Iman Keivanloo[±], Chanchal K. Roy[*], Juergen Rilling[±]

[±]*Department of Computer Science*
Concordia University, Canada
{i_keiv, rilling@cse.concordia.ca}

[*]*Department of Computer Science*
University of Saskatchewan, Canada
croy@cs.usask.ca

***Abstract*—While finding clones in source code has drawn considerable attention, there has been only very little work in finding similar fragments in binary code and intermediate languages, such as Java bytecode. Some recent studies showed that it is possible to find distinct sets of clone pairs in bytecode representation of source code, which are not always detectable at source code-level. In this paper, we present a bytecode clone detection approach, called SeByte, which exploits the benefits of compilers (the bytecode representation) for detecting a specific type of semantic clones in Java bytecode. SeByte is a hybrid metric-based approach that takes advantage of both, Semantic Web technologies and Set theory. We use a two-step analysis process: (1) *Pattern matching* via Semantic Web querying and reasoning, and (2) *Content matching*, using Jaccard coefficient for set similarity measurement. Semantic Web-based pattern matching helps us to find method blocks which share similar patterns even in case of extreme dissimilarity (e.g., numerous repetitions or large gaps). Although it leads to high recall, it gives high false positive rate. We thus use the content matching (via Jaccard) to reduce false positive rate by focusing on content semantic resemblance. Our evaluation of four Java systems and five other tools shows that SeByte can detect a large number of semantic clones that are either not detected or supported by source code based clone detectors.**

***Keywords*-clone detection; Semantic Web; Java bytecode**

I. INTRODUCTION

Two code fragments that share some degree of similarity are considered typically a clone pair. The major similarity types that can be distinguished are: (1) Syntactic and (2) Semantic similarities. Syntactical similarity refers to the situation where clone pairs share similar code pattern which leads to type-1, 2, and 3 clone types [1, 2]. Semantic similarities focus on pairings' functionality [2] regardless of their code patterns. Different definitions exist on what constitutes such semantic clones in the literature, e.g. Roy et al. [1] considered them to be type-4 clones. Yoshioka et al. [3] proposed a more general definition, with *semantic clones* being code fragments which are semantically identical or syntactically (i.e. pattern) similar.

Fig. 1 shows a snapshot of two similar method blocks, each block shown in one column. Although the first 30 lines of the both methods are the same and therefore easy to detect as similar code fragments, code similarities in the remaining part of the methods are much more difficult to detect. For example, there is an offset (gap) of 45 lines within the two code fragments, before the second similarity occurs. This second similarity within the method is based on semantically similar sub-blocks which are not only once but twice repeated in the right code fragment. The last similarity can be found at the end of the two method blocks, involving two similar blocks which have some dissimilarity due to repetition and re-ordering. The arrows show the similarity relationships among the method sub-blocks.

Classical clone detection tool might be able to detect that the two methods in Fig. 1 are a type-3 clone by using extreme thresholds settings in these tools (e.g. set the gap threshold to 45 lines). However, such extreme configuration will decrease the precision drastically due to the high number of false positives it will generate. Alternatively, it is possible to call the clone pair in Fig. 1 a *semantic clone* according to Yoshioka et al.'s definition [3], since these two method blocks are both semantically and to some extent also syntactically similar.

This example shows clearly that humans can quite easily detect both syntactic and semantic similarities of these two methods. However, automated detection of semantic similarities at method level is a non-trivial task, especially for semantic clones being an undecidable problem [2].

In this paper, we present SeByte, a Java binary code clone detection approach that classifies two method blocks as clones if they are either similar in their patterns or functionalities or both. Our objective is to find semantically similar *methods* (based on Yoshioka et al.'s semantic clone definition [3]) by comparing their functionality and pattern similarities.

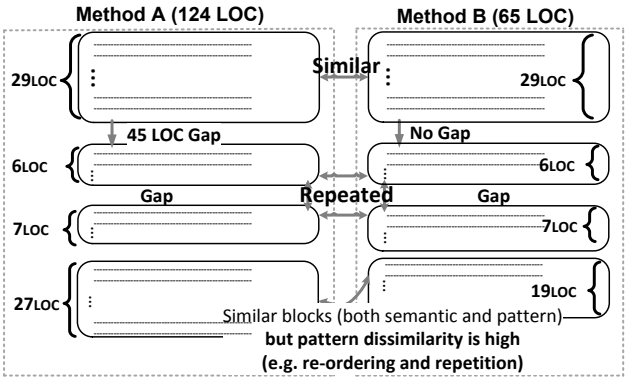

Figure 1. Two cloned method blocks that share both sytactic (i.e. pattern) and semantic (i.e. functionality) similarities.

