# PAN@FIRE: Overview of CL-SOCO Track on the Detection of Cross-Language SOurce COde Re-use

Enrique Flores[*]
Universitat Politècnica de València
Spain
eflores@dsic.upv.es

Paolo Rosso
Universitat Politècnica de València
Spain
prosso@dsic.upv.es

Esaú Villatoro-Tello
Universidad Autónoma Metropolitana
Unidad Cuajimalpa
Mexico
evillatoro@correo.cua.uam.mx

Lidia Moreno
Universitat Politècnica de València
Spain
lmoreno@dsic.upv.es

Rosa Alcover
Universitat Politècnica de València
Spain
ralcover@eio.upv.es

Vicente Chirivella
Universitat Politècnica de València
Spain
vchirive@eio.upv.es

## ABSTRACT

The detection of source code re-use is an important research field for both software industry and academia fields. This paper summarizes the goals, organization and results of the second SOCO competitive evaluation campaign for systems that automatically detect the source code re-use phenomenon. PAN@FIRE shared task, named Cross-Language SOurce COde Re-use (CL-SOCO), focused on the detection of cross-lingual re-used source codes in C and Java programming languages. Participant systems were asked to annotate several source codes as whether or not they represent cases of cross-lingual source code re-use. In total five teams participated and submitted 12 runs. The training and test collections were generated using an automatic translation tool establishing a standard evaluation framework for future research works in cross-language source code re-use detection. Although the results obtained by the participants look promising, the problem requires more efforts to be accurately solved.

## CCS Concepts

**•General and reference → General conference proceedings;**

## Keywords

CL-SOCO; Cross-Language Source Code re-use; Plagiarism detection; Evaluation framework; Test collections

## 1. INTRODUCTION

The digital era and the growth of the Web have turned easily accessible the information through blogs, forums, repositories, etc. This easy access tempts the programmers to re-use source codes from external resources. According to a report from the Business Software Alliance, the losses from fraudulent use of software ascend to billions of euros[1]. In academia, a survey asserts that the 30% of instances of re-use occur in source codes [2]. It is practically impossible to compare manually between large collections of source codes. Hence, there is a real need of developing automatic tools to accurately detect the source code re-use phenomenon [3].

A particular type of source code re-use that is being studied recently is the cross-language scenario [1, 4]. A programmer finds a source code written in a programming language $PL$ but he/she needs it in a different language $PL'$. By manual or automatic translation, source code re-use is committed in a cross-language way. Another possible situation is for retrieving source codes, when a programmer requires a certain implementation of an algorithm in the programming language $PL'$ but he owns a source code written in $PL$. This is a more challenging scenario than the monolingual one because of different programming languages could not share reserved words, libraries or programming syntax.

Whereas at PAN@FIRE 2014 the shared task addressed source code re-use detection in a monolingual context [5], this year edition (CL-SOCO) focuses on the detection of source codes that have been re-used in a cross-lingual environment. Particularly, CL-SOCO involves identifying and distinguishing the most similar source code pairs among a source code collection written in C and Java. In the rest of the paper we will first define the task and then summarise all participant systems approach as well as their obtained results during the CL-SOCO 2015 shared task.

## 2. TASK DESCRIPTION

CL-SOCO shared task focuses on cross-lingual source code re-use detection, which means that participant systems have to deal with the case where the suspicious and original source codes are written in different programing languages. Accordingly, participants are provided with a set of source codes written both in C and Java languages, where source codes have been tagged by language to ease the detection. Thus the task consists in retrieving source code pairs that have been re-used. It is important to mention that this task must be performed at document level, hence no specific fragments inside of the source codes are expected to be identified; only pairs of source codes. Therefore, participant systems were asked to annotate several source codes as whether or not they represent cases of source code re-use.

