ELSEVIER

SeByte: Scalable Clone and Similarity Search for Bytecode

Iman Keivanloo[a], Chanchal K. Roy[b], Juergen Rilling[c]
[a]Department of Electrical and Computer Engineering, Queen's University, Canada
[b]Department of Computer Science, University of Saskatchewan, Canada
[c]Department of Computer Science, Concordia University, Canada

Abstract

While source code clone detection is a well-established research area, finding similar code fragments in binary and other intermediate code representations has been not yet that widely studied. In this paper, we introduce SeByte, a bytecode clone detection and search model that applies semantic-enabled token matching. It is developed based on the idea of relaxation on the code fingerprints. This approach separates the input content based on the types of tokens into different dimensions, with each dimension representing the input content from a specific point of view. Following this approach, SeByte compares each dimension separately and independently which we refer to as multi-dimensional comparison in our research. As the similarity search function we use a well-known measure that supports our multi-dimensional comparison heuristic, the Jaccard similarity coefficient. Our preliminary study shows that SeByte can detect clones that are missed by existing approaches due to the differences in the input data and the search algorithm. We then further exploit the model to build a scalable bytecode clone search engine. This extension meets the requirements of a classical search engine including the ranking of result sets. Our evaluation with a large dataset of 500,000 compiled Java classes, which we extracted from the six most recent versions of the Eclipse IDE, showed that our SeByte search is not only scalable but also capable of providing a reliable ranking.

Keywords: clone detection; Semantic Web; Java bytecode; clone search; semantic search;

1. Introduction

Two code fragments that share syntactic or semantic similarity are considered to be a clone pair. Syntactic similarity refers to the situation where clone pairs share a similar code pattern resulting in Type-1, 2, and 3 clone types [1, 2]. Type-1 clones are exact copies of each other, except for possible differences in whitespaces and comments. Type-2 clones are parameterized copies, where variable names and function calls have been renamed and/or types have been changed. Changes (e.g., addition and deletion of statements) in a clone pair result in Type-3 clones. Semantic similarities (i.e., Type-4), in contrast, focus on pairings' functionality [2] regardless of their code patterns. Source code clone detection has been a major focus of the software analysis research [1] and has resulted in various clone detection techniques (e.g., [3 - 7]). Common to most of these detection techniques is that they have a complete off-line search step to find all possible clone pairs within a static source code repository. Recently, instant, real-time, or just-in-time clone search (e.g., [8]) has been introduced, complementing traditional clone detection approaches. These clone search approaches can be considered to be specialized search engines. Code clone search approaches accept as input a code fragment and provide close to real-time or just-in time search capabilities.

In this research, we are interested in providing a clone search model that handles Java bytecode. Similar to traditional Web search engines, our key objective was to derive a ranking approach that reports relevant cloned fragments (true positives) as top-ranked hits in the result set. In this paper, we first introduce SeByte, a Java bytecode clone search model which is able to detect Type-1, Type-2 and Type-3 clones[a] at the bytecode level. However, in certain cases, such as the presence of different forms of loops at source code level, our Type-3 bytecode clones may actually be considered Type-4 clones [2] at the source code level. Since Java bytecode is a low level programming language compared to common high level languages such as Java, we propose a novel set of heuristics to support clone search at the bytecode level. We first introduce our relaxation on code fingerprints heuristic. This heuristic considers only certain types of tokens during clone detection. Second, we introduce what we refer to in this paper as a "multi-dimensional matching" heuristic. This heuristic applies the clone detection algorithm independently on each type of tokens (a.k.a., dimension). Third, we use Jaccard similarity coefficient [36] as the core similarity calculation function. Furthermore, we extend the search model to support the notion of semantic search [10] using a manually created ontology which represents the semantic network (e.g., [9]) of Java bytecode instructions. This model allows us to provide a light weight semantic-enabled token matching approach to recover some of the clones that are missed by our pure syntactical matching. The semantic search is useful in particular for intermediate languages, since many instructions (e.g., summation instructions) include additional embedded meta information (e.g., data type). This additional information decreases the recall [21] of syntactical matching.

We have studied the performance of our search model using a large dataset consisting of 500K compiled Java classes and 73 million lines of bytecode. We deliberately chose some specialized measures from Information Retrieval (IR) and Web search domains to evaluate the quality of the "ranked" result set. The studied measures are First False Positive, Precision at K, and Normalized Discounted Cumulative Gain (NDCG). Our experience with this large dataset and special measures shows that our model is capable of providing high quality ranking for Java bytecode.

The remainder of this paper is organized as follows: Section 2 provides an overview of Java bytecode, followed by Section 3 which discusses opportunities and challenges of token matching for bytecode clone detection. Our heuristics for data manipulation and clone search approach are presented in Sections 4 and 5. The two major applications of SeByte, clone detection and search, are evaluated in Sections 6 and 7 respectively. Related work is reviewed in Section 8 with conclusions in Section 9.

2. Java Bytecode Overview

Java bytecode is designed as a stack-oriented language, with the stack being the major computation entity in the Java runtime environment. The compiler translates source code statements into their corresponding Java bytecode instructions. This mapping is usually one-to-many, since Java bytecode is limited to primitive instructions for stack manipulation such as simple push and pops. A total of 256 instructions[b] are defined as part of the Java bytecode language specification. These instructions can be classified in 10 major instructions families (summarized in Table 1) based on the Java 7 specifications.

Table 1 emphasizes some key aspect of Java bytecode, namely the fact that many bytecode instructions also include additional embedded information, like the data type for which a specific instruction is applicable. For example, several variations of the symbolic load instruction are available in Java bytecode (e.g., iload, iload_0, dload, lload, fload, and aaload), with the prefix specifying in this case the data type which is manipulated. Table 2 illustrates how some implicit semantics captured in these bytecode instructions can be further interpreted. There are other pre/postfixes which are less popular such as postfixes belong to "comparison instruction family" (e.g., g in "fcmpg").

Figure 1 shows a Java bytecode fragment in a plain text format. For example, the instruction in line 127 pushes an Integer with value 0 to the stack. Line 122 shows a method call statement, which is calling println from the java.io.PrintStream class. Each statement contains several types of information in a single bytecode line such as instruction, the class name, and method name.

[a]In this research, a true positive must be at least Type-1, 2, or 3 clone (necessary condition) and also holds similar functionality (sufficient condition)

[b]http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

```
…
122: invokevirtual   java/io/PrintStream.println:(I)V
123: astore_1
124: aload_1
125: arraylength
126: istore_2
127: iconst_0
128: istore_3
129: iload_3
130: iload_2
…
```

Figure 1. Java bytecode example (presented as plain text)

Table 1. The Java bytecode instruction overview

Instruction Family	Description	Example
Types	This family covers several areas such as: (1) Load and store data onto/from the stack from/to local variables etc. (2) Primitive arithmetic functions such as add, multiply etc. (3) Data type conversion	"dload" loads a Double local variable onto the stack. "dadd" sums up Double values. i2d converts Integer-typed value to Double format.
Load and Store	This is a dedicated family to two major types of instructions related to the stack which are loading onto and storing from the stack.	"dstore" stores a Double value from top of the stack to a local variable.
Arithmetic	This family has primitive instructions required for arithmetic and logical computation. The required data (all values e.g., 2) will be retrieved from the stack and the result will be saved onto the stack. The major families of functions are Add, Subtract, Multiply, Divide, Remainder, Negate, Shift, Bitwise OR, Bitwise AND, Bitwise exclusive OR, increment, and Comparison.	"fadd", "ishr" (Shift right Integer value), "ior", "iinc" (such as var++), fcmpg (compare – the greater operand)
Type Conversion	The dedicated family for type conversion	"i2d" and "i2f"
Object Creation and Manipulation	Create, load, and store object or array instances. Note that Java provides dedicated instructions for array creation and manipulation.	"new", "newarray", "getfield" (access Java classes' fields), "iaload" (load an array of Integer type to the stack), "arraylength", "instanceof"
Operand Stack Management	Primitive operations required for stack manipulation. These operations change the state of the stack directly.	"pop", "dup", "swap"
Control Transfer	Program control flow instructions. Several types of "if" are provided for simulation of all possible conditional branches.	"ifeq", "ifnull", "goto"
Method Invocation and Return	The instructions for handling method call statements are presented under this family.	"invokevirtual", "invokeinterface", "ireturn"
Exceptions	The dedicated family for error handling	"athrow"
Synchronization	The primitive instructions for synchronization in case of concurrency. Note that the specified synchronization semantics at the source code will be handled using monitor enter and monitor exit	"monitorenter" (specifies entering the secured code block in terms of concurrency)

Table 2. The symbol table assigned to known data types by Java bytecode

Symbol → The corresponding type			
a → reference	i → integer	s → short	l → long
c → character	b → byte	f → float	d → double

3. Semantic Search for Java Bytecode

3.1. Problem definition

As discussed in Section 2, the Java bytecode representation contains less ambiguity compared to higher-level source code due to the availability of additional explicitly embedded information at the instruction level. For example, bytecode level calculation instructions explicitly include as part of the instruction the data types. As a result, for each primitive data type, there is a dedicated "add" instruction (e.g., *iadd* and *fadd*). Similarly, object creation/access, method call, and field access instructions embed the data types (or other metadata) they can manipulate. That is, during compile time most message type resolutions are computed and captured as part of the bytecode instruction. For example, in line 122 Figure 1, the type of message receiver (i.e., println) is already resolved not only for the receiver class name (PrintStream), versus possible abstractions in the object-oriented programming context such as virtual classes and interfaces, but also for the actual implementation captured both by its fully qualified name (java.io.PrintStream) and the file address. Although, from a clone detection/search perspective, input data with less ambiguity is typically preferred, it reduces recall [21] in case of syntactical matching.

