EncCD: A Framework for Efficient Detection of Code Clones

Minhaj Khan
Department of Computer Science, Bahauddin Zakariya University, Pakistan

Abstract: *Code clones represent similar snippets of code written for an application. The detection of code clones is essential for maintenance of a software as modification to multiple snippets with a similar bug becomes cumbersome for a large software. The clone detection techniques perform conventional parsing before final match detection. An inefficient parsing mechanism however deteriorates performance of the overall clone detection mechanism. In this paper, we propose a framework called Encoded Clone Detector (EncCD), which is based on encoded pipeline processing for efficiently detecting clones. The proposed framework makes use of efficient labelled encoding followed by tokenization and match detection. The experimentation performed on the Intel Core i7 and Intel Xeon processor based systems shows that the proposed EncCD framework outperforms the widely used JCCD and CCFinder frameworks by producing a significant performance improvement.*

Keywords: *Clone detection, Software Engineering, Software Maintenance, Optimization, Speedup.*

Received February 5, 2017; accepted September 30, 2018

1. Introduction

Software maintenance requires a careful traversal of all the snippets of code which may be segregated. For fixing a bug that is common in different snippets of code, it is inevitable to make modification at all locations of the erroneous code. This activity becomes cumbersome for a large software having multiple instances of duplicate code. Consequently, the evolution and maintenance of a large software having a large number of code clones becomes a challenging issue. Currently, the open source inter-project clones [19, 21] are also being detected to develop corpora which may subsequently be used to minimize development effort.

Various clone detection techniques with diverse levels of automation have been proposed in the literature. The diversity of the code clone detection techniques even arises from the fact that the languages, parameters and the benchmarks for evaluation are yet to be standardized [10].

In general, there are four types of code clones: type-1, type-2, type-3 and type-4 [18]. The type-1 clones represent snippets of code which are exactly similar except minor variations of whitespaces and comments. The type-2 clones represent snippets of code with similar syntactical structure with the exception of variations in identifiers, literals and data types etc. The type-3 clones add further possible variation of addition and removal of statements to the exceptions of type-2 clones. Similarly, the type-4 clones represent snippets performing similar computation with variation in syntax.

The clone detection strategies incorporate the mainsteps of pre-processing, code transformation, matching and aggregation. The pre-processing phase is used to eliminate irrelevant code, for instance, the embedded language code or initialization code which might otherwise produce false positives. The transformation phase produces an intermediate form which is subsequently used for matching. The existing strategies usually produce dependence graphs, parse trees or token sequences which are then normalized for elimination of some elements such as whitespaces, formatting or comments etc., Using the output of the previous phase, the matching phase compares different units of the code. It produces a list of matches which are then represented in the form of the source code coordinates. The code may then by further analyzed for removal of false positives through a manual or some heuristic based approach.

The code clone detection techniques are greatly dependent on the transformation output which impacts the execution performance of the code clone detection technique. For instance, the efficient parse tree approach requires the entire code to be represented in the form of tree nodes prior to matching. However, as the code size becomes large, the parsing techniques suffer from performance degradation due to architectural constraints such as the limited size of cache memory.

In this paper, we propose a framework called Encoded Clone Detector (EncCD), which is aimed at improving the performance of the code clone detection techniques. The proposed framework uses an efficient encoding mechanism to store statement level constructs with reduced code size. Due to reduction in size of the code, the parsing and matching phases become efficient which subsequently improve the overall performance of the code clone detection mechanism. We perform experimentation on a wide collection of open source software for evaluating the performance obtained through the EncCD framework and compare it with other well-known code clone detection frameworks.

The rest of the paper is organized as follows. Section 2 discusses the related work in the context of the code clone detection techniques. The architecture and implementation details of the proposed EncCD framework are described in section 3. The experimental setup and results are given in section 4 before the conclusion and future work which are described in section 5.

2. Related Work

The techniques proposed for clone detection range from simple text based syntactic comparisons to the complex semantic comparisons. A categorization of clone detection techniques based on text, trees, tokens, metrics and graphs is given by Roy *et al.* [18].