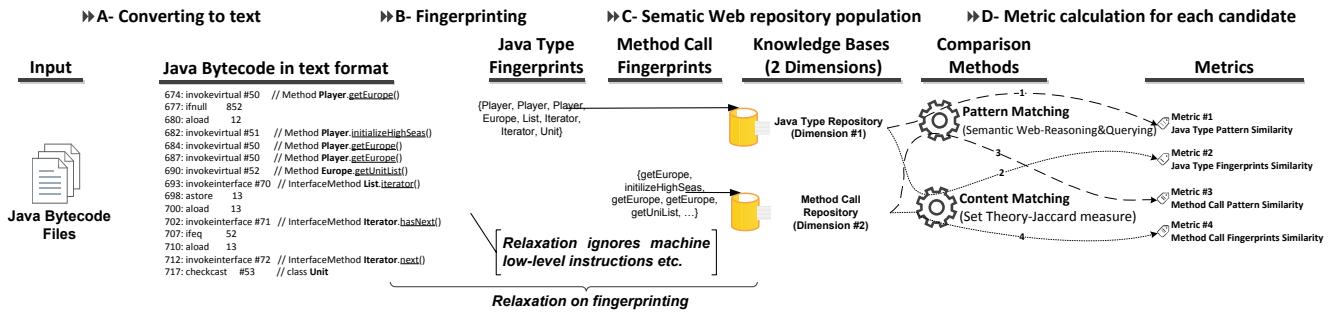

Figure 2. Our approach overview

Fig. 2 shows a schematic diagram of our approach. First it converts binary files to textual format (step A). To achieve high recall, we introduce two heuristics for Java bytecode clone detection. First *relaxation on code fingerprint,* which means only certain token types, will be considered for the clone detection (step B). Second, the clone detection method will be applied *independently* on each token type (a.k.a. *dimension*) previously saved in the knowledge bases (step C), which is referred as *multi-dimensional matching*.

Matching extreme dissimilarities such as large gaps or repetitions (Fig. 1) is a major challenge. We address this issue using Semantic Web based transitive closure querying and reasoning. As a result, our *pattern matching* mechanism (Fig. 2 step D) can handle extreme dissimilarities. Moreover, in order to guarantee the semantic relevancy of the candidate clone pairs, we use a *Set theory* function (Jaccard coefficient) to apply semantic comparison (*semantic matching*), which is shown in Fig. 2 step D (the bottom process). At the end, the combination of two data sources (i.e. dimensions in step C) and two clone detection types (step D) will result in four similarity values for each candidate clone-pair. SeByte is considered to be a metric-based clone detection approach (according to Roy et al.'s [1] definition), which uses Semantic Web for *pattern matching* to achieve high recall and Set theory for *similarity matching* to keep the precision (step D) high as well. Therefore, we use *similarity matching* to cancel out the *pattern matching* deficiency (high false positive rate). Like Juergens et al. [4] pointed out, the reduction of false positives is often essential to increase the usability and acceptance of a clone detection approach.

Our research questions being addressed in this paper are as follows:

1. Can semantic clones at method blocks be detected using code fingerprint relaxation and the metric-based matching on Java bytecode?
2. What is the performance of the approach with respect to precision and recall?
3. What are the limitations of the presented approach?

The remainder of this paper is organized as follows: Section II discusses basic challenges. Section III reviews Java bytecode. Heuristics, our approach, Semantic Web-based pattern matching, and evaluation results are presented in Sections IV, V, VI and VII respectively. Related Work is reviewed in Section VIII with conclusions in Section IX.

II. CHALLENGES: MAJOR PATTERNS IN METHOD LEVEL CLONE MATCHING

In this research, we are interested to find semantic clone-pairs which also share some degrees of syntactical resemblance. Fig. 3 illustrates seven examples of such clone-pairs.

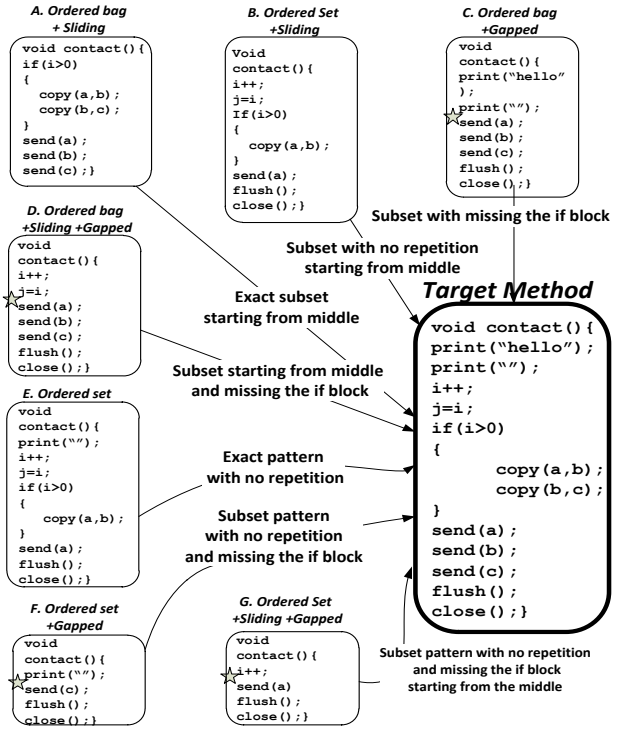

Figure 3. Seven interesting parings matched with the target method. Note the extreme differences due to repetitions, slidings, and gaps.