Figure 2 illustrates the challenges for detecting clones at Java bytecode versus source code level. While at source code level only one token (i.e., +) is used to represent the "add" functionality, at bytecode level, the representation of the same functionality depends on the source code's implicit semantics. Therefore, contrary to the source code level clone detection, syntactical matching fails to match all of the summation tokens since they are presented by different tokens at bytecode. As a result, the source code line, `x=x+y` can have four possible corresponding bytecode level representations depending on the data types of the variables `x` and `y`. This issue becomes even more challenging with the inclusion of other statements (e.g., `var.println()`). As a result, there exist $4 \times N \times M$ different bytecode interpretations for the original source code fragment, where N is the number possible instructions at the second line and M is the number of possible types at the first line (Figure 2).

This issue makes clone detection at bytecode level different from source code level. While for example in Figure 2, block A and B at the source code level are identical clones (Type-1), their bytecode representation can be different. Therefore, the corresponding bytecode of the Type-1 source code (blocks A and B) clone-pair can constitute a Type-2 or 3 clone-pair. This example shows the challenges of bytecode clone detection for some clone-pairs which are trivial (i.e., Type-1) to detect using source code.

3.1.1. Existing solutions

In cases where input data contains more information than the clone detection algorithm requires or can process, either preprocessing (e.g., data filtering) or post processing (e.g., result grouping) is applied. For source code content, normalization is commonly used to remove unnecessary differences so that the pattern matching algorithm can return better results. For example, many approaches [17, 19, 20] replace the token names (e.g., class names) with predefined symbols, e.g., $ or enumerated $ where the order information must be preserved such as $1, $2. Such approaches provide the opportunity to detect Type-2 clones at source code level. Similar approaches are also used for clone detection on intermediate languages, e.g., Baker et al.'s approach for Java bytecode [16] and our earlier study on .NET intermediate language [21].

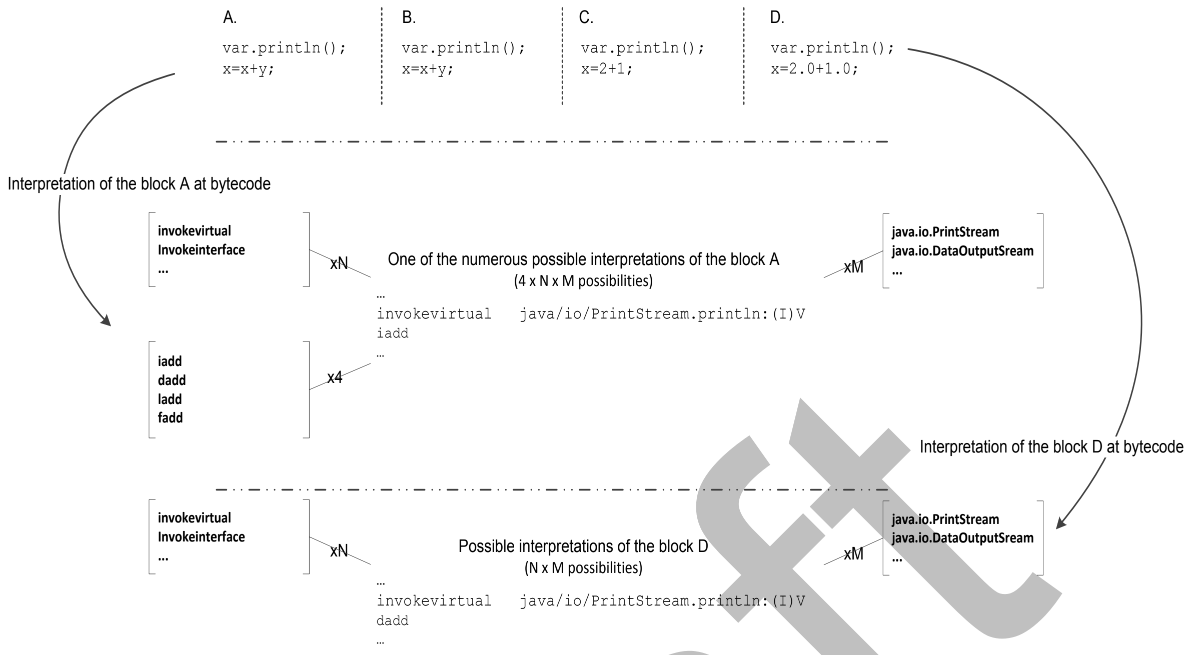

A.
var.println();
x=x+y;
B.
var.println();
x=x+y;
C.
var.println();
x=2+1;
D.
var.println();
x=2.0+1.0;
Interpretation of the block A at bytecode
invokevirtual
Invokeinterface
...
xN
One of the numerous possible interpretations of the block A
(4 x N x M possibilities)
xM
java.io.PrintStream
java.io.DataOutputSream
...
invokevirtual java/io/PrintStream.println:(I)V
iadd
iadd
dadd
ladd
fadd
x4
Interpretation of the block D at bytecode
Possible interpretations of the block D
(N x M possibilities)
invokevirtual java/io/PrintStream.println:(I)V
dadd

Figure 2. A few examples showing the differences between source code and bytecode clone detection

3.1.2. Our solution - semantic search

Existing solutions for intermediate representations have focused on the use of data normalization (e.g., filtering), which often involves some form of data loss. While this approach works well for clone detection, the information loss due to filtering restricts the applicability for our bytecode clone search approach. A key aspect of any search approach is its ability to differentiate and rank hits based on closeness of hits to the query. However, the data loss (including semantics) through data filtering used by traditional clone detection and search approaches will affect their ability of providing an accurate ranking. For example, suppose a user is searching for code fragments that implement the summations of two numbers and more specifically the summation of floats. In the context of clone search, more relevant results should be ranked higher in the result list than other. For our summation example, search results containing a float summation such as fragment D in Figure 2, a Type-1 clone, should be ranked higher than search results containing summations involving other data types, e.g., summation of integer numbers such as fragment C in Figure 2 - i.e., Type-2 clones. Likewise, semantic information associated with other bytecode level instructions can be used to enhance the search and ranking processes. In our approach we take advantage of semantic search [10] which uses the presence and degree of similarities for bytecode content matching, to semantically rank search results.

Presence of Similarity. Given the classification of bytecode level instructions introduced earlier (Section 2), it is possible to identify similar instruction types based on their relationships with each other. Therefore, similar instructions can be identified by analyzing the associated tokens in the domain of discourse. For example, in Figures 3 and 4 *iadd* and *java.io.PrintStream* can be mapped with other tokens in the tree. The core idea is based on the fact that association links can be used to interpret the similarity between tokens and therefore can be used to infer that an iadd token is similar to *dadd* and other siblings in the graph. Furthermore, the available connections, both direct and indirect, between *java.io.PrintStream* and other types (e.g., `PipedOutputStream`) in Figure 4 can also be used as part of this similarity measure.

Degree of Similarity. The types of links and the distances between tokens can be used to interpret the degree of similarity among tokens. This is different from syntactical token matching when the result is boolean (i.e., true or false). For example at source code level, the result of syntactical token matching is true if both tokens are represented using the same character string (e.g., the summation tokens in block A and B at Figure 2). Semantic matching focuses on the existence of links (e.g., *iadd* is related to *dadd* and to *XOR* through the arithmetic instruction set) and their degree of similarity. Such degree of similarity, measures the distance between nodes in the given network (Figure 3). The measure can also be applied to other types of tokens found in bytecode, e.g., class and data types in Figure 4.

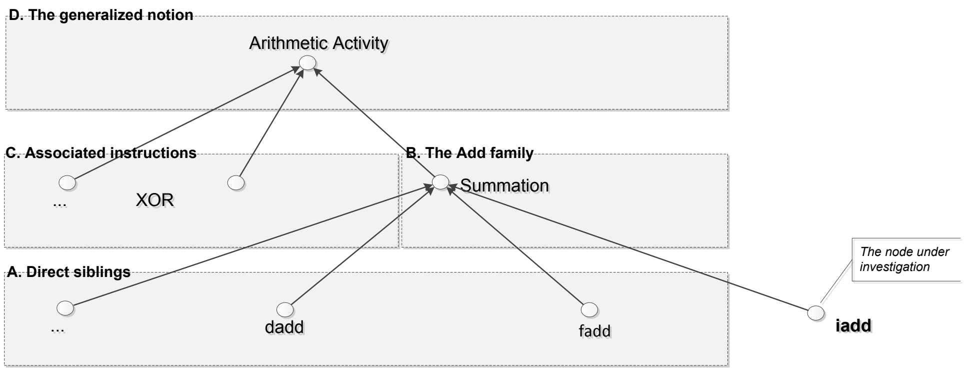


Figure 3. A slice of domain of discourse (i.e., Java bytecode specification) related to iadd instruction

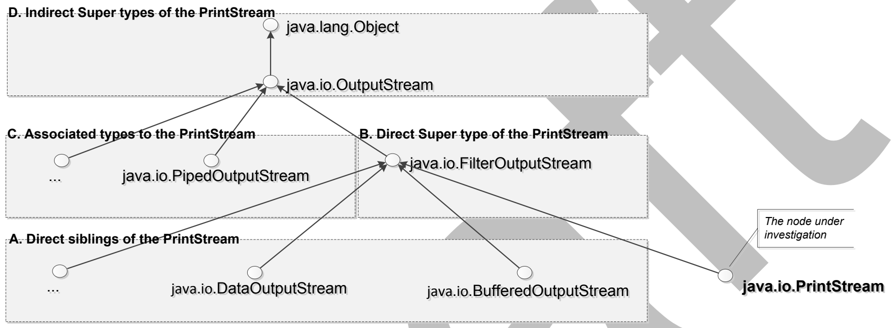

D. Indirect Super types of the PrintStream
java.lang.Object
java.io.OutputStream
C. Associated types to the PrintStream
...
java.io.PipedOutputStream
B. Direct Super type of the PrintStream
java.io.FilterOutputStream
A. Direct siblings of the PrintStream
...
java.io.DataOutputStream
java.io.BufferedOutputStream
The node under investigation
java.io.PrintStream

Figure 4. A slice of domain of discourse (i.e., the program inheritance tree) related to java.io.PrintStream token