The text based techniques rely on performing substring comparisons using fingerprints or hashes. The technique by Johnson [13] initially applies hashing on a snippet of code, followed by a sliding window based comparison to search for the lines of code having the similar hash codes. A similar technique by Smith *et al.* [20] computes fingerprints by finding all sequences of tokens of a particular length, also called n-grams. Another approach using line-based hashes for finding similar code is given by Ducasse *et al.* [7]. Their technique uses dot plots for visualization of clone detection. A single dot is used to represent a similarity of two lines based on hash values. A pattern matcher is then run on the dot plot to automate detection of clones of different types.

The token based approaches use sequences of small substrings called tokens similar to those produced during lexical analysis. The sequences of tokens are then matched to determine the clones within the code. The token based clone detection approach proposed by Baker [1] uses two types of tokens. For tokens such as identifiers or literals, the location in the code is determined, whereas, for other tokens, a hashing function is applied. A suffix tree is then used to represent the sequences resulting from the previous step. In the suffix tree, common prefixes are used to indicate clones and are represented by shared edges in the tree. Similarly, a widely used framework called CCFinder [14] uses the token based approach for detecting clones. It supports clone detection for different languages and works by incorporating the conventional suffix trees.

While the token based approaches use tokens as the basic construct for searching, the tree based approaches find similarity by using subtrees as the basic constructs for finding clones. The tree based approach by Baxter *et al.* [2] uses annotated parse trees which are then divided into buckets. The subtrees in the buckets are then compared to search for clones. The searching phase is further improved by comparing the hashed subtrees. Another technique to search for similar subtrees through dynamic programming is proposed by Yang [25]. The technique may work for searching clones having different syntax. An XML based approach proposed by Wahler *et al.* [23] converts Abstract Syntax Tree (AST) into XML. The technique then uses data mining approach for finding clones of various types. Similarly, another technique using deep-learning based detection of clones is proposed by White *et al.* [24]. Their technique works by linking patterns mined at syntactic and lexical levels. The training phase of such techniques may however dominate the overall performance of clone detection procedure.

The widely used Java Code Clone Detection (JCCD) tool [4] incorporates a pipelined approach for detecting clones. The pipeline uses the phases of parsing, pre-processing, pooling and filtering, which generates AST after parsing of source code. Its enhancement proposed in [16] using a divide-and-conquer approach divides an input source file into smaller files, which are then refactored for clone detection. A smaller size of file is shown to produce better performance. Similar to the EncCD framework, their approach also uses the JCCD pipeline, however, in contrast to our approach, it is limited to dividing a source code into smaller parts instead of actually reducing the size of input.

3. Architecture of the EncCD Framework

The EncCD framework incorporates an efficient encoding mechanism to represent the method bodies as encoded text which results in the reduced code size. It deploys the generic pipeline model [4] for detecting clones. The framework initially parses and transforms the code using the steps (parsing, labelled encoding and output) described below:

Let $S = S_1 \times S_2 \times ... \times S_n$ be the set of n input source files. Let parse P be the function that transforms a source code file into a set M with q units, so that, we have:

$P: S_i \rightarrow M$, for i=1,2,..., n, where,

$M = \bigcup(M_k)$, for k=1,2,..., q, and, $M \subseteq S_i$

Let ψ be the labelled encoding function which transforms a unit into encoded form, so that, we have,

$$\psi: M_k \rightarrow M_k^E, \text{ for } k=1,2,...,q.$$

The λ output function uses the encoded set M^E to produce the encoded file S^E, so that, we have,

$$\lambda: M^E \rightarrow S^E$$

Which is then processed through the generic pipeline for the steps of parsing, preprocessing, pooling, comparison and filtering for clones detection.

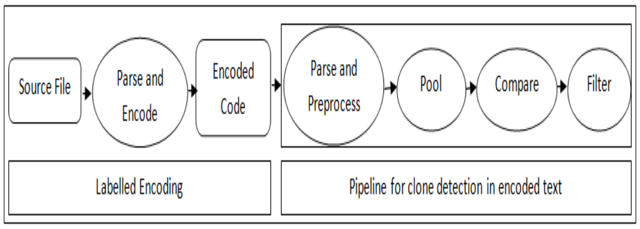

Figure 1. Working mechanism of the encoded pipelined clone detection.