The seven clone-pairs are chosen and categorized based on their sources of syntactical dissimilarities. The major sources are *repetition*, *sliding*, and *gaps* (marked by asterisk). We use *ordered bag* and *ordered set* to denote the presence or absence of *repetitions*. Note that in this paper, sliding indicates upwards or downwards block shifting in the source code between a pairing which leads to finding the matching points. For example in Fig. 3 code block A is an exact copy (i.e. ordered bag with no gap) of a part of the target method. It is possible to match the two blocks, by

sliding block A through the target method. Block B requires not only the sliding but also ignoring of repetitions (i.e. ordered set with no gap) to establish the similarity. Code block E illustrates a case where both blocks have the same ordered set of statements, whereas in block F we have distinct set of commands from the beginning and ending of the target method including a gap in between. Blocks C, D, and G represent some possible combinations of blocks A, B, E and F. Further, we aim to detect not only these seven patterns but also all their possible combinations, making the detection method an inherently more challenging task.

III. JAVA BYTECODE OVERVIEW

Similar to other low level languages (i.e., machine level), Java bytecode uses machine instructions to simulate basic functionalities such as conditions and loops. There exist several approaches to convert Java bytecode to a structured format from which facts can easily be extracted. These approaches include, (1) Java's environment native commands (used in our approach), (2) Jimple [5] which provides a three address code presentation, and (3) Davis et al.'s Javap2 [6]. Fig. 2 shows an example output of the first approach that we used. Different types of tokens such as Java virtual machine instructions, strings, method names and Java type names are available in the bytecode representation. These tokens form the *code fingerprint* in our case. In the following, we use Java bytecode, bytecode, and binary keywords interchangeably to refer to any content similar to the one in Fig. 2 (output from step A).

Opportunity. Clone detection at bytecode level can detect clone pairs, which might not be syntactically similar at source code level but are in fact semantically similar. Since through the compilation of the source code to a binary format a unified representation of source code is generated. At the bytecode level (1) method inlining has taken place, and (2) syntactic dissimilarities of various loops and conditional blocks in the source code have been transformed to unified format. As a result, bytecode can eliminate some of the challenges for semantic clone detection.

Challenge. While compilation techniques such as method inlining are useful in semantic clone detection, they also introduce new challenges. Fig. 4 shows one of these clone detection challenges caused by method inlining in bytecode. In this example, the size of the method *send()* in the first fragment is large in terms of lines of code and it has been considered for inlining. The resulting dissimilarity between the two fragments at bytecode level will increase by several folds, making the detection of these methods as clone pairs inherently difficult.

IV. FROM CODE FINGERPRINTING RELAXATION TO MULTI-DIMENSIONAL COMPARISON

A major part of the clone detection involves matching source code content (e.g., from AST). The state of the art is to consider the sequence of source code statements as a single fused information source to be compared.

Contrary to the state of the art in clone detection and search, we established in our approach a novel heuristic called *Relaxation on Code Fingerprint* which leads to a *Multi-dimensional Comparison* approach (2-dimensional in our case). Instead of comparing code content as sole fused fact sequence, we extract two data families (i.e., sequences), each of which constitutes a dimension. Each dimension represents only part of the method block's characteristics. We then compare these generated dimensions *independently* using the clone detection algorithm to detect candidate clone-pairs. Clone-pairs detected for each dimensions are then used as input to a decision making function for final results.

Fig. 2 step B shows an example for such a *relaxation on code fingerprinting*. In the bytecode column, Java type fingerprints are marked as bold and method names are underlined. The first dimension contains the names of accessed Java types denoted by t. The second dimension only contains the name of called methods denoted by m. Both dimensions contain ordered sequences, based on their actual appearances in the bytecode (e.g., Fig. 2). Due to our relaxation heuristic, we ignore everything else such as strings and virtual machine instructions.

The underlying rationale of the relaxation on code fingerprint is to develop a robust clone detection approach that can survive even extreme dissimilarities. Using our 2-dimensional matching, we can increase the recall for *semantic clones* by comparing each data family independently.