[*]Corresponding author.
[1]http://globalstudy.bsa.org/2013/

**Table 1: Characteristics of the training corpus. Here, it is described the tokens different than punctuation marks, the number of lines of code and the amount of re-used cases between C and Java collection.**

| | # Source codes | Tokens | Lines | # Re-used cases |
|---|---|---|---|---|
| C | 599 | 86,001 | 25,226 | 599 |
| Java | 599 | 102,239 | 31,713 | |

The task was divided in two main phases: training and testing. For the training phase we provided an annotated corpus for each programming language, *i.e.,* C and Java. Such annotation includes information about whether a source code has been re-used and, if it is the case, what its original code is. It is worth mentioning that the order of each pair was not important, *e.g.*, if $X$ has been re-used from $Y$, it was considered as valid to retrieve the pair $X \rightarrow Y$ or the pair $Y \rightarrow X$. An additional challenge in plagiarism detection is to determine the *direction* of the plagiarism, *i.e.,* which document is the original and which the copy. Finally, the only annotation that has been provided for the test phase is the programming language that each source code belongs to.

## 3. CORPUS

In this section we describe the two corpora used in the CL-SOCO 2015 task. For the training and testing phases, a corpus composed by source codes written in C and Java programming languages was released. In both phases, the cross-language source code re-use was created automatically by means of the source code translator *C++ to Java Converter*[2]. *C++ to Java Converter* is able to refactor source code if necessary. That is, an alteration of the source code but keeping exactly the same behaviour.

### 3.1 Training Corpus

The training collection consists of source codes written in C and thereafter translated into Java. For the construction of this collection we employed the Rosettacode repository[3]. `Rosettacode.org` is a website that presents solutions to the same task in as many different programming languages as possible. In a snapshot of Feb. 27, 2012, there were 599 solved tasks of solutions written in C programming language. Table 1 shows the characteristics of the training corpus.

### 3.2 Test Corpus

The provided test corpus has been created from the C corpus used in the training phase of SOCO 2014 and also in [1]. From 79 source codes in C programming language, a set of automatically translated 79 source codes written in Java were created. Each of the 79 source code in C and its translation to Java are considered a cross-language re-used pair. This re-used source code pair is named *translated re-use.*

The C collection has a particularity as to its origin, it contains monolingual re-use among the C source codes. Therefore, we name this re-use has spread across programming languages when translating the source codes as *propagated re-use.* For example, if $A.c \leftrightarrow B.c$ was a monolingual re-used case in the original C partition. It generated two cross-lingual re-used source code pairs after the translation process $A.c \rightarrow B.java$ and $B.c \rightarrow A.java$ in addition to the automatically translated $A.c \rightarrow A.java$ and $B.c \rightarrow B.java$. Figure 1 shows the example of the types of the existing re-use described previously.

Both the *propagated* and *translated re-use* were considered for the cross-lingual task. In Total 131 re-used cases were considered for CL-SOCO, 79 instances of *translated re-use* and 52 of *propagated re-use.* Table 2 shows the characteristics of the test corpus.

[2]http://www.tangiblesoftwaresolutions.com/Product_Details/CPlusPlus_to_Java_Converter_Details.html
[3]http://rosettacode.org/

**Table 2: Characteristics of the test corpus. Here, it is described the tokens different than punctuation marks, the number of lines of code and the amount of re-used cases between C and Java collection.**

| | # Source codes | Tokens | Lines | # Re-used cases |
|---|---|---|---|---|
| C | 79 | 13,171 | 7,169 | 131 |
| Java | 79 | 15,829 | 7,144 | |

