Enhancing Notation-Based Code Cloning Method With an External-Based Identifier Model

NGOC-TU CHAU® AND SOUHWAN JUNG®
School of Electronic Engineering, Soongsil University, Seoul 06978, South Korea
Corresponding author: Souhwan Jung (souhwanj@ssu.ac.kr)

This work was supported in part by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea Government (MSIT) (Development of 5G Edge Security Technology for Ensuring 5G+ Service Stability and Availability) under Grant 2020-0-00952, and in part by the Ministry of Science and ICT (MSIT), Korea Government through the Information Technology Research Center (ITRC) support program supervised by the Institute for Information & Communications Technology Planning & Evaluation (IITP) under Grant IITP-2020-2020-0-01602.

ABSTRACT Code clone detection is known for solving code paradigm problem in software development. Malware analysts also apply this technique to detect whether a set of malware applications originated from the same malware family based on the similarity in their source code. Until now, existing notation-based approaches are based on standard identifier notations and generating signatures from the notations output. Specifically, for a set of words (or lexemes), the analysts apply rules to determine the type of a lexeme and label each lexeme with a token type. So far, none of the existing code clone detection models considers collecting identifiers from an external source. In this paper, we propose a novel External-based Identifier Model for code clone detection. The proposed model assumes the existence of an external source code that can be used as a supervisor for identifying and labeling a specific set of lexemes. By introducing the external-identifiers into source code detection, our model could distinguish between multiple fragments of code that share the same sequence of standard tokens. One of the study cases for our model is Android analysis where the Android Open Source Project can be used as an external source. The experiment on one millions line of Android source code has shown that our proposed solution could reduce the number of multiple code mapping to single signature situations in comparison with the traditional method. Furthermore, the experiment on code suggestion has proved that our model could reduce the suggestion step for providing faster output than a notation-based approach.

INDEX TERMS Program analysis, code clone detection.

I. INTRODUCTION

Code cloning (also known as code paradigm or code reuse) is an act of duplicating and modifying a code fragment of the other software. The definition of code fragment was stated by M. White et al. in their paper as follows: “A code fragment is a contiguous segment of source code, specified by the source file and the lines where the segment begins and ends” [1]. The copied and original code are considered semantically similar and can be characterized through their syntax [2]. The research for code cloning detection has several impacts for both software security as well as software analysis (e.g. Vulnerability clone detection [3], source code analysis [4], software maintenance [5]). During the analysis process, the program structure is converted into a more semantic-based context using parser techniques.

Abstract Syntax Tree, also known as Syntax Tree (or AST, in short), is an intermediate code representation (or IR) of given source code. AST is originally a part of the program compilation process and based on the study of lexical analysis. Compilers are tools that are used to translate a program’s language into another type of language. The translating process is called a compiling process. Common compiling consists of two stages: 1) Translate from an input program code to intermediate code (which called front-end stage) and 2) Generate target language from intermediate code (which called as back-end stage).

Fig. 1 illustrates the front-end stage of compilation process. This stage consists of a tokenizer and parser. The tokenizer (or also known as lexer) is the process of dividing the stream of
characters in the source code (lexemes) into tokens. A lexeme is a stream of characters included in the program source code that represents a specific context. The context of each lexeme is described by a token. The parser helps the compiler to understand the grammar of source code and fixes the syntax error. The AST is a representation of the output generated by the front-end stage [6].

Abstract syntax model is not only used in the compiling process but also applied in static analysis of program source code years ago [7]. In 1998, ID. Baxter et.al. has proposed the use of AST for clone detection of the source code [8]. In 2005, I. Neamtiu and others have applied AST for track changes in source code [9]. In 2013, E. Söderberg et.al has introduced the use of AST for flow analysis at the abstract-level [10]. Not only C language, but the AST-based model has also been applied across various high-level languages for different purposes, such as Java [11].

For generating syntax outputs from given source code, analysis tools follow a set of metasyntax rules and notations. One of the well-known notation techniques is Backus–Naur Form (BNF) [6]. Another well-known version is Extended Backus–Naur Form (EBNF) which was adopted by the International Organization for Standardization (ISO) in 1996 [12]. A programming language can declare its notations based on BNF or EBNF standard [13], [14]. A simple Java-based BNF notation is as in grammar I. In the grammar I, the left operand is an assignment of token type and the right operand is a set of lexemes and their conditions.