Our 2-dimensional approach also reduces input data size for the clone detection process since for each dimension only a subset of the available data is considered for comparison – either the names of the called methods or Java types. Fig. 5 shows the effectiveness (i.e. reduction) of our approach in terms of number of token to be analyzed. Using this fingerprinting approach for our bytecode datasets (Table 1), we were able to achieve a reduction in data size of 50-80%. The size reduction is an important for our approach, since its computation complexity is high (e.g., $O(n^2)$). Therefore, our 2-dimensional approach not only allows for the (1) detection of clone-pairs with extreme pattern dissimilarity but also (2) increases performance of our approach by several folds.

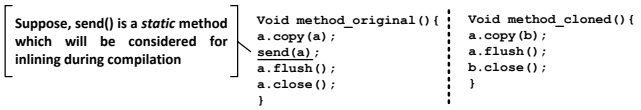

Figure 4. The dissimilarity at source code level is only one line, at the binary level due to method inlining, it depends on the size of method *send()*

TABLE I. DATASETS

Dataset	*Size (#file)*		*Application Context*
	Bytecode	*Source code*	
EIRC	83	64	Network-based comm. client
Freecol (server)	220	79	Server application
Freecol (full)	1120	570	A strategy-based game
Apache (DB)	1093	448	Database system

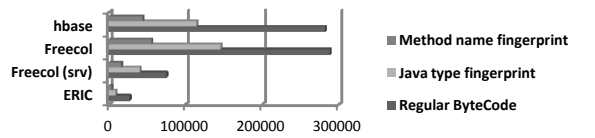

Figure 5. Effects of the code fingerprint relaxation on data reduction (with respect to the number of tokens)

V. OUR APPROACH

In what follows we introduce SeByte, our novel hybrid approach for Java bytecode clone detection. SeByte combines metric-based clone detection [1] with pattern based clone detection. This approach computes the similarity independent measures [7, 8] (e.g. LOC) to find clone pairs and then use a similarity threshold to determine the actual clone pairs. In pattern-based clone detection (e.g., [9]), on the other hand, dissimilarity thresholds (e.g., number of consecutive dissimilar tokens) play a major role in the detection of clones. SeByte takes advantage of both metric and pattern based approaches, by engaging the pattern-based clone detection as one of the similarity metrics. The major steps being applied by SeByte are:

Repository Preparation. Based on our fingerprinting relaxation and 2-dimensional heuristics, we extract fingerprints for Java types and names of called methods, which we publish to two Semantic Web repositories (Fig. 2).

Pattern Similarity. Using Semantic Web based techniques, we find candidate clone-pairs based on their pattern resemblance. Note, we consider patterns as similar, if their *order* of tokens is identical. The objective of this phase is to detect clone pairs that share similar code patterns (i.e. fingerprints). As a result of our Semantic Web-based approach, SeByte can detect clone-pairs even in scenarios with very *gapped* clones such as, method pair $\langle a_1, a_2, a_3, a_4, \dots, a_{1000}\rangle$ and $\langle a_1, a_2, a_{1000}\rangle$. Our approach also can survive *repetitions* with high frequency within a code block, such as $\langle a_1, b, b, b, \dots, b, a_2\rangle$ and $\langle a_1, b, a_2\rangle$. It also supports sliding for clone detection such as $\langle a, b, c, d, e, f, g, h\rangle$ and $\langle c, d, e, f, g, h\rangle$. More importantly, its search power is not limited to these three separated examples, but also supports combinations of them for every single token in the sequence (similar to Fig. 3 samples and their combinations thereof). The resulting search (detection) approach is very comprehensive and finds various types of similar method blocks based on these patterns. However, due to the number of possible matches, it also increases the false positive rate.

Content Similarity. As part of our detection of semantic clones, we also consider content similarity values, to allow for the removal of false positive results detected during the pattern matching step. Instead of using primitive measures such as LOC, we utilize Set theory functions. We measure the content resemblance among two method blocks using the *Jaccard Coefficient* (1) as a comparison function. We denote s_1 and s_2 as the method dimension m or type dimension t for the two subject code fragments respectively. Note s_i is a *set* therefore neither repeated elements nor orderings between elements exist. The key responsibility of content similarity process is to calculate the semantic resemblance of two method blocks based on their contents (e.g., tokens) *regardless* of the *order* of the elements in them. We should note that the ordering of elements has already been addressed in the pattern match part of our approach above.