During our semantic-enabled token matching, two tokens are matched if there is a path between the nodes in the given semantic network (e.g., Figure 3). Contrary to syntactical matching, the result of semantic-enabled matching is not boolean but rather a degree of similarity. This semantic matching requires a semantic network that models the connections between all possible tokens. We therefore created two ontologies for Java bytecode, one representing the instruction semantic network, and the other representing the inheritance semantic network. The inheritance ontology, depends on the actual input data (code corpus), which is dynamically created through data extraction (e.g., the inheritance tree at Figure 4) during clone detection. For the instruction semantic network, we introduce a manually created static ontology that supports the conceptualization of Java programming language specification, (e.g., Figure 3). We created an ontology called Bytecode Ontology (a.k.a., byteon), which represents a hierarchical conceptualization of bytecode instructions and includes all the 256 bytecode instructions. All instructions are classified into families of related instructions. As discussed earlier, 10 major instruction families can be distinguished at the bytecode level (see Table 1). We extend this initial classification by adding (1) classifications (horizontal extension), and (2) hierarchal relationships between families (vertical extension). For example, intermediate concepts such as “IntergerAccess” are added to associate all functions defined over Integer data types. Using our modeling approach, a family of instructions might subsume other families. The resulting bytecode ontology is available online at the project website[c]. The ontology contains 296 concepts (40 family entities and 256 instructions). Figure 5 provides an overview of the high-level families (and their relations), which are located in the center of the figure (identified by the rectangle). An expanded view of the higher level *Access* concept and its immediate concepts are also provided as an example in Figure 5. A complete overview of the ontology is shown in Figure 6, with its major instructions families being labeled in the circles. The visual complexity of the graph is high due to large number of links since most instruction types belong to several families. While we are using the Bytecode Ontology for Java bytecode level clone search, it can also be reused for other intermediate languages and application contexts.

[c] http://secold.org/projects/sebyte

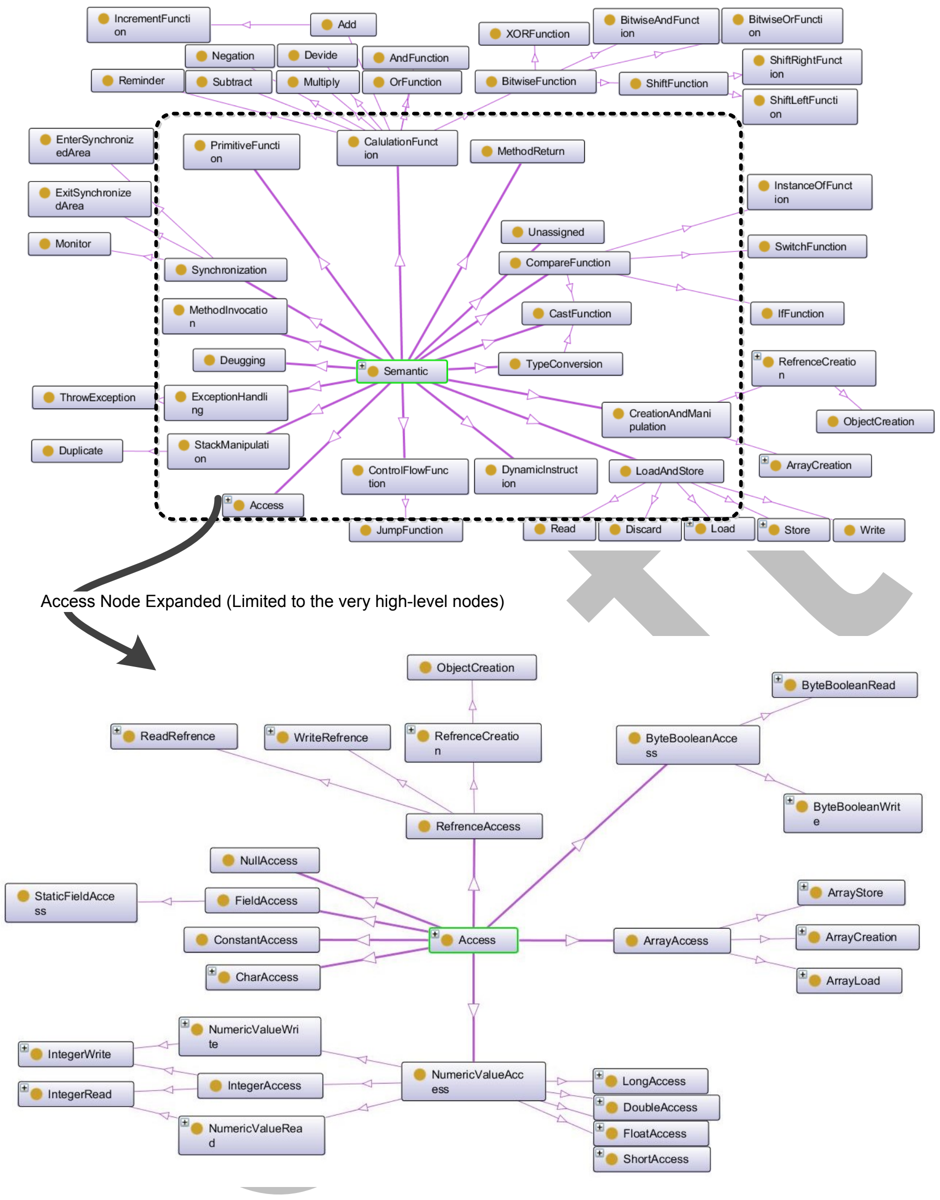


Figure 5. Partial preview of Bytecode Ontology

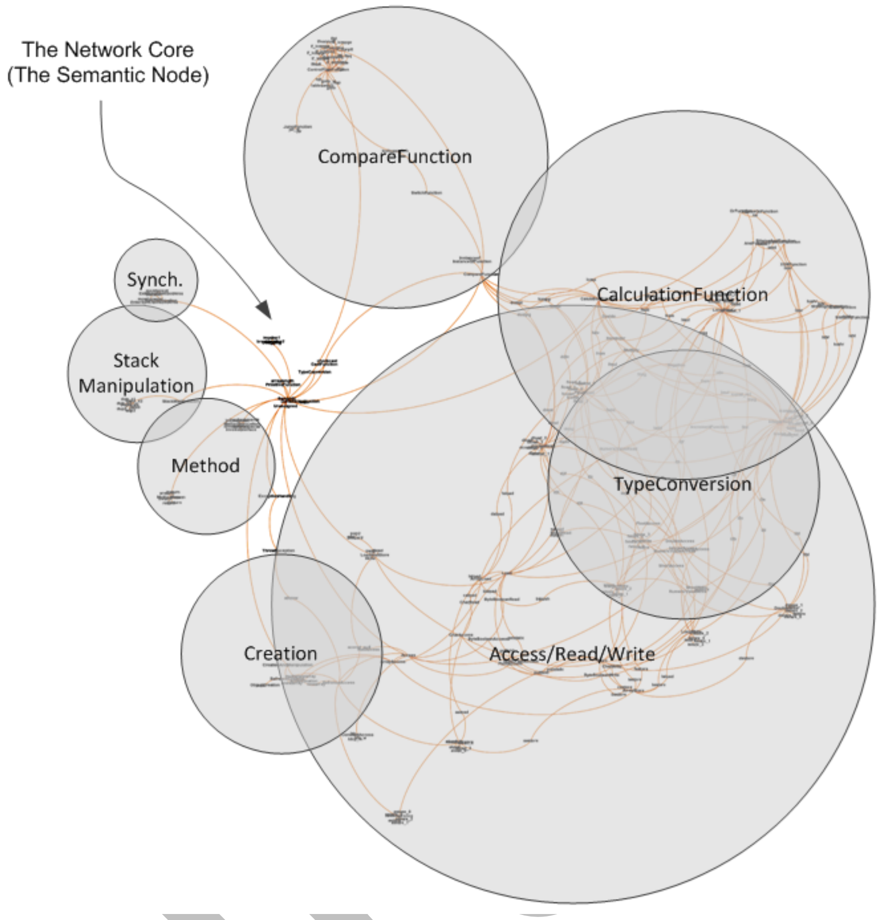


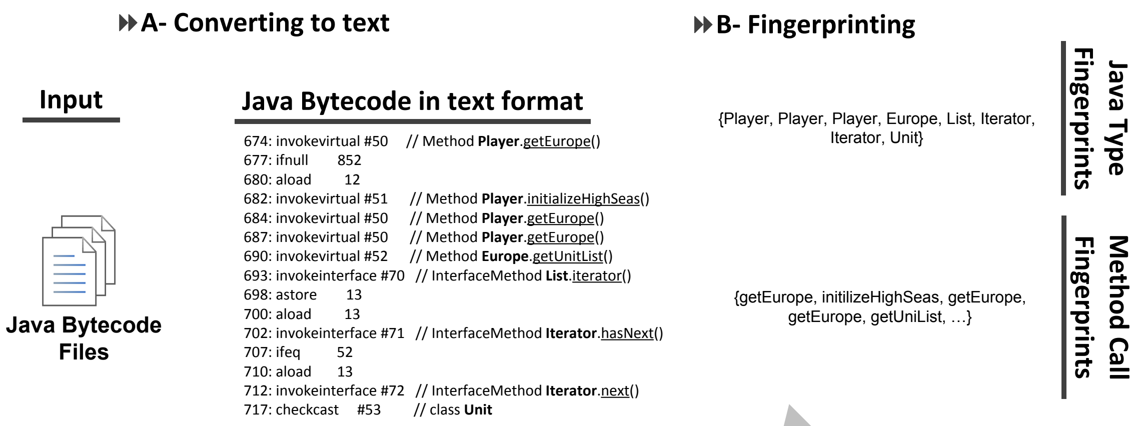
Figure 6. Bytecode Ontology overview highlighted with most popular families

4. The Proposed Data Presentation and Manipulation Approach

A major part of clone detection involves matching source code content. 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 current state of the art, we introduce a heuristic called *relaxation on code fingerprint* which leads to a *multi-dimensional comparison* approach. Instead of comparing code content as sole fused fact sequence, we extract different pieces of information as knowledge resources, each of which corresponds to a dimension in our approach. This approach is motivated by the fact that each Java bytecode statement (e.g., Figure 7) contains several but predefined types of information in a single bytecode line such as the instruction, class name, and method name.