Grammar 1. Example of Java-based BNF notation

Although applying BNF and similar standards to the analysis tool can generate abstract contexts from the input source, different inputs can still produce the same outputs. In this paper, we call the same output cases as indistinguishable cases. Figure 2 illustrates the indistinguishable case within the Android Open Source Project (AOSP) source code. The figure 2 shows the two lines of code (the third level) that share the same consequence of tokens (the second level).

For solving the problem of indistinguishable cases, analysts have to do an exhausting works of defining all distinguishable cases in their notations. In this paper, we propose the utilization of using external sources to enhance the accuracy of code clone detection tools. This proposal assumes the existence of an external source code that can replace a notation rule. This approach provides the flexibility to adjust the semantic contexts in different programming languages as well as platforms sharing the same programming language (e.g. Android and Java).

The proposed model consists of two sub-models: 1) Aggregation model and 2) Reference model. Aggregation model is a solution for generating aggregated lexeme from aggregating multiple-lexemes and a reference-based model is a solution for labeling an aggregated lexeme. Both of the sub-models use an external source for supporting. The experiments were conducted on Android that supports both Java and Android contexts.

This paper is organized as follows. Section II contains the backgrounds about syntax analysis and the previous research on security as well as Android analysis using syntax analysis. The section II also consists of our literature review about existing code clone detection research. Section III introduces the aggregation algorithms and reference-based models. Section IV provides experiments that provide a comparison between traditional Java-based and proposed models on Android, called as Android AST (AAST). Section V provides discussions and section VI concludes this paper.

II. BACKGROUND AND RELATED WORKS

This section covers the knowledge and research related to this paper. The first subsection covers the beginning and development of syntax analysis. This section also provides related works about the applicability of syntax analysis with related to security and Android. The second subsection provides background knowledge of code clone detection. Recent studies related to code clone issues are also covered.
A. BACKGROUNDS ON SYNTAX ANALYSIS

The syntax analysis (or lexical analysis) was originated from morphology which is the study of the internal structure of words [15]. In 1976, DeRemer, Franklin L. has introduced early research on lexical analysis [16] for constructing compilers. K.Church and others provided their research on lexicographic and computational tools in 1991 [17]. In 2004, Chess, Brian, and G. McGraw have published their research on applying lexical analysis on security [7].

A program code contains a sequence of characters that could share the same structure of data with the same meaning. In the syntax analysis, an instance of data, often a word, is defined as lexemes and the structure of data is defined as tokens. The common structure of a token consists of name and value. Token value is the lexemes and token names are various depending on the programming language. Figure 3 illustrates the list of common tokens.

Although the lexical analysis was introduced a long time ago, the application of lexical and abstract syntax analysis into security has only been recently noticed. In 2006, Pozza, Davide, et al. conducted their comparisons of lexical analysis tools for network security [18]. G. McGraw published his research on enhancing automated code review with lexical analysis [19]. In 2016, Mamun, Mohammad Saiful Islam, et al. published their research on applying lexical analysis for detecting malicious URLs [20]. Until now, lexical analysis still has been applied for various security purposes, especially on application analysis [21]–[23].

With the growth of lexical analysis study on the security field, the research of applying abstract syntax on Android - an operating system on mobile - also growth. In 2013, W. Wang and MW. Godfrey has mentioned in their study on the method of detecting Application Programming Interface (API) usage by analyzing abstraction output from compiling source code [24]. In 2018, K.Moran and other colleagues have introduced MDroid+, a mutant testing framework for Android. The research took advantage of AST for organizing profile and to check for the potential fault of that profile based on the output [25]. To now, a lot of studies have been conducted for improving Android security using abstract syntax techniques [26], [27].