$$J(s_1, s_2) = \frac{|s_1 \cap s_2|}{|s_1 \cup s_2|} \quad (1)$$

Metric-based Clone Detection. In this step, we combine the *Jaccard* and *pattern* similarity results for each of the two dimensions (the Java types and method calls) to detect final clone pairs. We thus have four metrics (presented in the last column of Fig. 2) for detecting the similarity between two code fragments, which are either numerical (content similarity) or ordinal (pattern matching). In particular, we calculate the overall similarity of two code fragments, c_a and c_b using the $Detect(c_a, c_b)$ function (2) where we use the *intersection* of four sub-functions for each of the metrics. c_i denotes the input method block, with its extracted dimension method m_i and Java type t_i. In our approach we calculate approximately twenty pattern similarity values for each of the clone pair candidates in order to cover the cases shown in Fig. 3, and their combinations thereof. The number of queries is denoted by q in (2). The first boolean function considers clone pair as candidates, as long as one of the queries using the union operator returns a positive value. We designed our Jaccard similarity functions (i.e., *js*) to report its final result as a boolean. It means that the degree of similarity of the candidate parings must be more than a static value denoted by ω, and φ to be detected as clone pair by *js* functions. Finally, the intersection of boolean values for *js* and *query* functions determines whether the candidate clone pair is selected or discarded. The computation complexity of this approach is $O(l * n^2)$, where l is the largest method size. However, l can be considered as a constant therefore our actual complexity is $O(n^2)$.

$$Detect(c_a, c_b) = \bigcup_{x=1}^{q} query_x(t_a, t_b) \cap \bigcup_{x=1}^{q} query_x(m_a, m_b) \cap Js(t_a, t_b, \varphi) \cap Js(m_a, m_b, \omega) \quad (2)$$

VI. TRANSITIVITY VIA SEMANTIC WEB QUERYING AND REASONING

The Semantic Web[1] provides an open scalable logic-based computation platform, which has evolved over the last 10 years from a pure research to an actual industrial strength technology. Its primary goal is to bring *openness* to *knowledge modeling* and *reasoning*. Based on its theoretical aspects, several data modeling languages for graphs and querying languages have been proposed, implemented, and standardized. OWL is the primary modeling language which supports up to First-Order logic with SPARQL being the standard OWL compatible graph-based query language. One of the unique features of the Semantic Web compared to relational databases is its native support for transitive closure computation. Significant progress has been made in the last couple of years in enhancing the performance of Semantic Web reasoners by optimizing them both for in-memory and disk-based computation of scalable transitive closure.

In this section, we discuss how the Semantic Web can be applied towards pattern matching in clone detection. The use of Semantic Web technologies simplifies not only the creation of queries to model patterns in a target code block, but also queries can also be executed against an existing

[1] http://www.w3.org/standards/semanticweb/

pattern knowledge base. A sample query code fragment and its corresponding simplified SPARQL query are shown in Fig. 6. By default, the query finds all blocks with the same nodes {Root→B→C→E} (considering their order).

In order to detect similar patterns with minor or extreme dissimilarities, the query engine (including the inference engine) takes advantage of the transitive property (i.e., `hasConnectionTo`). Executing the same query with reasoning enabled configuration will detect approximately similar code blocks with gaps, repetitions and sliding (e.g., Fig. 6 Root_2 method block). Note that in this example we only used single query to model some aspects of *exact and approximate code pattern matching* to illustrate the power of the Semantic Web. In order to support all possible dissimilarities, a combination of several queries is required.

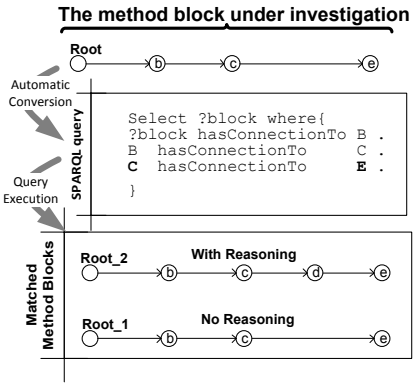

Figure 6. Code pattern matching using Semantic Web reasoning

VII. CALIBRATION AND EVALUATION

For the evaluation of our bytecode clone detection approach, we selected projects from different application domains and of different sizes (Table 1). Datasets were manually extracted and checked for completeness. For each dataset, we created two equivalent subsets, (1) the bytecode and (2) the source code collections. Given the compilation effects, the corresponding collections contain different number of files, but their overall contents remain very similar. Note that, we adopted the EIRC source code set from Bellon et al.'s [10] oracle.

A. Calibration

In SeByte, we used two different thresholds to evaluate the resemblance of method content. The two thresholds are (1) ω (Jaccard threshold for method similarity) and (2) φ (Jaccard threshold for Java type similarity). The objective of our calibration process is to determine the values for ω and φ through an empirical analysis such as that the precision and recall can be optimized. In what follows we describe the major steps of this calibration approach.

Step 1. We manually created an oracle for *bytecode* clones, by annotating 700 candidate semantic pairings (including both true and false positives) using EIRC's binary representation. The similarities of these pairings were then verified against EIRC's source code level similarities. From this analysis we then manually determined if a clone pair should be considered a *semantic clone pair* (true positive) or not (false positive).