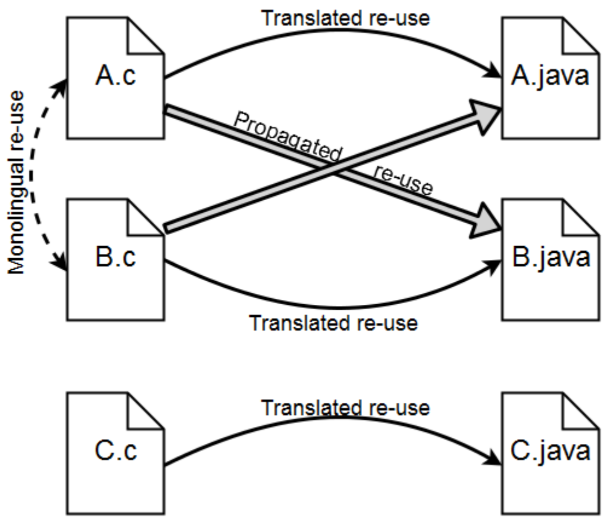

**Figure 1: Example of source code re-use.**

## 4. EVALUATION METRICS

All the participants were asked to submit a detection file with all the considered re-used source code pairs. Participants were allowed to submit up to three runs. All the results were required to be formatted in XML as shown below. As can be noticed, for each suspicious source code pair it must be one entry of the <reuse_case .../> in the XML file. Figure 2 shows an example of the structure of the XML file.

To evaluate the detection of re-used source code pairs we calculate Precision, Recall and $F_1$ measure. For ranking all the submitted runs we used the $F_1$ measure in order to favour those systems that were able to obtain (high) balanced values of Precision and Recall.

**Figure 2: Example of the XML submission file.**

```
<document>
<reuse_case
source_codeC="X1" source_codeJ="Y1"
/>
<reuse_case
source_codeC="X2" source_codeJ="Y2"
/>
...
</document >
```

## 5. PARTICIPATION OVERVIEW

In total five teams participated and submitted 12 runs. Particularly, the Department of Computer Science, Gujarat University, India (`CLSCR`), the SkyLine LLC, Zhytomyr State University, Ukraine (`Palkovskii`), the PES Institute of Technology, PES University, India (`PES_BSec`), the Autonomous University of the State of Mexico (`UAEM`) and the Universidad Autónoma Metropolitana - Unidad Cuajimalpa (`UAM-C`). `CLSCR` team submitted only one run, `Palkovskii` team submitted two while the rest of the teams submitted three.

`UAM-C` [8] presented a method to represent a pair of source code documents using five high level features: *(i)* Lexical similarity (3-grams of characters without reserved words); *(ii)* Stylistic similarity (considering a set of 11 features, such as, number of line of code, white spaces, tabulations, number of empty lines, number of lower case letters, etc.); *(iii)* Comments similarity (text within the comments sections); *(iv)* Similarity of the text set by the programmer (either it is produced by the program or it is passed as an argument to a function); and *(v)* Structure similarity (considering a set of 9 features, such as, number of relational operations, assignations, number of function calls, number of looping statements and number of return statements). Accordingly, each pair of source code is represented using these five features, and later, classified as re-used case or not using the Random Forest algorithm. The first run was trained considering the re-used source code reuse pairs of SOCO 2014 (C training partition) using the representation described above. In the second run, the model was trained using the re-used source code pairs of CL-SOCO 2015 (training set) using the representation described above. In the third run, only the lexical similarity was taken into account, with a manually defined similarity threshold set to 20%. That is, every pair of source code in the test set with a lexical similarity of 20% or more is labelled as a re-use case.

`PESB_Sec` [9] used a model divided into four steps: *(i) Preprocessing* - All characters converted into lowercase, whitespace removal and accent strip. The output from this stage is a stream of tokens for each document; *(ii) Weighting* - Tf-idf vectors are created from the tokenized documents; *(iii) Similarity threshold estimation* - A similarity threshold is established considering the average cosine similarity of the training corpus; and *(iv) Decision* - A pair of source codes is considered as a cross-language re-used case if its cosine similarity value is greater than the threshold established. The first run applies the four steps previously described. In the second run, the top-$n$ tokens most frequent in C and Java training set were removed after pre-processing. In the third run, similar C and Java operations were replaced by *opcodes*, e.g. *println* and *printf* by *op1* or *argv* and *argValue* by *op2*. The replacements were done manually after selecting the top-n tokens more frequent in the training set, and assigning opcodes based on meaning of the word.