The main steps of the proposed framework are illustrated in Figure 1. The input source is initially parsed and encoded to produce a compact code. For implementation of the EncCD framework, the methods in the source code are used as basic units which are encoded. The encoded code is then input to the generic pipeline of the Java Code Clone Detection (JCCD) tool which parses the code to generate AST representing the syntactical structure of the encoded code. This is followed by pre-processing phase which normalizes the code units. The normalized units are then joined as pools based on different criteria such as similar variable names or values. The contents of every pool are compared for clone detection and then filtered to remove false positives.

The labelled encoding of all the source files is performed by the Encode_Source algorithm (Algorithm 1) as given below.

Algorithm 1: Encode_Source

```
1. foreach (file in source folder)
2. {
3.        Get compilation unit by parsing code
4.        Call Encode_Methods for the compilation unit
5.        Write modified compilation code to file in target folder
6. }
```

Using the Encode_Source algorithm, a compilation unit is initially obtained by parsing the code through JavaParser [5]. The compilation unit is then encoded using the steps 1-6 of the algorithm which contain the main loop iterating over all the files of an input source folder.

For each file, the source code is parsed to obtain a compilation unit which is then passed to the algorithm Encode_Methods (Algorithm 2) to generate an encoded form of the source code. The compilation unit obtained after encoding is then stored as encoded source file. The reduced size of the encoded source file results in efficient parsing and match detection thereby improving the overall performance of clone detection.

Algorithm 2: Encode_Methods

```
1.  Get List of Method Declarations from compilation unit
2.  foreach (method in Method Declarations)
3. {
4.        Let B be the block statement (body) string
5.        start = B.indexof("{")
6.        end = B.lastIndexof("}")
7.        $S_0$ = substring(start+1, end-1)
8.        String arStr [] = split $S_0$ into array of strings (statements)
9.        Create empty Statement stmt, and let lStm be LabeledStatement
10.   List <Statement> aList = new ArrayList()
11.    foreach (line in arStr)
12.    {
13.        Remove spaces from line
14.        if (line is not empty) then
15.        {
16.            String hStr = (String) temp.hashCode()
17.            Replace '-' with '_' in hStr
18.            lStm = new LabeledStatement ( "m"+hStr, stmt)
19.            aList.add ( lStm )
20.        }
21.    }
22.        BlockStmt bst = new BlockStmt()
23.        bst.setStmts(aList)
24.        Set Block Statement of the Method Body = bst
25. }
```

The Encode_Methods algorithm is invoked by the Encode_Source algorithm. The step 1 of the algorithm obtains the list of method declarations whose body blocks are subsequently modified through the loop in steps 2-25. In steps 4-7, the statements of the body block are obtained and placed collectively in a string S_0. Using step 8, the string S_0 is split into statements thereby producing an array of strings arStr. An empty statement and an array list are created in steps 9-10. The loop at steps 11-21 works for each line (substring) in the array of strings arStr to encode each statement.

Table 1. Software together with source size used for experimentation.

Software	**DirBuster [9]**	**J8583 [26]**	**JHotDraw [11]**	**Open Visual Traceroute [15]**	**SableCC [6]**
Size	758 KB	297KB	2.05 MB	474 KB	0.99 MB
Software	Jalopy [22]	PKI Applet [12]	Class Editor [17]	JavaCSV [8]	Apache HTTP Server [3]
Size	3.35 MB	52.0 KB	528 KB	137 KB	11.1 MB

Table 2. Specification of the machines used for experimentation.

	Machine-A	**Machine-B**
Architecture	Intel Core i7 processor, 4 cores	Intel Xeon X5560 processor Server, 2x4 cores
Operating System	Windows 7 64-bit, JDK 1.7	Windows Server 2008, JDK 1.7

After removing spaces at step 13, the hash code is generated for non-empty lines and subsequently processed to generate labelled statements using steps 14-20. At step 16, the hash code is generated, which is then modified at step 17 by replacing '-' with '_' to ensure a valid token in the language. A labelled statement is created at step 18 with a label having an empty statement *stmt*. The label contains the encoded value concatenated with the prefix 'm' so that a valid token is generated. Each labelled statement is added to the array list at step 19, which is then used for generating main block statement of the method in steps 22-24. The overall encoding technique incurs a small overhead which is amortized through reduced parsing time during clone detection, thereby improving the efficiency of code clone detection.