Step 2. In order to determine the optimum combination values for ω and φ, we calculated the F-measure for all observed combinations (~ 6400 cases) based on $0.1 \leq \varphi \leq 0.9$ and $0.1 \leq \omega \leq 0.9$ with our window size being equal to 0.01. Fig. 7 shows the F-measure from the front and back.

From the calibration experiment, we were able to identify the values for the ω and φ where the F-measure peaked in Fig. 7 (both precision and recall were optimized). The optimum combination for the dimension thresholds are $\{\omega = 5.3,\ \varphi = 1.9\}$. It means that if the pairing's contents are ~50% and ~20% similar (according to the comparison function) for method calls and Java type dimensions respectively, the candidate is true positive from the semantic similarity point of view with high confidence.

Step 3. As a part of the validation phase, we further validated the selected range for thresholds by random clone-pair checking on three other datasets (Table 1). We evaluated around 500 clone pairs manually, in order to determine if we can find a configuration (other than the one in Step 2), which might lead to better results, and experienced that Step 2 combination was the best.

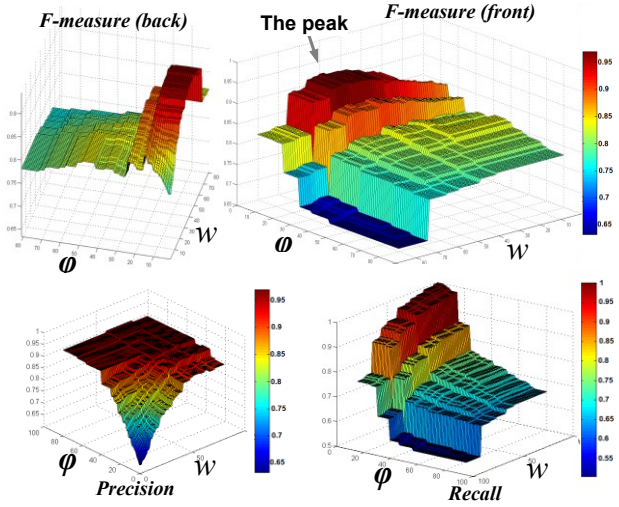

Figure 7. The F-measure, precision and recall using all combinations of two thresholds based on the manually made clone oracle for Java bytecode

B. Evaluation

Run-time. After determining the threshold settings, we conducted some performance evaluation in terms of run-time performance and agreements on the results. Fig. 8 shows the processing time for some major queries (with and without reasoning enabled), and processing steps including Jaccard similarity computation. The results show that for medium size projects, our approach can complete the clone detection process within a few minutes. Moreover, it also shows that the bottleneck is not the computation of the transitive closure for pattern matching, which is an essential observation to support our Semantic Web approach to clone detection.

Scalability. Apparently, the semantic similarity could be a potential threat to SeByte (Fig. 8). For the investigation of our approach we used several large enterprise Java projects

as input data. We successfully used SeByte to detect clones on binary code as large as 300 KLOC on a desktop computer with single core CPU and 3 GB RAM. For larger projects, additional memory is required and our built-in reasoner should be replaced with a Semantic Web inference engine that scales well up to tera-byte data (e.g., [11]).

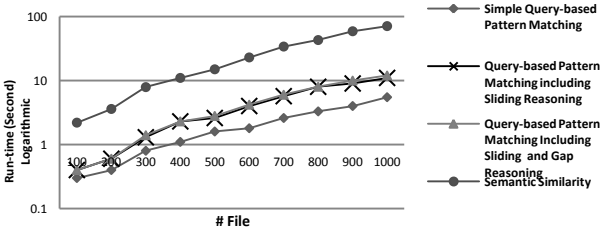

Figure 8. Processing time report categorized based on the computation type. It shows that the time increases significantly for semantic similarity.

Comparison. In the next step of our evaluation, we assessed the performance of SeByte in terms of agreements. The goal was to see whether SeByte is able to detect clones that are missed by other clone detection tools and vice versa. We compared its result with the tools listed in Table 2.

TABLE II. SUMMARY OF CLONE DETECTION TOOLS AND COMPARISON METHOD DETAILS

Tool	*Input Data*	*Given Input Type*	*Comparison Method*	*Tool Granularity*	*Tool Original Purpose*
NiCad [9]	Table 1	Source Code	Automatic	Method-level	Near-Miss Source Code Detection
Merlo dataset (From Bellon Dataset [10])	Bellon Oracle	Source Code	Automatic	Method-level	Metric-based Method-Level Type-3 Source Code Clone Detection
Scorpio [12]	Table 1	Source Code	Manual (sampling)	Line-level (token)	Gapped Clones on Source Code PDG
JCD [13]	Table 1	Binary	Manual (sampling)	Line-level (pcode)	Type-3 Binary Clone Detection
SimCad [14]	Table 1	Binary	Automatic	Method-level	Near-Miss Source Code Detection

Note that, we have used NiCad (near-miss clone detector) and Scorpio (semantic clone detector) on the source code level, and JCD (type-3 on Java binary) and SimCad (source code based near-miss clone detector) on bytecode level. Note that, SeByte detects clone-pairs at *method-level*, so in theory it is only feasible to compare its result *automatically* with tools working at the same granularity. Therefore, NiCad, Merlo's clone set from Bellon et al. [10], and SimCad are only options for automatic comparisons. However, we attempted to manually compare with other tools with their different capabilities.