`UAEM` [6] proposed a system that divided in four phases: *(i) preprocessing* it only the lexical items of each source language are separated and more than one whitespaces are removed. It also replaces some Java commands to C commands; *(ii) similarity estimation* it uses as similarity measure the sum of the different lengths of the longest common substrings between the two source codes, normalised to the length of the longest code; *(iii)* ranking a set of parameters is obtained from the previous comparisons that allow later the identification of re-used cases. The parameters obtained are: the value of the *distance* (1-similarity), the *ranking* of the distance (rank order of the most similar), the *gap* that exists with the next closest code (it is only calculated for the first 10 closest codes) and, using the maximum gap between the 10 most closest codes, the source codes that are $B$efore or $A$fter the maximum gap *relative difference* are labelled. The result of the third phase is a matrix where each row represents a comparison of a source code with other codes (columns); and *(iv) decision* here, a source code pair $X \leftrightarrow Y$ will be a re-use case if there is evidence of re-use in both directions, it means, $X \rightarrow Y$ and $Y \rightarrow X$. A re-used case exists when the *distance* is less than 0.45 or the *gap* is greater than 0.14, but also it is important that one of the additional conditions is achieved. The first condition is that the *ranking* must be, at least, in the second position and, the second condition, that the label of the *relative difference* must be *Before.* The first run was processed with above conditions. However, in some cases the evidence in one direction was very high and in the other direction was almost reliable. In the second and third run, if there were not high evidence of re-use in one direction, then the pair could be considered under less restrictive conditions but with a certain evidence.

`CLSCR` [10] mainly used two components: *(i)* a compiler that compiles and translate the language specific source code into a tool specific internal format; and *(ii)* the similarity is computed between internal formats of different programs.

`Palkovskii` [7] approach uses a common sequence of tokenizer, an n-gram splitter (5-words for a token). Then, it uses DB-scan algorithm for match clustering, spatial index for cluster generation and several post-processing heuristics including merging and skipping. It also applies stopwords removal and several means of additional $n$-gram generation - structural $n$-grams, regular $n$-grams, skip-word $n$-grams, NER-based $n$-grams. And it also uses $tf$-$idf$ sliding windows for detection of highly obfuscated segments.

## 6. RESULTS AND ANALYSIS

The results obtained by the participants are shown in Table 3. As we mentioned before, we ranked the obtained results by means of the $F_1$ measure, given that we prefer systems that are able to obtain (high) balanced values of Precisions and Recall. We also considered important to show results by type of re-use detected. Table 4 shows the results of the *translated re-use* while Table 5 shows the *propagated re-use*.

The best results according to $F_1$ were obtained by `UAM-`

**Table 3: Overall evaluation results for CL-SOCO task. The ranking is upon the $F_1$ values.**

| Run | F1 | Precision | Recall |
|---|---|---|---|
| UAM-C_run1 | 0.772 | 0.988 | 0.634 |
| Palkovskii_run1 | 0.752 | 1.000 | 0.603 |
| PES_BSec_run2 | 0.740 | 1.000 | 0.588 |
| UAEM_run1 | 0.739 | 0.975 | 0.595 |
| Palkovskii_run2 | 0.724 | 0.962 | 0.580 |
| UAEM_run2 | 0.709 | 1.000 | 0.550 |
| UAEM_run3 | 0.703 | 1.000 | 0.542 |
| PES_BSec_run3 | 0.697 | 1.000 | 0.534 |
| UAM-C_run2 | 0.687 | 0.620 | 0.771 |
| PES_BSec_run1 | 0.683 | 1.000 | 0.519 |
| UAM-C_run3 | 0.655 | 0.496 | 0.962 |
| CLSCR_run1 | 0.611 | 0.952 | 0.450 |

`C` team in CL-SOCO. Nevertheless, an analysis of variance (ANOVA) showed that there is no statistical difference between the run 1 of `UAM-C` and the following three runs. In fact, the difference between the first and the fourth is only 0.033. Overall results are shown in Table 3. In general, all the runs achieved a good performance scoring a $F_1$ value higher than 0.6. Most of the participant runs obtained better value of Precision than Recall except `UAM-C` in runs 2 and 3. In this two runs, their model retrieve more re-used source code pairs but with an impact on the Precision.