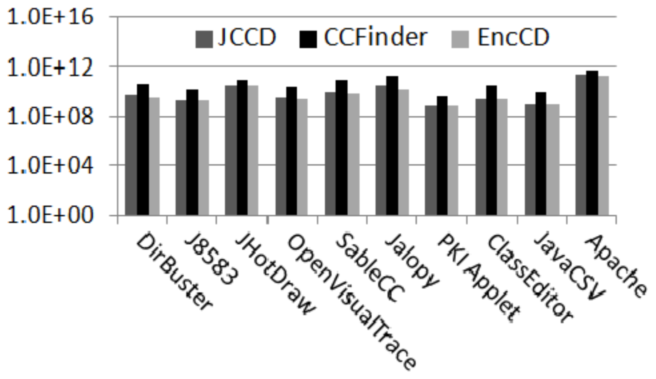

Figure 2. Execution time (in nanoseconds) for clone detection on Machine-A.

4. Experimentation: Implementation and Results

The EncCD framework is implemented by incorporating the algorithms of Encode_Source (Algorithm 1) and Encode_Methods (Algorithm 2) while using the generic pipeline model of JCCD [4]. The code is parsed and modified by using the JavaParser software [5]. For performance evaluation, we perform experimentation using the well known open source software available from sourceforge.net. The source code of every software is initially filtered to contain only .java source files for which clone detection is performed.

4.1. Performance Results on Machine-A

Figure 2 shows the results obtained for the machine having Intel Core i7 based processor. Corresponding to each software, the execution time in nanoseconds (in logarithmic scale) is presented in the Figure. The EnCCD clone detection takes a very small amount of time in comparison with JCCD and CCFinder. It outperforms both these clone detectors in terms of average execution time which is 31999901385, 100859289559, and 2475104568 nanoseconds for JCCD, CCFinder and EnCCD, respectively.

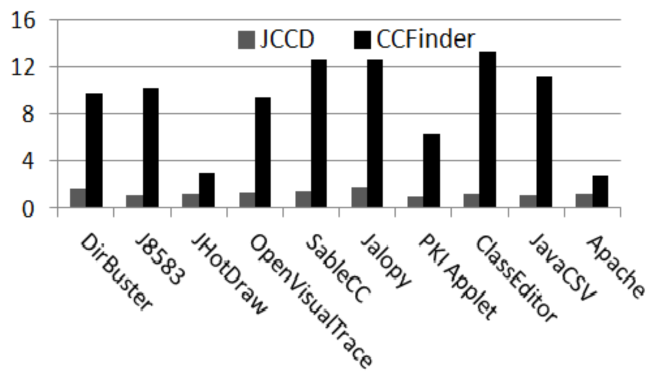

Figure 3. Speedup obtained by EncCD over JCCD and CCFinder on Machine-A.

The speedups obtained by EncCD over JCCD and CCFinder are given in Figure 3. On machine-A, the overall average and maximum speedups obtained by the EncCD clone detector over JCCD are 1.31 and 1.76, respectively. Similarly, the overall average and maximum speedups obtained by EncCD over CCFinder are 9.12 and 13.29, respectively. The significant performance improvement demonstrates the effectiveness of the proposed EncCD framework for clone detection.

4.2. Performance Results on Machine-B

For the machine having the Intel Xeon processor, the performance results in terms of execution time are shown in Figure 4. Similar to the results on Machine-A, the EnCCD clone detection takes a very small amount of execution time and outperforms both the JCCD and CCFinder clone detectors. The average execution time taken for clone detection by JCCD, CCFinder, and EncCD is 23559356466, 81169657557, and 15085237687 nanoseconds, respectively.

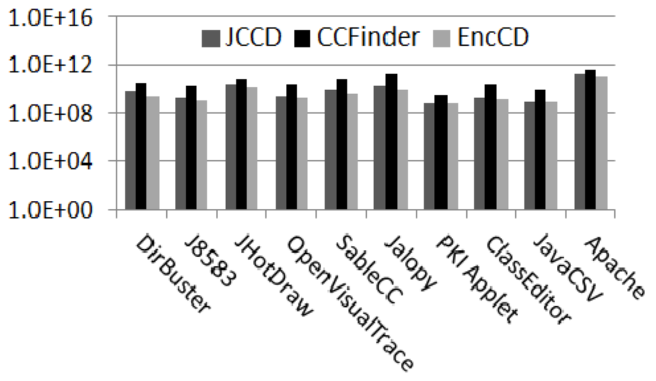

Figure 4. Execution time (in nanoseconds) for clone detection on Machine-B.