The agreements/disagreements with NiCad including a detailed report is shown in Table 3. As expected the agreement percentage is not so high, since each one has been designed to detect different types of clones. Moreover, this observation complies with earlier studies both by Selim et al. [15] and Davis and Godfrey [13], that the disagreement in clones is due to differences between binary and source code. Table 3 also shows that there are no relations between agreement values and project size in this experiment. Moreover, SeByte detects usually large number of clone-pairs since it does not filter out very small-size method blocks. We also used SimCad, originally designed for source code clone detection on binary (based on the approach by Selim et al. and Baker and Manber [16]). Again, as expected the agreement between SeByte and SimCad was very low, less than 20% on average. Finally, we compared SeByte with Merlo's clone detection tool (CLAN) [10]. The total observed agreement for Type 1 and 2 clones was about 18% while for Type 3 was negligible.

TABLE III. SEBYTE AND NICAD RESULT COMPARISION

	Runtime (second)		*# Clone pair*		*# Clone class*		*Agreement*
	SeByte Distributed	*SeByte* Single core	*SeByte*	*NiCad*	*SeByte*	*NiCad*	*(~%)*
EIRC	0.4	1.9	198	63	24	17	40%
FreeCol (server)	4	16	708	43	46	19	70%
Freecol (full)	135	414	1593	1339	149	305	60%
Apache (DB)	132	491	26955	15378	190	297	30%

Furthermore, we manually compared SeByte with JCD on binary, and Scorpio on source code content. We used JCD 1.0.10 using the same configuration recommended by its authors. Since Scorpio requires large amount of memory for Java heap and stack, we executed it on dedicated hardware with 24 GB RAM. As noted earlier, we were unable to automate the comparison process for JCD and Scorpio since both of them detect clones at finer granularity levels than SeByte. We therefore manually verified whether results from SeByte (i.e. cloned methods) are detected by them within a reasonable coverage. For JCD the agreement ratio was ~40%, whereas for Scorpio it was negligible.

Summary. The primary reason for the low agreement ratio was due to different search approaches, goals, objectives, and input data. While each tool achieves an acceptable recall/precision based on the tool specific definition of a clone, each tool reports different clone-pairs compared to other tools. While for example SeByte and Scorpio support specifically *semantically similar and gapped clones,* their different heuristics and input information (source code vs. bytecode), result in almost completely different result sets.

C. Answers to the Research Questions

In what follows we revisit our original research questions listed in the Introduction. From our experimental evaluation, we were able to observe that SeByte can detect semantic clones that are missed by other source code or bytecode based tools. As a result, the semantic clones reported by SeByte are complimentary to existing clone detection tools.

Quality. According to the result in the calibration phase (Fig. 7), 92% is the best recall that SeByte can achieve for semantic clones at *method-level* using bytecode content. Note that, this recall is for the clones that can actually be detected using only binary information – and might not reflect the clones that can be detected in the source code. We also measured the precision of our approach by manually checking 500 randomly selected clone pairs from our two large datasets. The achieved average precision was around 79%. This observed precision matches the results from our 6400 experiments, which we performed during our calibration step. Since, there are no other clone detection approaches for *Java bytecode* which support *semantic clone*

detection at *method-level,* there is no other way to evaluate SeByte on a head to head comparison.

Threats to Validity and SeByte Limitations. We calibrated SeByte based on a manually annotated dataset which we created. Therefore, it is possible that the selected thresholds are not the best due to bias or error in our oracle. Regarding the last research question, the mediocre agreements in Table 3 illustrates a major limitation (and strengths) of our approach. It shows that our approach is able to detect clones, which are missed by other tools, at the same time, it misses clones detected by other tools. Therefore, we consider SeByte to be a complimentary approach to improve the overall recall of the state of the art clone detection tools.

VIII. Related Work

Metric-based clone detection [7, 8, 17] is one of the scalable approaches. Unique features of metric-based approach are evaluated by Bellon et al. [10]. There are also several semantic clone detection approaches [3, 12, 18] proposed in the literature. However, two main challenges remain for this research domain. First, there is no well-established definition for semantic clones. Second, the problem in general is undecidable [2]. Recently, diverse approaches are proposed such as (1) a formal method-based approach for embedded systems [18], (2) clustering of entities in different granularities to achieve scalability [3].