Each dimension provides a specific perspective of a code block. As part of our multi-dimensional comparison approach, we then compare these dimensions *independently* using a similarity search algorithm to detect candidate clone-pairs. We then merge all the result-sets created from the analysis of each dimension to produce our final clone pair set.

Figure 7 shows an example of using two different dimensions as part of the *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. The second dimension contains only the names of the called methods. Based on their actual appearances in the bytecode, all dimensions will be represented using ordered sequences.


A- Converting to text
B- Fingerprinting
Input
Java Bytecode in text format
Java Bytecode Files
Java Type Fingerprints
Method Call Fingerprints
{Player, Player, Player, Europe, List, Iterator, Iterator, Unit}
{getEurope, initilizeHighSeas, getEurope, getEurope, getUniList, …}

```
674: invokevirtual #50    // Method Player.getEurope()
677: ifnull       852
680: aload        12
682: invokevirtual #51    // Method Player.initializeHighSeas()
684: invokevirtual #50    // Method Player.getEurope()
687: invokevirtual #50    // Method Player.getEurope()
690: invokevirtual #52    // Method Europe.getUnitList()
693: invokeinterface #70  // InterfaceMethod List.iterator()
698: astore       13
700: aload        13
702: invokeinterface #71  // InterfaceMethod Iterator.hasNext()
707: ifeq         52
710: aload        13
712: invokeinterface #72  // InterfaceMethod Iterator.next()
717: checkcast    #53      // class Unit
```

Figure 7. Fingerprinting examples of Java bytecode for method and type dimensions

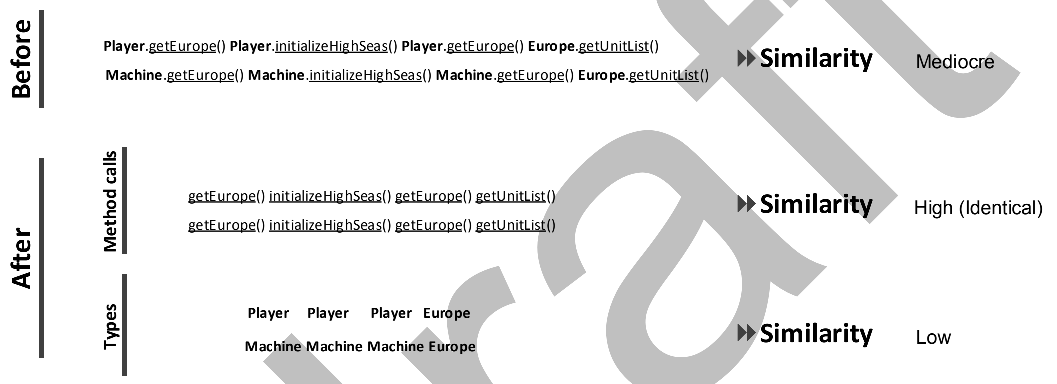

Before
Player.getEurope() Player.initializeHighSeas() Player.getEurope() Europe.getUnitList()
Machine.getEurope() Machine.initializeHighSeas() Machine.getEurope() Europe.getUnitList()
Similarity
Mediocre
After
Method calls
getEurope() initializeHighSeas() getEurope() getUnitList()
getEurope() initializeHighSeas() getEurope() getUnitList()
Similarity
High (Identical)
Types
Player Player Player Europe
Machine Machine Machine Europe
Similarity
Low

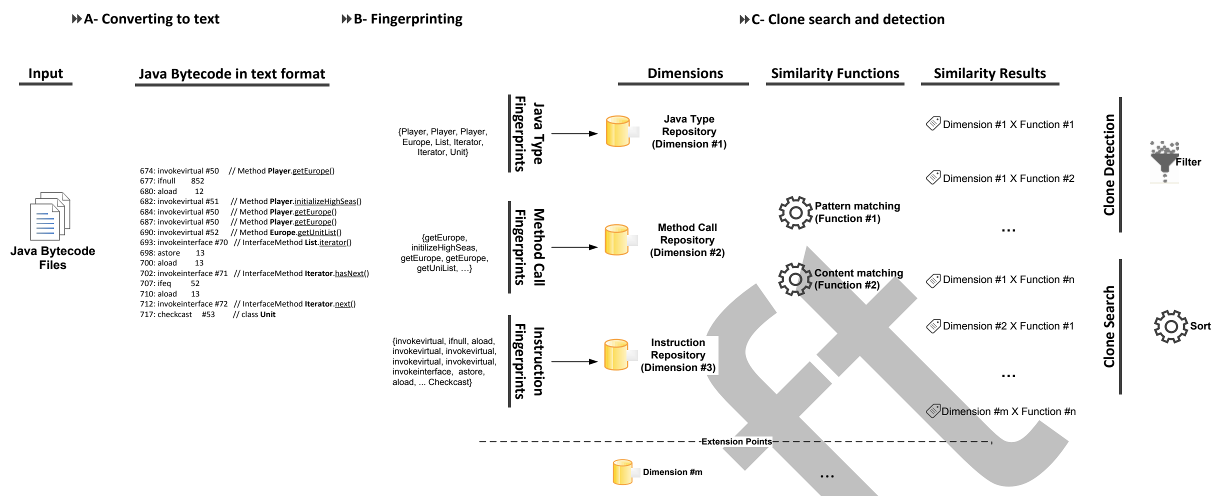
Figure 8. A simplified example which highlights the effect of the relaxation on code fingerprints on similarity calculation

Motivation #1. The underlying rationale of the relaxation on code fingerprint is to develop a similarity search approach that handles extreme dissimilarities limited to one of the dimensions (e.g., type). This feature is useful when the fingerprints of the other dimensions (e.g., method calls) are holding a certain degree of similarity. Using our multi-dimensional matching, we can increase the recall for such clones, by comparing each dimension independently. Therefore, dissimilarity in each dimension is limited to its corresponding result set. Figure 8 illustrates how relaxation on code fingerprint controls the dissimilarity scattered throughout the original pair by separating the concerns.

Motivation #2. The multi-dimensional approach also reduces input data size (search space) for the clone detection process since each dimension only contains a subset of available data that are considered for comparison. In our example (using two dimensions), we use either the names of called methods or Java types. Therefore, our multi-dimensional approach not only (1) supports the detection of clone-pairs with extreme pattern dissimilarity for the certain conditions which are discussed earlier but also (2) improves its scalability by several folds.

5. SeByte – A Clone Detection and Search Approach for Java bytecode

In what follows we introduce SeByte, our approach for Java bytecode clone detection and search. Figure 9 summarizes the overall approach and its heuristics. SeByte is based on our approach of *relaxation on code fingerprint* and the *multi-dimensional comparison* heuristics. As a result, when there are m dimensions and n similarity functions, $m \times n$ similarity values are calculated for each candidate pair. Each similarity value (Figure 9 - last column) shows the resemblance between the candidate pair using the similarity functions for a given dimension. This is opposed to common clone detection approaches, where a similarity value is only calculated for each candidate pair using a single data source (e.g., all transformed lines of code) and a single similarity calculation function (e.g., longest common subsequent). In our approach we summarize all similarity values using one of two decision making schema (clone search or detection) to derive a final result set. In what follows we discuss in more detail the dimensions, similarity calculation functions, and decision making schemas applied in SeByte.


A- Converting to text
B- Fingerprinting
C- Clone search and detection
Input
Java Bytecode in text format
Dimensions
Similarity Functions
Similarity Results
Java Bytecode Files
Java Type Fingerprints
{Player, Player, Player, Europe, List, Iterator, Iterator, Unit}
Java Type Repository (Dimension #1)
Method Call Fingerprints
{getEurope, initilizeHighSeas, getEurope, getEurope, getUniList, ...}
Method Call Repository (Dimension #2)
Instruction Fingerprints
{invokevirtual, ifnull, aload, invokevirtual, invokevirtual, invokevirtual, invokevirtual, invokeinterface, astore, aload, ... Checkcast}
Instruction Repository (Dimension #3)
Pattern matching (Function #1)
Content matching (Function #2)
Dimension #1 X Function #1
Dimension #1 X Function #2
...
Dimension #1 X Function #n
Dimension #2 X Function #1
...
Dimension #m X Function #n
Clone Detection
Clone Search
Filter
Sort
Extension Points
Dimension #m
...

```
674: invokevirtual #50    // Method Player.getEurope()
677: ifnull       852
680: aload         12
682: invokevirtual #51     // Method Player.initializeHighSeas()
684: invokevirtual #50     // Method Player.getEurope()
687: invokevirtual #50     // Method Player.getEurope()
690: invokevirtual #52     // Method Europe.getUnitList()
693: invokeinterface #70   // InterfaceMethod List.iterator()
698: astore        13
700: aload         13
702: invokeinterface #71   // InterfaceMethod Iterator.hasNext()
707: ifeq         52
710: aload       13
712: invokeinterface #72   // InterfaceMethod Iterator.next()
717: checkcast    #53       // class Unit
```

Figure 9. SeByte – our clone search and detection Java bytecode

5.1. Data extraction - dimensions

The plain text presentation of the bytecode content constitutes the input data of our approach (Figures 7 and 9 - A). Following our fingerprinting relaxation and the multi-dimensional heuristics, three dimensions are identified and created for bytecode content. These three dimensions which we use in our SeByte implementation are the instruction, the Java type and the name of method call dimension. It should be noted that depending on the application context, additional dimensions (e.g., literals) can be included or a subset of the defined dimensions can be selected to customize the search.

5.2. Similarity calculation functions

5.2.1. Basic content similarity

We adapted Jaccard similarity coefficient (Eqn. 1) as a function that calculates the content similarity between two sets. As part of the content similarity process, the objective is to calculate the semantic resemblance of two method blocks based on their contents (e.g., tokens) regardless of the order of their elements. The token matching itself is based on the syntactical presentation, with s_i being a set containing neither repeated elements nor ordering information.