B. BACKGROUNDS AND RELATED WORKS ON CODE CLONE DETECTION

Code clone detection methods focus on two type of features of the source code: 1) The similarity in context between two code fragments (Textual similarity) and 2) The similarities in code functions (Function similarity) [1], [28]. There are 3 types of textual similarities and 1 type of function similarity which are:

- Similarity of code fragments with no modification except for white space and comments (Type-1)
- Structural similarity of code fragments with change in identifier names and literal values (Type-2)
- Identical code fragment modifications at statement level (Type-3)
- Code fragment that contain syntactically dissimilar fragments and similarity in function (Type-4)

Until now, researchers have proposed various techniques for detecting code paradigm, each focus on a typical type of feature. In [28], P. Gautam and H. Saini conducted a survey and classified the code detection techniques into 5 types as follows:

- Text/string approach is a textual-based similarity technique that compare between two sequences of text. The technique is unable to normalize and use only for small size inputs.
- The tree-based approaches is an AST-based method that represent the code as syntax tree and use tree-based matching algorithm to find the similar pairs of code fragments.
- The semantic-based which uses the concept of data flow and control flow for detecting code paradigm.
- The syntactic approach is used for detecting function similarity cases by assembling and evaluating the dissimilar metrics from code fragments.
- The lexical or token-based method divides the source code into tokens and provide a signature from a sequence of tokens.

Until now, there are still a lot of discussions and applications with code paradigm and code clone detection. In 2017, L.Liuqing et al. have proposed their clones detection method based on deep learning and AST [29]. In 2018, G.Zhao and J.Huang have proposed deep learning approach for detecting type-4 function similarities in Java [30]. In 2019, B.Lutz et al. have proposed the use of AST aggregation schemes which detects similarities by comparing two RNN outputs from AST inputs [31]. Also, in 2019, A.Qurat U1 and others conducted the systematic study of code clone detection approaches [32]. These ongoing studies have shown that code clone detection is still a topic of great interest.

III. PROPOSED MODEL

In this section, we propose an External-based Identifier Model (EIM) that contains two sub-models: 1) Aggregation model and 2) External-based model for solving the existing
problems in modern abstract syntax techniques using external source code. The proposal can be applied to any programming language as long as we have external sources and can be applied to models that use lexical-based method. This section provides design and algorithms related to EIM models. One of the study cases for our proposed model is the Android environment with AOSP used as external-source. Therefore, this section also describes the algorithms for Android. Because EIM is in fact an extension of token-based AST, we call the EIM solution for Android is Android AST (AAST).

A. EXTERNAL-BASED MODEL

In this sub-section, we introduce the design of a external-based model. Figure 4 illustrates our design for referenced-based model. The design of the EIM model through Figure 4 shows the difference between the traditional method and our proposed model. The base code language is the main language for creating external sources. An example is Java or C++ used as the main language to create the Android framework. An application made from external sources is called an external app.

**FIGURE 4.** Design of external-based model.

The dictionary is generated from one or more reference sources which satisfy one of the following conditions:

- Source contains either packages, class, field or method information
- Sources which contain methods information MUST include a reference to a class and package
- Sources which contain fields information MUST include a reference to a class and package

The format of each identifier in the dictionary is as follow:

```
<package> . <class> . <method|field> @ <M/F> @ <return>
```

According to the format, each of the dictionary values should contain package and class names follow with the name of either the method or field. This indicates the identifier value. The @ symbol is a separator. The type of identifier can be either M as method or F as a field. The last information is the return type of the identifier.

**B. EXTERNAL-BASED MODEL ON ANDROID**

In this section, we introduce the pre-processing step for the Android-based AST model (AAST) which is dictionary generation. To generates a dictionary for Android-based AST, we could extract information in the Android Open-Source Project (AOSP) (such as methods, packages, classes). The reference sources for dictionary can be collected from aosp source frameworks base api/. The pseudo-code for our dictionary generation function described in Algorithm 1. The algorithm name is in the header that starts with **Algorithm number**. The DATA and RESULT shows the input and output values of the algorithm.