Figure 5 shows the speedups obtained by EncCD over JCCD and CCFinder on machine-B. The overall average and maximum speedups obtained by the EncCD clone detector over JCCD are 1.70 and 2.41, respectively. Similarly, the average and maximum speedups obtained by the EncCD clone detector over CCFinder are 11.40 and 19.04, respectively.

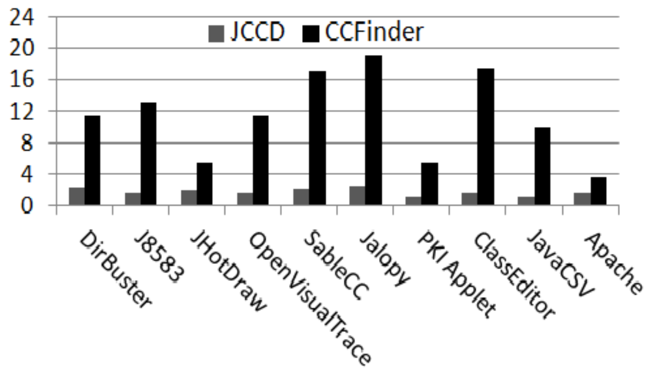

Figure 5. Speedup obtained by EncCD over JCCD and CCFinder on Machine-B.

4.3. Performance Results Summary and Discussion

For both the machines used for experimentation, a summarized view of performance is given in Table 3. On machine-A, the maximum speedup obtained by EncCD over JCCD and CCFinder is for Jalopy and ClassEditor software, respectively, whereas, the minimum speedup by EncCD over JCCD and CCFinder is for PKIApplet and Apache software, respectively. On machine-B, however, the maximum speedup obtained by EncCD over JCCD and CCFinder both is for the Jalopy software, whereas, the minimum speedup by EncCD over JCCD and CCFinder is for PKIApplet and Apache software, respectively. Overall, a better performance enhancement is obtained on the machine-B in comparison with the machine-A.

The difference in the speedup occurs mainly due to the source code pattern and number of files being processed for clone detection in a software. On the Intel Xeon based system, the clone detection performance is better since the work-pool of threads used for clone detection by the frameworks is able to fully exploit the cores available on the system.

Table 3. A summarized analysis of the speedup obtained by the *EncCD* framework.

	Machine-A		Machine-B	
	JCCD	**CCFinder**	**JCCD**	**CCFinder**
Max. Speedup	Jalopy	Class Editor	Jalopy	Jalopy
Min. Speedup	PKIApplet	Apache	PKIApplet	Apache

5. Conclusions

This paper proposes a framework called *EncCD*, which aims at efficient detection of code clones. The proposed framework combines the pipelined approach with encoded detection. The source code after parsing is encoded with labelled statements through a lightweight mechanism. It incurs a very small overhead which is amortized through enhanced efficiency obtained due to smaller size of the source code.

The proposed EncCD framework outperforms the well-known JCCD and CCFinder clone detectors in terms of execution speed. On the Intel Core i7 based system, the average speedups of clone detection obtained by EncCD over JCCD and CCFinder are 1.31 and 9.12, respectively. Similarly, on the Intel Xeon based system, the average speedups of clone detection obtained by EncCD over JCCD and CCFinder are 1.70 and 11.40, respectively.

As future work, we intend to incorporate a multi-pipeline architecture to further improve the performance of clone detection while supporting dynamicity in terms of phases depending upon the available computational resources.

References

[1] Baker B., “On Finding Duplication and Near-duplication in Large Software Systems,” *in Proceedings of the 2nd Working Conference on Reverse Engineering*, Washington, pp. 86-95, 1995.

[2] Baxter I., Yahin A., Moura L., Santanna M., and Bier L., “Clone Detection Using Abstract Syntax Trees,” *in Proceedings of the International Conference on Software Maintenance*, Bethesda, pp. 368-377, 1998.

[3] Behlendorf B., “Apache HTTP Server Project", Apache, Available at: https://httpd.apache.org/, Last Visited, 2016.

[4] Biegel B. and Diehl S., “JCCD: A Flexible and Extensible API for Implementing Custom Code Clone Detectors,” *in Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, New York, pp. 167-168, 2010.