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

5.2.2. Semantic-enabled content similarity

SeByte uses the semantic-enabled content similarity measure with different input data when semantic search is required as noted in Section 3. The motivation is to match further items from the given sets via the provided semantic network. A naïve implementation of function with linear complexity is achievable by using Jaccard (Eqn. 1) when the input is a set of the original items and also all of their ancestors (recursively).

Table 3. Examples for the seven similarity forms (excluding the identical case)

Target pattern														
A	A	B	C	D	E	E	F	F	F	G	H	**Functionality required for matching**		
Examples of the potential matches												***Sliding***	***ignoring the repetitions***	***handling the middle gap***
		B	C	D		E		F		G	H	X	X	
A	A	B	C				F	F	F	G	H			X
		B	C				F	F	F	G	H	X		X
A		B	C	D	E		F			G	H		X	
A							F			G	H		X	X
		B					F			G	H	X	X	X
				D	E	E	F	F	F			X		

5.2.3. Pattern resemblance

Similarity resemblance between two method blocks can be calculated not only based on shared items but also the ordering of the items. We consider in our research patterns to be similar, if the *relative order* of their tokens is identical, therefore allowing two fragments being matched to contain sliding, gaps, or repetition. We observed that gaps, sliding, or repetition can constitute a major source of dissimilarity between two similar but not identical patterns for bytecode. Gaps are referring to two patterns that share some tokens with same ordering, however, with one of the patterns missing some tokens. A repetition means that the gap is introduced because of consecutively reoccurring tokens. When sliding occurs, one of the patterns has some other tokens before the similar shared pattern starts. The combination of these three dissimilarity types can produce seven different types (excluding exact matches) of similarities (Table 3). Common to all of them is that the relative order of the shared items conforms to the target pattern. Regardless of their implementation approach, we require a function $\mathrm{P}(s_1, s_2)$ to determine the pattern resemblance. Note s_i is an ordered sequence which allows repeated elements. A function P determines if these blocks are sharing at least one known similarity form (Table 3).

5.3. Decision making schemas

While providing different similarity functions to determine code similarities among code fragments for each dimension is an essential step, further interpretation or classification of these individual results is required. This determines whether two methods can be considered a clone candidate or not. In what follows we introduce two schemata (clone detection and clone search), which automated this decision making process.

5.3.1. Clone detection

For the clone detection problem, we need a decision making schema that answers whether two method blocks constitute a candidate clone-pair or not. That is the output of the function is boolean. As a result, all method blocks which meet the clone requirements specified in the decision making schema are added to the final clone result set, while the remaining candidates are discarded (as potential true negatives). The reduction of false positives is essential in order to increase the usability and acceptance of a clone detection approach, which is pointed out by Juergens et al. [34]. Therefore, we considered pattern resemblance functions as a necessary condition to be met by each clone pair. This is a conservative approach, since for many clone detection applications precision is crucial.

SeByte, which can be a considered a concrete instantiation of our general approach, must deal with more than one similarity indicator. We adapted the same approach as of metric-based [1, 4, 11] clone detection tools. When there are m dimensions and n similarity functions deployed as part of the SeByte, $m \times n$ similarity values are calculated for each candidate pair. Each similarity value shows the similarity resemblance between the candidate pair using the similarity definition (i.e., function) from the chosen perspective (i.e., dimension). We consider each as a metric, which is either numerical (e.g., content similarity functions) or boolean (e.g., pattern resemblance functions). Finally, our decision making schema converts all non-boolean metrics to boolean, and makes its final decision based on the conjunction of all the indicators. For the conversion of numerical metrics to boolean, the schema requires a set of thresholds. For example, if there are two dimensions such as method call fingerprints and type fingerprints, to convert the numerical output of Jaccard coefficient two thresholds are required. Each threshold determines the acceptable boundary of the output of the Jaccard for a corresponding dimension. The inherent computation complexity of this decision making schema is $O(l * n^2)$, with no optimization, where l is the largest method size. However, l can be considered as a constant therefore our actual complexity[d] is $O(n^2)$.

5.3.2. Clone search

A major difference between clone detection and clone search is the characteristics of the output data and its presentation. In contrast to the code detection problem, the result of a clone search is expected (ideally) to be a "ranked result set" where the order of items reflects the "degree of relevance" between the query and the matched item. Moreover, the result set includes all matching fragments therefore no item will be discarded due to its low similarity degree to the query. Although most of the hits with low similarity degree are probably the false positives, we are expected them to be placed toward the end of the result set. This is different from clone detection problem when it is expected from the model to discard all the suspicious matches.

In order to facilitate such ranking, a decision making schema is required that produces a single-value numerical similarity indicator for each two method blocks. For each query, its result set will be sorted and presented based on the values generated by the schema. Therefore for the $m \times n$ similarity values (Figure 9 - the similarity results column), a summation is required to generate the single-value output. Each similarity indicator requires a coefficient as part of the configuration which determines the importance of the corresponding dimension and similarity function.

In our SeByte implementation, we considered three dimensions, with the resulting schema being represented through a triple (I, M, T), where I indicates the weight of the instruction dimension, M the weight of the method and T the weight of the type dimensions. The weight indicator can be either "*leveraged*" or "*regular*". For a leveraged dimension, a larger coefficient is required. Our schema does not restrict the number of members (i.e., dimensions) that belong to a particular group (leveraged or regular), even allowing all dimensions belong to the same group (and therefore have an equal weight). We use $+$ to denote that a dimension is leveraged and $-$ to indicate that a dimension has a regular weight. For example $-+-$ indicates that the instruction and type dimensions have a regular weight, while the method dimension will be leveraged by using a larger coefficient.

6. The Clone Detection Study

We conducted a study on the performance of SeByte for clone detection application, to compare our solution with results obtained from alternative approaches. For this preliminary study we analyzed datasets from different application domains and project sizes (Table 4). These datasets were manually extracted and checked for completeness (i.e., source code versus corresponding bytecode contents). 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 the overall content remains similar.

Table 4. Datasets for the clone detection study

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

[d] Using inverted index as an implementation level optimization technique, it is possible to achieve $O(n)$, when $p \ll n$. Note that p denotes the maximum number of possible pairs belong to a single clone class.

6.1. Finding proper thresholds for clone detection

For our clone detection study we selected two dimensions corresponding to method call and type fingerprints, and two similarity functions, the Jaccard-based content similarity and pattern resemblance. For the decision making schema, we used the clone detection schema, which requires two thresholds for our similarity functions with non-boolean output. First, we had to determine applicable thresholds for our (1) ω (Jaccard threshold for method similarity), and (2) φ (Jaccard threshold for Java type similarity). The objective of this calibration process was to determine values for ω and φ through an empirical analysis for each schema, such as that the overall 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 pairings (including both actually cloned and non-cloned pairs) using EIRC's binary representation. The similarities of these pairings were then verified against EIRC's source code level similarities.

Step 2. In order to determine the optimum combination values for ω and φ, we calculated the F-measure for all observed combinations based on $1 \leq \varphi \leq 90$ and $1 \leq \omega \leq 90$ with our window size being equal to 1 Figure 10 shows the F-measure values.

From this experiment, we were able to identify the combination of ω and φ values (Figure 10) for which their corresponding F-measure showed the highest value and therefore the overall highest combined precision and recall. The observed optimum combination for the dimension thresholds are $\{\omega = 53,\ \varphi = 16\}$. That is based on a 2-dimensional configuration. If a paring's content similarities are above 53% for method call fingerprints and 16% for type fingerprints, the pairing constitutes an actual clone-pair view with high confidence. We also observed the best achievable recall and precision by our approach are 92% and 79% respectively.

Step 3. As part of this validation phase, we further evaluated the performance of our identified thresholds against randomly selected clone-pairs obtained from the other three datasets (Table 4). We manually evaluated 507 clone pairs from the other datasets of our study and we found that the same thresholds are also outperformed the other thresholds (e.g., 79% precision).

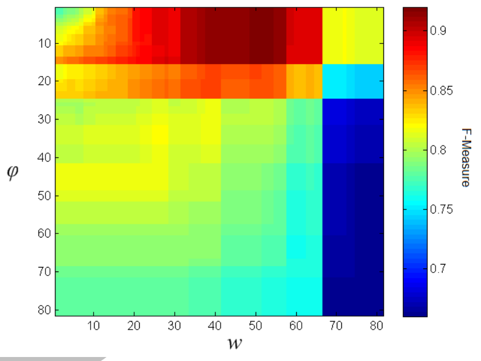


Figure 10. The F-measure values using all combinations of the two thresholds

Table 5. Summary of clone detection tools and comparison method details

Tool	Input Data	Input Format	Comparison Method	Tool Granularity	Tool Original Purpose
NiCad [17]	Table 3	Source code	Automatic	Method-level	Near-miss source code detection
Merlo dataset (From Bellon Dataset [1])	Bellon oracle	Source code	Automatic	Method-level	Metric-based method-level Type-3 source code clone detection
Scorpio [18]	Table 3	Source code	Manual (sampling)	Line-level (token)	Gapped clones. Source code PDG
JCD [15]	Table 3	Binary	Manual (sampling)	Line-level (pcode)	Type-3 binary clone detection
SimCad [6]	Table 3	Binary	Automatic	Method-level	Near-miss source code detection

6.2. Observations

As part of this evaluation step, we assessed the performance of SeByte in terms of (dis)agreements with other alternative approaches (Table 5). For example, we are interested to observe the differences between SeByte, which is a bytecode method-level clone detection approach, with a source code clone detection approach. The candidate approaches are using different input sources (source code vs. binary) and operating at different levels of granularity (method vs. line). NiCad, a near-miss clone detector and Scorpio, a semantic clone detector were both used at the source code level, while JCD, a Type-3 on Java binary clone detector, and SimCad, a near-miss clone detector were applied on bytecode level. We also adapted for comparison reason Merlo clone result as a metric-based approach from Bellon et al.'s study [1].