**Algorithm 1 Dictionary Generation**

| Data: AOSP_path |
|-----------------|
| Result: AOSP Dictionary |

```python
api = AOSP_path + api.txt ;
init(dictionary);
while readLine(api) do
  indent = getIndent(line);
  if indent == 0 then
    packageName = extractLastString(line);
  end
  if indent == 2 then
    className = extractLastString(line) ;
    return = extractBeforeLastString(line);
  end
  if indent == 4 then
    if “=” in line then
      fieldName = line.split(“=”)[0].extractLastString(line);
    else
      methodName = extractLastString(line);
    end
    dictionary.append(packageName, className, methodName);
  end
end
Return dictionary
```

The algorithm 1 is explained as follows. For each line in the *api.txt* file, we extracted the components (such as package, class, field, and method names) based on their indents. The output of this pseudo-code is a dictionary that can be used in the algorithm 2 and 3.

**C. AGGREGATION MODEL**

This section contains the introduction of the aggregation that consists of the following sub-functions: 1) Android Identifier Extraction and 2) Token Labeling. The token labeling function extracts the dictionary information and implements a logic check for labeling the token. The Android Identifier Extraction function performs a comparison between and
Algorithm 2 Multiple Lexemes Labeling

| Data: | AOSP_path, sourceFile |
|-------|-----------------------|
| Result: | Labelled tokens and lexemes |

\[
\text{dictionary} = \text{dictionaryGeneration}(\text{AOSP_path}); \\
\text{source} = \text{fileOpen(sourceFile)};
\]

\[
\text{while line} = \text{read(source)} \text{ do} \\
\text{ignoreStep} = 0; \\
\text{while word} = \text{readWord(line)} \text{ do} \\
\text{if isIdentifier(word) then} \\
\text{if ignoreStep} > 0 \text{ then} \\
\text{ignoreStep} = 1; \\
\text{continue;} \\
\text{end} \\
\text{token} = \text{Identifier}; \\
\text{check, lexemes, step} = \text{getAndroidIdentifier(word, dictionary)}; \\
\text{if check then} \\
\text{tokenObj} += \text{AndroidIdentifier}; \\
\text{lexemes} += \text{out}; \\
\text{ignoreStep} = \text{step}; \\
\text{end} \\
\text{end} \\
\text{end} \\
\text{Return tokenObj, lexemes}
\]

inputted word and the dictionary values. The logic of both algorithms are described in algorithm 2 and 3.

The algorithm 2 is used for labeling multiple lexemes into one token based on AOSP dictionary. For each line (or n-gram [33]) of code, the isIdentifier function checks whether the word (or lexeme) is an identifier or not. The identifier-type lexeme is then check whether it is an AndroidIdentifier and label those with the AndroidIdentifier token. Since an AndroidIdentifier is a set of tokens, a jump step of value \( n \) is return to tell the function to ignore the next \( n \) lexemes.

The algorithm 3 is an algorithm for extracting Android identifiers. This algorithm is used to filter out the java-based identifier and compare the input word with dictionary values. The algorithm 3 will return true if there is a matched value.

Algorithm 3 Android Identifier Filter Algorithm

\[
\text{Data: Word, Dictionary} \\
\text{Result: Result} \\
\text{while value} = \text{getDictionaryContent(dictionary)} \text{ do} \\
\text{if word in value then} \\
\text{if value.getPos(word) == 0 then} \\
\text{1 continue} \\
\text{end} \\
\text{if value.startwith("java") then} \\
\text{1 continue} \\
\text{end} \\
\text{if value.prevChar !="." then} \\
\text{1 continue} \\
\text{end} \\
\text{Return True} \\
\text{end} \\
\text{Return False}
\]

2) External-based AST with Android source code, also known as Android AST (or AAST)

With the two AST tools, we performed some implementations and evaluated the advantages and disadvantages between external-based and notation-based. The following points were considered during the implementations:

- **Optimization**: The proposed model could provide optimized data while maintaining the same context in comparison to the traditional model. With a high-volume of information, an optimized data could reduce the computation cost.

- **Accuracy**: For Type-1 code clone, when abstracted data should not be confused about the dot and spaces because each different position of the notations will have different functions. The model should provide an accurate conversion rate from original code to abstract representative and vice versa.

- **Practical**: The model should provide better performance in practice than notation-based model. The comparison contains two sub-experiments: 1) The number of signatures and code clone detection rates, 2) Better performance in Code Suggestion (Based on signature checking - a part of the code clone function)

The source code for our implementation is shared at the following links: https://gitlab.com/tu.ngocchau/android_ast

A. PREPARATIONS

The preparation for our implementation is as follows.