As test partition contains different kinds of cross-language source code re-use (*translated* and *propagated*), it is worth to analyse them separately. The results of the *translated* re-use is summarised in Table 4. This type of re-use was created taking each source code in C and automatically translated into Java. The *translated* re-use may have included some changes to the expected exact translation of a source code. For example, if a library function is not known by the translator, the translator treats the function as unknown and creates a new function with the same name leaving its body function to be completed by the programmer. This *refactoring* process makes this scenario a little more complicated than a simply verbatim copy between programming languages. `Palkovskii` and `UAEM` runs have shown high performance in this scenario. Also `PES_BSec` and `CLSCR` achieved $F_1$ values higher than 0.8. Considering the Recall value, mostly all the runs were able to retrieve a high percentage of the *translated* re-use cases (10 out of 12 retrieved more than 0.8 of Recall). The precision value is not so important as Recall in this table because we are considering the *propagated* re-used cases as non re-used case on this results. Here, the precision only provides a reference of the amount of *translated* re-used cases reported out of the retrieved source codes. For example, `Palkovskii_run1` shows that practically all its retrieved source code pairs correspond to this scenario while the 0.311% of the `UAM-C_run3` retrieved source code pairs correspond to *translated* re-used cases.

The second case contemplated *propagated* re-used cases. This kind of re-use consists of monolingual re-used cases that were translated into another language. This is a more challenging scenario than the *translated* re-use because it takes into account alterations of the re-used source code at monolingual level. Most of the runs have achieved poor results if Recall value is considered. Only `UAM-C_run3` is able to retrieve more than a half of this kind of cross-language source code re-use (0.904%), as shown in the overall results in Table 3, this has a high impact on its Precision values. The Precision value is not so important as Recall in this table because we are considering the *translated* re-used cases as non re-used case on this results. The Precision values show that the *propagated* re-used cases retrieved are a minority of the total retrieved in all the runs submitted.

**Table 4: Performance of participant runs only considering the *translated re-use*.The ranking is upon the $F_1$ values.**

| Run | F1 | Precision | Recall |
|---|---|---|---|
| Palkovskii_run1 | 0.975 | 0.975 | 0.975 |
| UAEM_run3 | 0.933 | 0.986 | 0.886 |
| UAEM_run2 | 0.927 | 0.972 | 0.886 |
| Palkovskii_run2 | 0.924 | 0.924 | 0.924 |
| PES_BSec_run2 | 0.910 | 0.922 | 0.899 |
| UAEM_run1 | 0.893 | 0.887 | 0.899 |
| PES_BSec_run1 | 0.871 | 0.941 | 0.810 |
| PES_BSec_run3 | 0.859 | 0.914 | 0.810 |
| CLSCR_run1 | 0.823 | 0.935 | 0.734 |
| UAM-C_run1 | 0.736 | 0.714 | 0.759 |
| UAM-C_run2 | 0.628 | 0.466 | 0.962 |
| UAM-C_run3 | 0.474 | 0.311 | 1.000 |

**Table 5: Performance of participant runs only considering the *propagated re-use*.The ranking is upon the $F_1$ values.**

| Run | F1 | Precision | Recall |
|---|---|---|---|
| UAM-C_run1 | 0.338 | 0.274 | 0.442 |
| UAM-C_run3 | 0.307 | 0.185 | 0.904 |
| UAM-C_run2 | 0.233 | 0.153 | 0.481 |
| UAEM_run1 | 0.106 | 0.087 | 0.135 |
| PES_BSec_run3 | 0.098 | 0.086 | 0.115 |
| PES_BSec_run2 | 0.093 | 0.078 | 0.115 |
| PES_BSec_run1 | 0.067 | 0.059 | 0.077 |
| Palkovskii_run2 | 0.046 | 0.038 | 0.058 |
| UAEM_run2 | 0.032 | 0.028 | 0.038 |
| Palkovskii_run1 | 0.031 | 0.025 | 0.038 |
| CLSCR_run1 | 0.018 | 0.016 | 0.019 |
| UAEM_run3 | 0.016 | 0.014 | 0.019 |

In general, different approaches were applied to solve the problem of cross-language source code re-use detection. Proposed approaches vary from string-matching to compiler based models. Additionally, given that all these approaches were evaluated under the same conditions employing the same collections, it was possible to make a more fair comparison among participant systems. Accordingly, the best performing model was the combination of lexical and structural features [8] but with no statistical differences between the following three runs. `UAM-C` showed a more robust performance in both scenarios that is reflected in the overall results. It does not achieve the best results when detecting the *translated* re-used cases, but it does achieve a balanced results on this and also a good recall in the *propagated* re-use scenario causes that obtain the best results.