6.2.1. Automatic comparison

Since SeByte detects clone-pairs at *method-level*, automated result *comparison* could be performed only with tools working at the same granularity. Automated result comparison therefore could be applied only for experiments involving NiCad (using the recommended configuration for Type-3), Merlo's clone set from Bellon et al. [1], and SimCad. The agreements/disagreements with NiCad including a detailed report are shown in Table 6. As expected the agreement percentage is quite low, due to the fact that these comparison tools are designed to detect different types of clones. Moreover, this observation complies with earlier studies both by Selim et al. [14] and Davis and Godfrey [15], that these disagreements occur due to differences between binary and source code. We also used SimCad, which has been originally designed for source code clone detection on binary code following the same approach by Selim et al. and Baker and Manber [16]. Again, as expected the agreement between SeByte and SimCad was low with less than 20% on average. Finally, we compared SeByte with Merlo's clone detection tool (CLAN) [1]. The total observed agreement for Types 1 and 2 clones were about 18%, while for Type-3 a negligible agreement was observed.

Table 6. SeByte and NiCad result Comparison

	# Clone pairs		**# Clone classes**		**Agreement (~%)**
	SeByte	NiCad	SeByte	NiCad	
EIRC	198	63	24	17	40%
FreeCol (server)	708	43	46	19	70%
Freecol (full)	1593	1339	149	305	60%
Apache (DB)	26955	15378	190	297	30%

6.2.2. Manual comparison

We also manually compared SeByte with JCD on bytecode and Scorpio on source code content. We used JCD 1.0.10 with the configuration recommended by its authors. Since Scorpio requires more memory than regular clone detection tools, we executed Scorpio on a hardware with 24 GB RAM. As noted earlier, we were unable to automate the comparison process for JCD and Scorpio since both detect clones at line level granularity rather than the method-level granularity provided by SeByte. We therefore manually verified whether results obtained from SeByte (cloned methods) are also detected by the line-level clone detection tools. For JCD the agreement ratio was 40%, whereas for Scorpio no considerable agreement was observed.

6.3. The clone detection study summary

From our experimental evaluation, we were able to observe that SeByte can detect clones that are missed by other source code or bytecode based tools and vice versa. As a result, the clones reported by our approach can be considered to be complementary to existing clone detection tools. The primary reason for the low agreement between SeByte and the other detection tools are due to several factors: difference in the clone detection approaches, the objectives (type of clones), and the input data. We also evaluated SeBytes scalability for clone detection on a large dataset, which was populated with six versions of the Eclipse IDE, totaling 500K compiled Java classes and 73M LOC bytecode. The experiment completed[e] successfully (excluding the time required for result serialization to the external disk) in 37 hours processing 870K methods and with each method having at least 5 tokens for each dimension.

7. SeByte Clone Search Performance Evaluation

Using our clone search schema (Section 5.3.2), we conducted a study for which we used SeByte's core function to implement a scalable clone search approach for Java bytecode. The objective of this study was to evaluate the performance of our solution for bytecode clone search. For the bytecode clone search, the input data for each experiment is a given query (code fragment) and the corpus. The output is fragments that are similar to the given input query.

A major difference between traditional clone detection and clone search is the characteristics of the output data and how to evaluate the quality of the output results. In contrast to traditional code detection, the result of a clone search is expected to be a "ranked result set" where the order of items represents the "degree of relevance" between item/query tuple (q, h_i), with i denoting the position of the hit in the ordered list. Therefore, evaluation of such systems is different [35] since not only the quality of the returned results in terms of correctness and completeness (e.g., precision and recall) but also their ranking has to be evaluated [37].

We use the following example to illustrate the necessity for using ranking positions when evaluating the quality of clone search algorithms. Suppose for a given code fragment, q which constitutes the input query, there are four cloned code fragments (a, b, c, d) in the dataset. Suppose that in our example, the expected ideal ranked result set is $\langle a, b, c, d \rangle$ where the search algorithm returns $\langle a, b, \boldsymbol{d}, c \rangle$. While returning all the expected hits (precision=100% and recall=100%), the search algorithm failed in this case to return the ideal expected answer, by reporting an actual less relevant result d prior to the more relevant one c. Therefore, additional measures (except precision and recall) are required to evaluate all aspects of the ranked result set. We have adapted three measures traditionally used for evaluation in the Information Retrieval [35] to evaluate the quality of our ranked result sets.

7.1. The Eclipse dataset

A key requirement for our study was not only to have a sufficient large dataset, but also a dataset that contains (1) a few highly similar clones, (2) several relatively similar clones, and (3) a large number of irrelevant fragments. A dataset which meets these requirements allows us to evaluate our search approach in situations when for each query the number of irrelevant fragments (i.e., noises) is considerably larger than the number of actual clones, making the ranking a challenging task. For the study, we have therefore created a dataset consisting of the bytecode (including all binary dependencies) of the latest six major versions (2007 – 2012) of Eclipse IDE (Table 7).

Table 7. The Eclipse Dataset Overview

Feature	**Value**
Total Jar (library) files	3,900
#File (Java class)	482,768
#LOC (bytecode level)	73 M
Total method	3,898,475
Total significant method (min 2 information token or more)	~1,780,000
Total significant method (min 5 information token or more)	~780,000

[e] Note, since our latest SeByte implementation does not require main memory for indices storage and the complexity of SeByte remains linear therefore it can scale up to the size of available external disk while providing reasonable performance.

7.2. Results

We applied our decision making schema for clone search problems (Section 5.3.2), which is represented by a triple (I, M, T), where I indicates the weight of the instruction dimension, M the weight of the method and T the weight of the type dimensions. We use $+$ to denote that a dimension is leveraged and $-$ to indicate that a dimension has a regular weight. For example $-+-$ indicates that the instruction and type dimensions have a regular weight, while the method dimension will be leveraged. In this study, the performance of the seven possible combinations are reported and compared. The overall pre-processing time for the study is reported in Table 8.

Table 8. Pre-processing time report

Step	Time (seconds)
Jar file and bytecode extraction (unzipping + disassembling)	3422
Crawling[f]	0.802738268
Fact processing	267
Index construction (+ fact processing)	755

7.2.1. First False Positive measure

Introduction. A widely used source of information for evaluating search engines in the Information Retrieval (IR) domain are the top displayed items (a.k.a., hits) in the result set. Studies in the IR domain have shown that end-users tend to browse only these top items. Therefore, for search engines it is essential to have as many true positive as possible in the high-ranked hits (e.g., top-10). It is impossible to always provide 100% precision (i.e., no false positive) within the top-k hits. Based on these facts, the place of the first false positive in the result list is used as a fair measure for the performance evaluation of search engines. For example R1 and R2 are two ordered result sets, with both containing 10 hits, R1 = $\langle h_1, \boldsymbol{fp}, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9\rangle$ and R2 = $\langle h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, \boldsymbol{fp}, h_9 \rangle$. Nine out of ten hits are true positives and one is the false positive (fp). While the precision for both results set is 90% (9 out of 10 hits are correct), the user satisfaction for R2 is considered higher. Note that the first false positive occurs later in the ranked result set R2 (position 9 vs. 2 in result set R1).

Result. For our evaluation we measure the position of the first false positive in a set of 20 randomly selected queries tested on each of the 7 possible search schemata. We have evaluated 140 result sets containing 30 clone pairs each (4200 clone-pairs/hits in total). Figure 11 summarizes the results from our manual evaluation in terms of “first false positive” position within the top 30 hits.

Result Interpretation. We believe this measure represents one of the stricter measures when evaluating the performance of the clone search system, especially in our case with a noisy corpus. For example, in our study, on average only 6 out of ~1.7 million code fragments in our corpus were actually highly relevant fragments, whereas the remaining ones are mostly non-relevant. From a clone search viewpoint our search approach had to deal with two major challenges: first, being able to detect the few relevant fragments, and second, to assign these true positive hits a higher priority than the false positives in the result sets.

Our study (Figure 11) shows, the performance varies considerably in terms of placing the first false positive in the ordered result set for the different search schemata. Second, there are few schemas for which their performance is stable. For example, the $--+$ search schema places the first false positive within its top 3 answers for 12 out of 20 queries which is a poor performance. In contrast, the best performance was achieved with the $-+-$ search schema, which leverages the method dimension over the other two dimensions. This schema returns the first false positive at 6th position in 75% cases.

Furthermore, our manual validation also revealed that the actual performance of each schema is highly dependent on the actual search query. For example, some schemata perform better for queries performed on small method fragment (e.g., 2 lines code such as setters and getters in Java) while others performed better for larger methods. The

[f] Retrieving the list of the binary files from the local file system

$- + -$ search schema outperformed in most cases the remaining schemata. The results from our clone search study also support our earlier observations for clone detection (Section 6.1) that leveraging the method dimension in clone detection and search can improve the precision of the results.

Weakness. Given the inherent nature of the measure, which is highly dependent on the data and query characteristics, the applicability of the measure is limited. For example if a corpus has only X actual true positives for a given query, the best achievable result using this measure is $X + 1$. This issue affects the applicability of this measure specifically where the value of X is rather small (e.g., 4). Moreover, this measure's results cannot be averaged across queries for direct result interpretation.

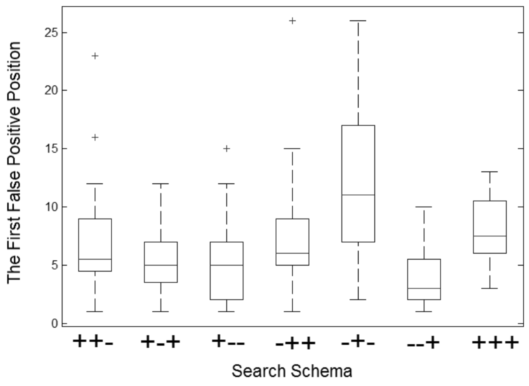


Figure 11. Summary of the First False Positive measure study

7.2.2. "Precision at 10" measure

Introduction. Precision at 10 (Eqn. 2) is an instance of the general measure, which is known as Precision at K where K can be any positive number. However, 10, 20, and 30 (window size 10 [35]) are some of the popular Ks. The K is derived from the general rule of thumb in designing Graphical User Interface of search engines where the first page usually shows only top 10 hits. The measure is established based on the general fact that quality of the search engine from end-users' point of view is highly related with the quality of the result set shown in the first page since the users rarely browse to the second page [35].

$$Precision_k = \frac{tp_k}{tp_k+fp_k} \qquad where\ tp\ and\ fp\ are\ limited\ to\ the\ top\ K\ hits \qquad (2)$$

Weakness. This measure seems to be a perfect candidate for evaluating a clone search system especially when one considers that fine-level granularity and strict evaluation (e.g., "first false negative" measure) are not required. However, its major limitation is its query dependency. For example, in order to provide a fair evaluation using "Precision at 10" measure, at least 10 actual cloned fragments must exist in the corpus for all executed queries. Similar to the previous measure, the measure's results cannot be averaged across queries for direct interpretation.