1) For analysis tools, we chose javalang which is a notation-based open-source project in Github [34]. javalang is a Java-based AST tool for generating abstract structure context from java source code. We chose simple code clone tools like javalang to help readers understand our ideas through our shared code.

IV. EXPERIMENTS ON EXTERNAL-BASED AND NOTATION-BASED APPROACHES

EIM is a pioneering solution for using external-based enhancements to traditional notation-based solutions. Therefore, comparing the proposed model with other code clone detection tools does not show the effectiveness of the proposed model. Instead, we decided to pick a representative notation-based tool. Based on the notation-based source code, we updated and created a new tool with built-in external-based integration. By that, we have 2 tools:

1) Notation-based AST (or AST)
2) Typically, code clone detection tools must undergo pre-processing to unify the format of the input code stream. However, pre-processing is out of scope for this paper, so we decided to choose a source code with strictly code convention. From that, we decided to use AOSP source code because the source code in AOSP satisfy our requirements on the unify of code convention and coding style. Specifically, we used AOSP version 10 revision 20 (build id: QQ1A.200105.002) to generate our AST dictionary.

During implementing the dictionary generation process, we have collected 9907 java source code files under the aosp/packages/apps of the AOSP folder. The length of the total dataset is more than 1 million unique line of code (LoC) and was divided into 37 small data-frames. Each row of the dataset corresponding to one line of source code. The size of the dictionary that was generated by the AST model is 186641 and AAST model is 190042 accordingly. The execution time for the AOSP source code is 3 hours and 22 minutes (around 10822 seconds). Table 1 provides the summary of data generation process on AOSP and figure 5 illustrates the AAST model.

| Information       | Statistic Value       |
|-------------------|-----------------------|
| Java Files        | 9907                  |
| Unique LoC        | 1149206 (> 1 million) |
| No of Dataframe   | 37                    |
| AST Dictionary    | 186641                |
| AAST Dictionary   | 190042                |
| Execution Time    | 10822 (s)             |

**TABLE 1.** Summary of data generation process.

B. EXPERIMENT 1: OPTIMIZATION

In the first experiment, we made a comparison of the compression rate between notation-based (traditional AST) and external-based Android AST (AAST) outputs. The figure 6 illustrates the difference in the compression rate between the two models.

The figure 6 has shown that AAST provides outputs with higher compact rate than AST, although the optimization rate is not much. To ensure that the context is kept after applying AAST, we generated an abstract tree representative for some of the source code lines. The figure 7 and 8 illustrate the abstract representative of both AST and AAST models. We could see that the lexemes at the third layer were maintained in both models. By looking at the sequence of tokens (the second layer) in both figures, we are able to classify between an Android sequence and a Java-only sequence.

C. EXPERIMENT 2: ACCURACY

For this experiment, we determined the accuracy between models by decoding the textual outputs back into the original code. The decoding outputs were compared with the original code using a hamming distance algorithm. The figure 9 illustrates the distance value in the first 300 lines of code in our dataset. The number in the x-axis is the data frame number and the y-axis is the distant value. To make a clean output, only the different lines are displayed in the figure.
The result of our experiment have shown that AAST provide more similar output with original code than existing AST. By looking further to both outputs of AAST and AST, we figured out the main reason for the similarity results. The main reason for AAST to give better results than AST is due to the concatenated characters such as spaces and dots. By default, each AST lexemes will be appended with a space character when reconnected. However, for dot characters, we cannot determine if that character is used to concatenate namespaces in a package or for other purposes. In that case, using AAST will identify lexeme as a package name and make it easier to join lexeme components with spaces. In this case, the AST can be improved by adding check logic during the concatenation process.

D. EXPERIMENT 3: PRACTICAL
To compare performance between models, we prepared two scripts to simulate two study cases of using AST for analyzing code clone.