[5] Bruggen D., “JavaParser,” Available at: http://javaparser.org, Last Visited, 2016.

[6] Cagnon E., “SableCC,” SableCC.org, Available at: http://www.sablecc.org/, Last Visited, 2016.

[7] Ducasse S., Rieger M., and Demeyer S., “A Language Independent Approach for Detecting Duplicated Code,” *in Proceedings of the IEEE International Conference on Software Maintenance*, Oxford, pp. 109-118, 1999.

[8] Dunwiddie B., “Java CSV,” Csvreader.com, Available at: https://www.csvreader.com/, Last Visited, 2017.

[9] Fisher J., “OWASP DirBuster Project,” Owasp.org, Available at: https://www.owasp.org/index.php /Category:OWASP_DirBuster_Project, Last Visited, 2017.

[10] Gauci R., “Smelling out Code Clones: Clone Detection Tool Evaluation and Corresponding Challenges,” *CoRR*, vol. abs/1503.00711, 2015.

[11] Gamma E. and Eggenschwiler T., “JHotDraw as Open-Source Project,” JHotDraw.org, Available at: http://www. jhotdraw.org/, Last Visited, 2016.

[12] Javacardos F., “Java Card PKI Applet,” Sourceforge, Available at: https://sourceforge.net/projects/java-card-pkiapplet/, Last Visited, 2016.

[13] Johnson J., “Identifying Redundancy in Source Code Using Fingerprints,” *in Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, pp. 171-183, 1993.

[14] Kamiya T., Kusumoto S., and Inoue K., “CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code,” *IEEE Transactions on Software Engineering*, vol. 28, pp. 654-670, 2002.

[15] Lewis L., “Open Visual Traceroute,” Visualtraceroute, Available at: http://visualtraceroute.net/, Last Visited, 2016.

[16] Mubarak-Ali A., Syed-Mohamad S., and Sulaiman S., “Enhancing Generic Pipeline Model for Code Clone Detection using Divide and Conquer Approach,” *The International Arab Journal of Information Technology*, vol. 12, no. 5, pp. 510-517, 2015.

[17] Mohapatra T., “Java Class File Editor,” Sourceforge, Available at: http://classeditor.sourceforge.net/, Last Visited, 2016.

[18] Roy C., Cordy J., and Koschke R., “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach,” *Science of Computer Programming*, vol. 74, pp. 470-495, 2009.

[19] Sajnani H., Saini V., Svajlenko J., Roy C., and Lopes C., “SourcererCC: Scaling Code Clone Detection to Big Code,” *in Proceedings of the 38th International Conference on Software Engineering*, Texas, pp. 1157-1168, 2016.

[20] Smith R. and Horwitz S., “Detecting and Measuring Similarity in Code Clones,” *in Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, USA, pp. 28-34, 2009.

[21] Svajlenko J., Keivanloo I., and Roy C., “Big Data Clone Detection Using Classical Detectors: an Exploratory Study,” *Journal of Software: Evolution and Process*, vol. 27, no. 6, pp. 430-464, 2015.

[22] Triemax S., “Jalopy Java Source Code Formatter Beautifier Pretty Printer,” TrieMax, Available at: https://www.triemax.com/, Last Visited, 2016.

[23] Wahler V., Seipel D., Gudenberg J., and Fischer G., “Clone Detection in Source Code by Frequent Itemset Techniques,” *in Proceedings of the Source Code Analysis and Manipulation, 4th IEEE International Workshop*, Washington, pp. 128-135, 2004.

[24] White M., Tufano M., Vendome C., and Poshyvanyk D., “Deep Learning Code Fragments for Code Clone Detection,” *in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, New York, pp. 87-98, 2016.

[25] Yang W., “Identifying Syntactic Differences Between Two Programs,” *Software-Practice and Experience*, vol. 21, no. 7, pp. 739-755, 1991.

[26] Zamudio E., “Introduction to ISO8583,” Sourceforge, Available at: http://j8583.sourceforge.net/ iso8583.html, Last Visited, 2016.

Minhaj Khan obtained his MS and Ph.D degrees from University of Versailles, France. He is currently working as Associate Professor at Bahauddin Zakariya University, Multan. His research interests include code optimization and high performance computing.