## 7. FINAL REMARKS AND FUTURE WORK

In this paper we presented the overview of the Cross-Language Detection of SOurce COde Re-use (CL-SOCO) PAN track at FIRE. Especially, CL-SOCO 2015 provided a task specification which is particularly challenging for participating systems. The task was focused on retrieving cases of cross-language re-used source code pairs from a collection of programs. At the same time, CL-SOCO provided an evaluation framework where all participants were able to compare their obtained results by means of applying different approaches under the same conditions and using the same corpus. With these specifications, the task has turned out to be particularly challenging and an opportunity to compare different approximations tackling cross-language source code re-use detection.

In total five teams participated and submitted 12 runs. We summarise the followed approaches by each of the participant systems and presented the evaluation of submitted runs along with its respective analysis. In general, different approaches were proposed, varying from string-matching approaches to compiler-based ones. The team that achieved the best results was `UAM-C` by means of their combination of views approach (lexical, stylistic and structural). The majority of the models achieved high performance when detecting *translated* re-used cases while not quite good results in the *propagated* scenario. This second kind of re-use is a more challenging scenario that needs to be considered in future research works.

Finally, a note has to be made with both training and test collections that represent a valuable resource for future research work on the field of cross-language source code re-use identification.

## 8. ACKNOWLEDGMENTS

PAN@FIRE (CL-SOCO) has been organised in the framework of DIANA-APPLICATIONS (TIN2012-38603-C02- 01) research project. The fourth author would like to thank to UAM Cuajimalpa and SNI-CONACyT for their support.

## 9. REFERENCES

[1] C. Arwin and S. Tahaghoghi. Plagiarism detection across programming languages. *Proceedings of the 29th Australian Computer Science Conference, Australian Computer Society*, 48:277–286, 2006.

[2] D. Chuda, P. Navrat, B. Kovacova, and P. Humay. The issue of (software) plagiarism: A student view. *Education, IEEE Transactions on*, 55(1):22–28, 2012.

[3] E. Flores, A. Barrón-Cedeño, L. Moreno, and P. Rosso. Uncovering source code reuse in large-scale academic environments. *Computer Applications in Engineering Education*, 23(3):383–390, 2015.

[4] E. Flores, A. Barrón-Cedeño, L. Moreno, and P. Rosso. Cross-Language Source Code Re-Use Detection using Latent Semantic Analysis. *Journal of Universal Computer Science*, In press.

[5] E. Flores, P. Rosso, L. Moreno, and E. Villatoro-Tello. On the detection of SOurce COde re-use. In *Proceedings of the Forum for Information Retrieval Evaluation*, FIRE '14, pages 21–30. ACM, 2015.

[6] R. García-Hernández and Y. Lendeneva. Cross-Language Identification of Similar Source Codes based on Longest Common Substrings.

[7] Y. Palkovskii. Submission to the 2nd International Competition on Cross-Language SOurce COde Re-use detection.

[8] A. Ramírez-de-la Cruz, G. Ramírez-de-la Rosa, C. Sánchez-Sánchez, H. Jiménez-Salazar, C. Rodríguez-Lucatero, and L.-R. W. High level features for detecting source code plagiarism across programming languages.

[9] S. Saimadhav Heblikar, P. Sharma, M. Munnangi, and C. Bankapur. Normalization based stop-word approach to source code reuse detection.

[10] D. Shah, H. Jethani, and H. Joshi. (CLSCR) Cross Language Source Code Reuse Detection using Intermediate Language.