1) PERFORMANCE OF CODE CLONE DETECTION
The first study case is the signature-based model using AST. Since AST maintains the code structure, analysts can use AST for detecting the source code that has a similar structure with code in a target application. In this study case, the number of variants of a signature generated by an AST model will reduce the false positive rate in detecting similar code. Figure 10 illustrates the number of unique signatures generated by AST and AAST for 10,000 lines of code. We could see that the AAST generated more signature in comparison to the AST and the number of unique signature increases with more input code.

Further, to compare the effectiveness of traditional method (notation-based) and proposed model (external-based) in code clone detection, with each solution we have implemented lexical analysis and tokens hashing to create signatures. Specifically, from over 30000 different line of code in AOSP source code, we converted each line into sequence of tokens using both AST and AAST tools and hashed sequences into signatures. For a line that has the same sequence with the other different line, we added one value for the signature. The figure 11 and 12 illustrate the experiment results.

The figure 11 and a sample of the output at table 2 show that notation-based approach (using AST tool) has many cases in which a single signatures represent multiple lines of code (one-to-many cases). In the table 2, each row contains two columns which are 1) the type of code and 2) its value. The type of code includes original code, notation-based code, and external-based code. An one-to-many match means that there is many lines of code that encode into single signature. On the other hand, proposed model has a large number of one-to-one match between signature and code line. An one-to-one match means that single signature match with single line of code. The proposed model does not consist of one-to-one cases,
TABLE 2. A sample of the code clone detection.

| Code                      | mBluetoothMonitor.registerListener(mMessengerDelegate) |
|---------------------------|--------------------------------------------------------|
| Notation-based            | context.startForegroundService(startIntent)           |
| External-based            | mBluetoothMonitor.registerListener(mMessengerDelegate) |

there are still one-to-many cases detected in our model as illustrated in figure 12. The figure 12 shows the number of same signature found in random 1000 lines of code.

The experiments on code clone detection proved that our proposed model could provide a more accurate code signature in the source code. More samples of the implementations can be found in our shared URL in section IV.

2) COMPARISON ON CODE SUGGESTION

The second study case is the predictor model. In practice, the AST can also be used for predicting the next word while coding, tools made from this technique are called as code completion. For this experiment, we determine the effectiveness and performance of AST models by the number of times needed for predicting the whole line of code. A representative case for this experiment is when the predictor model predicts a `import` package. Figure 13 illustrates the number of iterations needed for complete 500 line of `import` functions. The AAST model gave less iteration for completing the functions in comparison with traditional AST.

**FIGURE 13. Iteration needed to complete "import" function.**

In figure 13, we only used “import” function as a representative for comparison. It is helpful to know how frequent the AAST situation occurs in other type of code. In order to check for the frequency of code that influenced by AAST model, we chose first 10000 line of code in our dataset. With each code, we check for the iteration different between AST and AAST to find the influenced code that has longer AST iteration. For the next step, we assign each influenced code with one of the following categories:

- **Declaration**: The code that begin with one of the declaration types such as `Private`, `Public`, `Static`, `Boolean`, `Int`, `String`.
- **Import**: The code that begin with an `Import` lexeme.
- **Normal**: The other types of code.

Figure 14 shows the frequency of influenced code in each categories. We could see that although AAST model has the most impact at “import” function, the model still showed its effectiveness in other type of code.

**FIGURE 14. Graph that list the influenced code.**

V. DISCUSSIONS ON THE PROPOSED MODEL

A. PROS AND CONS OF THE MODEL

The proposed external-based model, with experiments in Android, could provide a better optimization, accuracy, and more efficiency in practical than the traditional methods. However, analysts should consider some of the discussion points in this approach:

- Referenced APIs provided for the dictionary needs to be large enough. In fact, the `system_api.txt` and `api.txt` files in our experiments did not cover all APIs and packages in the Android OS (only support Android version 10). Applying algorithm 1 to multiple versions of AOSP source code can solve this problem.

- **Conditions**: The code that begin with a condition operation such as `If`, `Else`, `Switch`, `While`.

- **Declaration**: The code that begin with one of the declaration types such as `Private`, `Public`, `Static`, `Boolean`, `Int`, `String`.

- **Import**: The code that begin with an `Import` lexeme.

- **Normal**: The other types of code.