Results. In order to provide a fair analysis, we split our candidate queries into two subsets: (1) queries with less than 10 actual cloned fragments in the whole corpus, and (2) queries with more than 10 actual cloned fragments. Note that due to the inherent nature (6 releases of Eclipse from 2006 to 2012) of our dataset in which most code fragments have 5 other cloned fragments which make it a 6-pair clone class, we studied "Precision at 5" for the first set of queries. For the second subset we actually were able to use "Precision at 10". We manually evaluated the top K hits of 40 queries for all seven schemata (2100 fragments in total) to calculate the precision (Figures 12 and 13).

Result Interpretation. Figures 12 and 13 summarize the result of our manual evaluation. The evaluation showed that for both sets, the $- + -$ schema provides the best overall result with 90% precision in the worst case (excluding the outliers which are tagged). When one considers the outliers the precision drops for $- + -$ schema in one case to 40%. Figure 14 provides a more detailed analysis of the different schemata for each of the queries. The added curves help to observe and compare the potential fluctuations. Figures 12, 13 and 14 show that $- + -$ schema achieves the best precision. However, there are some exceptions, e.g., query #15. For example, the $+ - -$ schema performs in one single instance better than $- + -$ but the precision of $+ - -$ fluctuates between 100% and 0%.

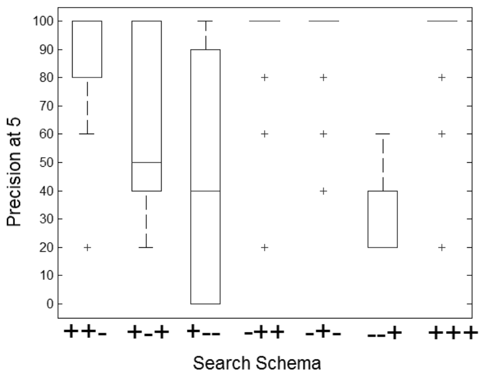


Figure 12. Summary of the Precision at 5 measure study

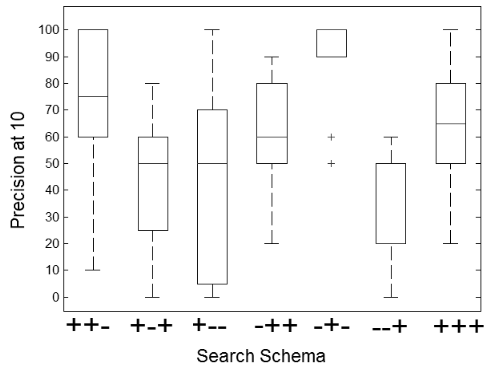


Figure 13. Summary of the Precision at 10 measure study

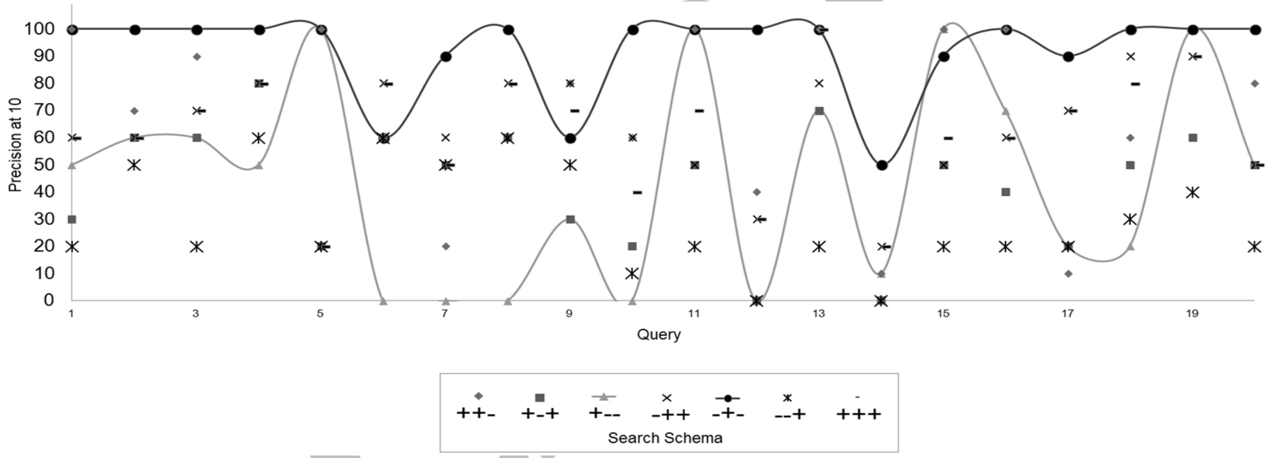


Figure 14. Details of the Precision at 10 measure study

7.2.3. Normalized Discounted Cumulative Gain measure

Introduction. Normalized Discounted Cumulative Gain (NDCG) measures the quality of the search engine in terms of how well it assigns higher ranks to high quality answers. This measure takes into consideration not only the relevance of hits to the query but also their order. Therefore, it is possible to compare the search result set with the ideal result set (the oracle). The output is a number, which can be used to compare different schemata. Note that, this measure requires to have a manually created result set (for each query) which is a sorted list of all possible answers based on their relevance to the query. Moreover, each answer in the ideal result set (the oracle) must be assigned a relevance score which presents its similarity degree to the query. This ideal result set represents the best achievable result set and order, regardless of local search configurations, search algorithm and search schema.

Details. DCG (Eqn. 3) calculates the discounted cumulative gain achieved using the given search schema for query q when compared to the oracle with its manually assigned relevance scores for the top n hits. The *similarity score* for each hit, denoted by $r(q,i)$, is achievable from the benchmark. The range of DCG depends on the query and available data within the corpus (e.g., could be any positive number $x >= 0$), therefore it is not possible to compare the DCG of different queries with each other since one might have higher number of positive hits due to data characteristics. We use NDCG in order to overcome this issue and to be able to summarize the results. To calculate the NDCG (Eqn. 4), we need to calculate the *Ideal DCG* (IDCG). $IDCG$ returns the ideal (highest achievable) DCG using the given relevance score set (from the oracle). That is DCG of the best system is the same as the value of the IDCG. Finally using DCG and IDCG, we can now calculate the NDCG.

Since the output of the NDCG function is normalized, it can be used for both (1) query comparison and (2) as an averaged measure for the overall performance of the search engine. The ability to average the measure results allows us to provide a concrete single output value. The averaged value can be used to compare the configurations (e.g., schema). The maximum value for the NDCG function is 1.0 for a result set that matches exactly the one from the oracle, and the minimum of 0.0 if there is no match at all.

$$DCG\ (q,n) = r(q,1) + \sum_{i=2}^{n} \frac{r(q,i)}{log_2(i)} \qquad (3)$$

$$NDCG(q,n) = \frac{DCG(q,n)}{IDCG(q,n)} \qquad (4)$$

Weakness. This measure provides a fine grained evaluation of the quality and ordering of a result set. Therefore, it is a good indicator to compare different search algorithms and search schemata. However, it is only applicable, when fine grained ordering is important in a given application context. Otherwise, measures such as Precision at K are preferred. Applying NDCG is expensive since all possible answers should be manually evaluated for *similarity score* assignment (e.g., identical, highly similar, similar, and irrelevant). Nevertheless, NDCG is one of the state of the art search engine measures used in the IR [35].

Result. For the study, we selected 20 queries and their clone results, with all queries providing at least 30 but less than 100 matches. To create the oracle for each query (required by NDCG), we manually evaluated a total of 1481 candidate clone-pairs and assigned scores between 0 and 3 (i.e., the *similarity score*). We used 0 to indicate totally irrelevant pairs (100% False Positive). 1, 2, and 3 are used for clone-pairs with some similarity (true positive clone-pair). Similarity degree increases as the score value arrives at 3. This manually tagged set and the related 20 queries constitute our oracle in this study. Finally, 10,367 hits are retrieved (from executing 20 queries over seven search schemata), to automatically calculate NDCG using the tagged oracle. Figure 15 presents the average NDCG value achieved by each search schema including the NDCG value for all query-search schema pairs. Note that there is no ordering between queries.