Binary code clone detection has not been a major research focus in the clone detection community. Baker and Manber [16] used a combination of three comparison based approaches such as Diff on almost the similar bytecode representation that we have used (e.g. Fig. 2 Section B) to detect syntactical clone (e.g. type 3). The JCD project [13] developed by Davis and Godfrey uses a combination of hill climbing and greedy algorithms to detect the maximum coverage (including a pretty-printing tool [6]). There is also a proposal to use process algebra on bytecode [19]. Selim et al. [15] converted bytecode to the Jimple format [5] and used third-party tools (originally designed for source code) on the Jimple content. They reported maximum of 49% and 78% agreements between clones from bytecode and source code. This agreement ratio is similar to ours between SeByte and NiCad and also supports the fact that clone detection at source code and bytecode lead to diverse but complementary results. Nevertheless, we showed that SeByte also detects diverse results from such approach (i.e., Selim et al. [15]) by comparing SeByte with SimCad on bytecode. The comparison not only shows the usefulness of using bytecode for clone detection but also highlights the strengths of our heuristics and Semantic Web in clone detection.

IX. Conclusion and Future Work

In this research, we introduced the idea of relaxation on code fingerprint which leads to a 2-dimensional code fragment comparison. The motivation was to find semantic clones which hold high and moderate similarity degrees from semantic and syntactic perspectives respectively. In other words, it helps the search algorithm to survive in case of extreme syntax dissimilarities to find semantic similarities. We devised a metric-based approach with two major criteria: (1) *Pattern similarity*, which is done using Semantic Web querying and reasoning, and (2) *Content similarity*, which is achieved using Jaccard coefficient. In general, SeByte provides the first metric-based approach for semantic clone detection on Java bytecode. By comparing to five tools with different clone detection algorithms, we showed that our approach is able to detect clones which are missed by them. As future work, we plan to (1) apply SeByte on large-scale subject systems not only on binary but also source code levels, (2) compare with other state of the art tools, and (3) examine whether certain combinations of the source code and binary detection produce satisfactory results.

References

[1] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach", Science of Com. Prog., vol. 74, no. 7, May 2009, pp. 470-495.

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

[3] S. Yoshioka, N. Yoshida, K. Fushida, and H. Iida, "Scalable Detection of Semantic Clones Based on Two-Stage Clustering", Proc. ISSRE, 2011, pp. 3-4.

[4] E. Juergens and N. Göde, "Achieving Accurate Clone Detection Results", Proc. IWSC, 2010, pp. 1-8.

[5] Soot Framework, http://www.sable.mcgill.ca/soot/, (Jan 2012).

[6] Javap2, http://www.swag.uwaterloo.ca/javap2/index.html, (Jan 2012).

[7] Mayrand, C. Leblanc, E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System using Metrics", Proc. ICSM, 1996, pp. 244–253.

[8] J. Patenaude, E. Merlo, M. Dagenais, B. Lague, "Extending Software Quality Assessment Techniques to Java Systems", Proc. IWPC, 1999, pp. 49–56.

[9] C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization", Proc. ICPC, 2008, pp. 172-181.

[10] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools", Tran. Soft. Eng. vol. 33, no. 9, 2007, pp. 577–591.

[11] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, and H. Bal, "OWL Reasoning with WebPIE: Calculating the Closure of 100 Billion Triples", Proc. ESWC, 2010, pp. 213–227.

[12] Y. Higo and S. Kusumoto, "Enhancing Quality of Code Clone Detection with Program Dependency Graph", Proc. WCRE, 2009, pp. 315-316.

[13] I. J. Davis and M. W. Godfrey, "From Whence It Came: Detecting Source Code Clones by Analyzing Assembler", Proc. WCRE, 2010, pp. 242–246.

[14] S. Uddin, C.K. Roy, K.A. Schneider, and A. Hindle, "On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems", Proc. WCRE, 2011, pp. 13-22.

[15] G. M. K. Selim, K. C. Foo, and Y. Zou, "Enhancing Source-Based Clone Detection Using Intermediate Representation", Proc. WCRE, 2010, pp. 227-236.

[16] B. S. Baker and U. Manber, "Deducing Similarities in Java Source from Bytecodes", Proc. ATEC,1998, pp. 179-190.

[17] T. Lavoie and E. Merlo, "Automated Type-3 Clone Oracle Using Levenshtein Metric", Proc. IWSC, 2011, pp. 34-40.

[18] B. Al-Batran, B. Schätz, and B. Hummel, "Semantic Clone Detection for Model-based Development of Embedded Systems", Proc. MoDELS, 2011, pp. 258-272.

[19] A. Santone, "Clone Detection through Process Algebras and Java Bytecode", Proc. IWSC, 2011, pp. 73-74.