Figure 14 shows the frequency of influenced code in each categories. We could see that although AAST model has the most impact at “import” function, the model still showed its effectiveness in other type of code.

- **Conditions**: The code that begin with a condition operation such as `If`, `Else`, `Switch`, `While`.

- **Declaration**: The code that begin with one of the declaration types such as `Private`, `Public`, `Static`, `Boolean`, `Int`, `String`.

- **Import**: The code that begin with an `Import` lexeme.

- **Normal**: The other types of code.

Figure 14 shows the frequency of influenced code in each categories. We could see that although AAST model has the most impact at “import” function, the model still showed its effectiveness in other type of code.
VI. CONCLUSION

The approach can be extended to provide a package-oriented identifier instead of Android Identifier. For example, an Android identifier with lexemes content of “android.widget.Toast.makeText” can be identified as AndroidWidgetIdentifier. We provided the source code for AndroidWidgetIdentifier in our shared repository.

In case the analyzed source code contains lexemes that match the information provided in the dictionary, the proposed model might label the lexemes with incorrect tokens. This problem requires additional context-based analysis algorithm which is out-of-topic on this paper.

B. SUMMARY OF THE EXPERIMENTS

The table 3 illustrates the experiment results in which the external-based solution provide better outputs in both accuracy, optimization, and practical metrics compare to the notation-based approach.

VI. CONCLUSION

Code clone detection is a common technique in the study of program analysis and program transformation. Until now, to the best of authors knowledge, the existing code clone detection tools use notation-based (or rule-based) for abstracting the source code content. In this paper, we assume the existence of an external source code that represents a custom (or external) identifier. The introduce of additional identifier provide better insights to the existing tools which helps increase accuracy and performance in finding duplicated code patterns. The experiment results have shown the innovation of our proposed model in comparison with the notation-based model.

REFERENCES

[1] M. White, M. Tufano, C. Vendome, and D. Poshivyanyk, “Deep learning code fragments for code clone detection,” in Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), 2016, pp. 87–98.
[2] C. Ragkhittwetsagul, J. Krinke, and D. Clark, “A comparison of code similarity analysers,” Empirical Softw. Eng., vol. 23, no. 4, pp. 2464–2519, Aug. 2018.
[3] S. Kim, S. Woo, H. Lee, and H. Oh, “VUDDY: A scalable approach for vulnerable code clone discovery,” in Proc. IEEE Symp. Secur. Privacy (SP), May 2017, pp. 595–614.
[4] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, “Do developers update their library dependencies?” Empirical Softw. Eng., vol. 23, no. 1, pp. 384–417, Feb. 2018.
[5] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazijnian, and D. Dig, “Accurate and efficient refactoring detection in commit history,” in Proc. 40th Int. Conf. Softw. Eng., May 2018, pp. 483–494.
[6] J. A. Farrell, Compiler Basics. Reading, MA, USA: Addison-Wesley, Aug. 1995.
[7] B. Chess and G. McGraw, “Static analysis for security,” IEEE Secur. Privacy Mag., vol. 2, no. 6, pp. 76–79, Nov. 2004.
[8] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” in Proc. Int. Conf. Softw. Maintenance, 1998, pp. 368–377.
[9] I. Neamtiu, J. S. Foster, and M. Hicks, “Understanding source code evolution using abstract syntax tree matching,” in Proc. Int. Workshop Mining Softw. Repositories (MSR), 2005, pp. 1–5.
[10] E. Soderberg, T. Ekman, G. Hedin, and E. Magnusson, “Extensible intraprocedural flow analysis at the abstract syntax tree level,” Sci. Comput. Program., vol. 78, no. 10, pp. 1809–1827, Oct. 2013.
[11] R. Pawlak, M. Monperrus, N. Petitjean, C. Nogueira, and L. Seuntjens, “SPOON: A library for implementing analyses and transformations of java source code,” Softw. Pract. Exper., vol. 46, no. 9, pp. 1155–1179, Sep. 2016.
[12] EBNF. Syntax Specification Standard. EBNF, ISO, IEC, Geneva, Switzerland, 1996.
[13] F. Pfening and C. Elliot, “Higher-order abstract syntax,” ACM SIGPLAN Notices, vol. 23, no. 7, pp. 199–208, Jul. 1988.
[14] C. Church, W. Gale, P. Hanks, and D. Hindle, “Using statistics in Lexical Acquisition: Exploiting On-line Resources to Build a Lexicon,” vol. 115. Research Gate and CiteSeerX, 1991, p. 164. [Online]. Available: https://www.researchgate.net/publication/230875926_Use-Statistics_in_Lexical_Acquisition and http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.136.6572
[15] D. Pozza, R. Sisto, L. Durante, and A. Valenzano, “Comparing lexical analysis tools for buffer overflow detection in network software,” in Proc. 1st Int. Conf. Commun. Syst. Softw. Middleware, 2006 pp. 1–7.
[16] G. McGraw, “Automated code review tools for security,” Computer, vol. 41, no. 12, pp. 108–111, Dec. 2008.
[17] M. S. I. Mamun, M. A. Rathore, A. H. Lashkari, N. Stakhovana, and A. A. Ghorbani, “Detecting malicious URLs using lexical analysis,” in Proc. Int. Conf. Netw. Syst. Secur. ( NSS ’05), 2005, pp. 1–5.
[18] L. P. Zhang and D. S. Liu, “AST-based multi-language plagiarism detection method,” in Proc. IEEE 4th Int. Conf. Softw. Eng. Service Sci., May 2013, pp. 738–742.
[19] M. Martinez, L. Duchien, and M. Monperrus, “Automatically extracting instances of code change patterns with AST analysis,” in Proc. IEEE Int. Conf. Softw. Maintenance, Sep. 2013, pp. 388–391.
[20] H. Liang, L. Sun, M. Wang, and Y. Yang, “Deep learning with customized abstract syntax tree for bug localization,” IEEE Access, vol. 7, pp. 116309–116320, 2019.
[21] W. Wang and M. W. Godfrey, “Detecting API usage obstacles: A study of iOS and Android developer questions,” in Proc. 10th Work. Conf. Mining Softw. Repositories (MSR), May 2013, pp. 61–64.