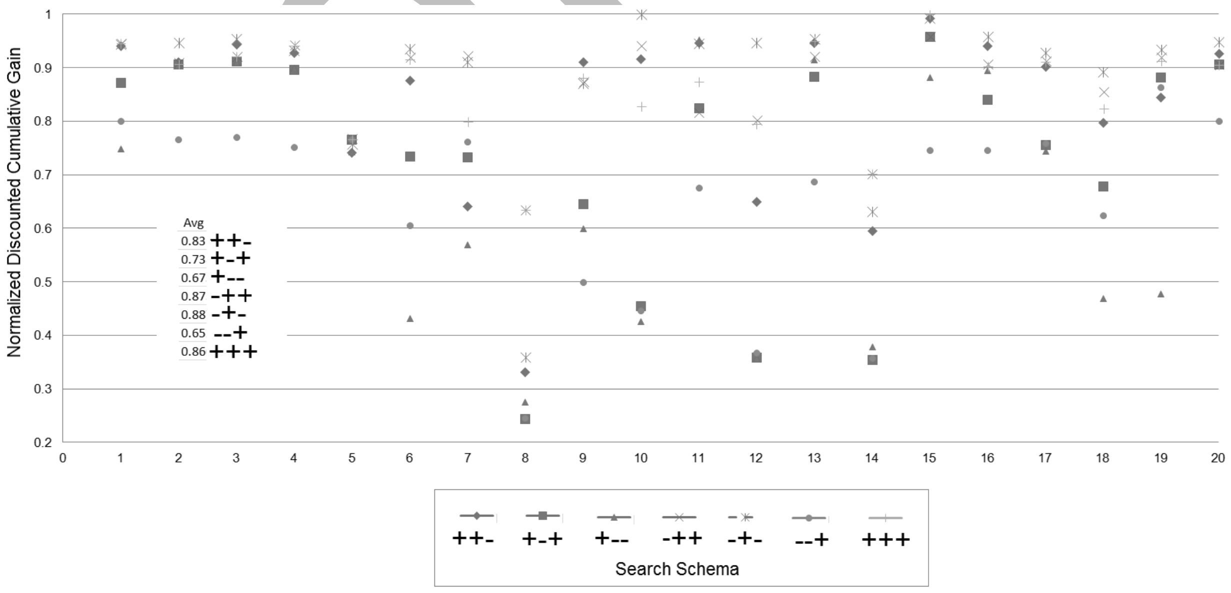


Figure 15. Details of the NDCG measure study presenting the averaged behavior of schemata

7.2.4. Summary and discussion

The $-+-$ search schema outperforms the other schemata by achieving on average a 0.88 NDCG (Figure 15). Considering the result of all measures altogether (i.e., First False Positive, Precision at K and NDCG), $-++-$ was the overall most reliable search schema. This study shows that our approach can be used for clone search applications on bytecode where ranking is required. We do believe that our results can be generalized to some extent, since our corpus and query set were significant in size and furthermore, we did not rely on a single measure but instead we selected measures that are evaluating different aspects of the result set.

8. Related Work

Binary code clone detection has not been a major research focus in the clone detection community. Regardless of the application, the selection of the detection algorithm depends on the type of binary content. There are two major families of binaries: (1) native machine code, and (2) intermediate language-based binaries (e.g., bytecode). Each category of binaries has its own unique characteristics which require different clone detection algorithms.

For clone detection on Java bytecode diverse approaches are introduced. Baker and Manber [16] used a combination of three comparison based approaches such as Diff. The JCD project [15] developed by Davis and Godfrey uses a combination of hill climbing and greedy algorithms to find the maximum coverage (including a pretty-printing tool [24]) for clone detection. There is also a proposal to use process algebra on bytecode [25]. Selim et al. [14] converted bytecode to the Jimple format [26] and used third-party tools (originally designed for source code) on the Jimple content.

Recently, license violations and malware detection have been promoted as applications for clone detection on binary content [27, 28, 29]. A major factor that supports adaptation of the binary content is often limited availability of source code. Both applications can be addressed by clone search since the goal is to find similar content (query context) in a typically large corpus. Hemel et al. [29] explored some generic similarity heuristics for license violation detection using their Binary Analysis Tool (BAT). In their approach they use string literals, extracted from the target binary, in the central database of literals as part of their first search heuristic. Note that the central literal database can be built using literals extracted from both source code and binary. That is, the assumption in their research is that the source code of the target entity is not available. Compression ratio as a similarity metric is their second heuristic which has been investigated previously in other similarity search domains such malware detection [29]. Computation of the delta between the target binary and the suspect binary (from the central repository) constitute their last heuristics. For binary content (i.e., native machine code), Sæbjørnsen et al. [28] proposed a solution which is more strict than Hemel et al [29]. Sæbjørnsen et al.'s approach is founded on common source code clone detection techniques (where the content is indexed based on pattern similarity). They apply normalization (similar to Baker et al. [16]) including token categorization. For binary content, they replace possible values of operands (e.g., register name, memory address, and constants) with their category name (i.e., memory, register, value). Finally, to retrieve the similar fragments, they model the normalized data using feature vectors. Chaki et al. [29] explored the applicability of classification techniques on the binary to detect similar binaries which come from (1) similar source code and (2) the same compiler. Provenance-similarity is defined for two fragments when both conditions hold. Chaki et al. have argued that holding these two conditions seems reasonable for their specific context of applications (malware and virus detection).

Another specific challenge in some domains such as .NET is detecting clones across multi-languages. To avoid dealing with several high-level languages, the intermediate language (i.e., form of binary content) has been adapted as the sole source of information in recent studies [30, 31, 21]. Kraft et al. [30] used the graph presentation created using the binary to detect cloning between languages. In our earlier studies on .NET [21], we also addressed the same problem by creating a filter set for noise reduction to improve the feasibility of such approaches. Recently, diverse approaches are proposed for semantic clone detection [3, 18, 23], such as: (1) a formal method-based approach for embedded systems [23], and (2) clustering of entities in different granularities to achieve scalability [3]. These approaches are proposed to improve the limitations of traditional clone detection approaches (e.g., [4, 7, 11]).

Selim et al. [14] reported up to 49% and 78% agreement between clones from bytecode and source code. This observation was our major motivation for this research to provide a clone detection approach which can be used as a complementary approach for clone detection and management systems. Our objective is to recover clones missed by other state of the art source code-based clone detection techniques. Our evaluation of SeByte not only confirms this claim, but we were also able to show that SeByte can detect more diverse results. Our comparison not only shows the usefulness of using bytecode for clone detection but also highlights the strength of our heuristics for clone detection. Furthermore, we showed how SeByte can be extended to provide a semantic clone search applications for bytecode which makes our approach to be the first clone search engine for bytecode that supports fine granularity ranking.

9. Conclusion and Future Work

In this research, we introduce SeByte, a bytecode clone detection and search model that applies semantic-enabled token matching. It is established based on the idea of relaxation on the code fingerprints. This approach separates the input content based on the token types. The idea is that the data belong to each token type or dimension represent the content from a specific point of view. Following this approach, SeByte compares dimensions separately and independently which is known as multi-dimensional comparison in our research. As the similarity search function, we proposed adaptation of Jaccard similarity coefficient for the multi-dimensional comparison heuristic. Moreover, we discuss how this comparison approach can be extended for semantic search for bytecode content. Our experimental results and comparative evaluations with other state of the art approaches show that SeByte can detect semantic clones that are missed by the other approaches. We then further exploit SeByte model to build a scalable bytecode clone search engine. This extension meets the requirements of a classical search engine including the ranking of result sets. Our evaluation with a large dataset of 500,000 compiled Java classes, which we extracted from the six most recent versions of the Eclipse IDE, showed that our SeByte search is not only scalable but also capable of providing a reliable ranking of the result sets. We also successfully tested the scalability of the SeByte for clone detection application using the Eclipse dataset. This work is based on our earlier work [33] with extensions towards semantic-enabled token matching for clone search problem. As future work, we plan to release the SeByte as an online search engine which provides bytecode clone search functionality over available bytecode on the Internet for the research community.

Acknowledgment

The authors would like to thank the anonymous reviewers and editors for their valuable comments that helped improve the original manuscript.

References

[1] 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.

[2] 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.

[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] 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.

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

[6] 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.

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

[8] M. Lee, J. Roh, S. Hwang, and S. Kim, "Instant code clone search," Proceedings of the eighteenth ACM FSE, pp. 167–176, 2010.

[9] M. R. Quillan, "Word concepts: A theory and simulation of some basic capabilities," Behavioral Science, 12, 1967.

[10] R. Guha, R. McCool, and E. Miller, "Semantic Search," 12th international conference on World Wide Web, 2003, pp. 700-709.

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

[12] 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.

[13] I. Keivanloo, C. Forbes, A. Hmood, M. Erfani, C. Neal, G. Peristerakis, J. Rilling, "A Linked Data Platform for Mining Software Repositories," 9th Working Conference on Mining Software Repositories (MSR), 2012.

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

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

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

[17] 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.

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

[19] B. Hummel and E. Juergens, "Index-based code clone detection: incremental, distributed, scalable," 26th IEEE international conference on Software Maintenance (ICSM), 2010, pp. 1–9.

[20] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," Software Engineering, IEEE Transactions on, vol. 28, no. 7, pp. 654–670, 2002.

[21] Farouq Al-Omari, Iman Keivanloo, Chanchal K. Roy and Juergen Rilling, "Detecting Clones across Microsoft .NET Programming Languages," 19th Working Conference on Reverse Engineering (WCRE), 2012.

[22] I. Keivanloo, C. K. Roy, J. Rilling, " SeByte: A Semantic Clone Detection Tool for Intermediate Languages ," 20th IEEE International Conference on Program Comprehension (ICPC), Tool Demo. Track, 2012.

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

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

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

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

[27] S. Chaki, C. Cohen, and A. Gurfinkel, "Supervised learning for provenance-similarity of binaries," Proceedings of the 17th ACM SIGKDD, 2011, pp. 15–23.

[28] A. Sæbjørnsen and J. Willcock, "Detecting code clones in binary executables," ISSTA, 2009, pp. 117–127.

[29] A. Hemel, K. Kalleberg, and R. Vermaas, "Finding software license violations through binary code clone detection," International Working Conference on Mining Software Repositories, 2011, pp. 63–72.

[30] N. Kraft, B. Bonds, and R. Smith, "Cross-language clone detection," 20th International Conference on Software Engineering and Knowledge Engineering, SEKE, 2008.

[31] V. Juričić, "Detecting source code similarity using low-level languages," Technology Interfaces (ITI), 2011, pp. 597–602.

[32] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," 6th Symposium on Operating Systems Design & Implementation, 2004, pp. 137-149.

[33] I. Keivanloo, C. K. Roy, J. Rilling, "Java Bytecode Clone Detection via Relaxation on Code Fingerprint and Semantic Web Reasoning," 6th International Workshop on Software Clones (IWSC), 2012.

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

[35] C.D. Manning, P. Raghavan, and H. Schütze, "Introduction to information retrieval," Cambridge University Press, 2008.

[36] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," Bulletin de la Societe Vaudoise des Sciences Naturelles, 37:547-579, 1901.

[37] A. Walenstein and A. Lakhotia, "Clone detector evaluation can be improved: ideas from information retrieval," 2nd International Workshop on Detection of Software Clones (IWDSC'03), 2003.