| Experiments | Objective | Outcomes |
|-------------|-----------|----------|
| Optimization | Optimization of data can save computation cost | Proposed model provides better optimization while maintaining the same context |
| Accuracy | Accurate conversion rate from original code to abstract representative and vice versa | Notation-based model provides less accurate rate due to the confusion of concatenated characters such as spaces and dot |
| Practical | Performance and effectiveness in practice | Proposed model provides higher performance in code suggestion and better accurate rate in code clone detection |
[25] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk, “MDroid+: A mutation testing framework for Android,” in Proc. 40th Int. Conf. Softw. Eng., Companion, May 2018, pp. 33–36.

[26] J. L. S. Miguel and S. Takada, “Generating test cases for Android applications through GUI modeling, usage modeling, and change analysis,” in Proc. 8th Int. Conf. Comput. Sci. Softw. Eng. (CSiSE), 2008, pp. 146–147.

[27] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk, “CrashScope: A practical tool for automated testing of Android applications,” in Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C), May 2017, pp. 15–18.

[28] P. Gautam and H. Saini, “Various code clone detection techniques and tools: A comprehensive survey,” in Proc. Int. Conf. Smart Trends Inf. Technol. Comput. Commun. Cham, Switzerland: Springer, 2016, pp. 655–667.

[29] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, “CC Learner: A deep learning-based clone detection approach,” in Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME), Sep. 2017, pp. 249–260.

[30] G. Zhao and J. Huang, “DeepSim: Deep learning code functional similarity,” in Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE), Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 141–151, doi: 10.1145/3236024.3236068.

[31] L. Buch and A. Andrzejak, “Learning-Based recursive aggregation of abstract syntax trees for code clone detection,” in Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reengineering (SANER), Feb. 2019, pp. 95–104.

[32] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, “A systematic review on code clone detection,” IEEE Access, vol. 7, pp. 86121–86144, 2019.

[33] W. B. Cavnar and J. M. Trenkle, “N-gram-based text categorization,” in Proc. 3rd Annu. Symp. Document Anal. Inf. Retr. (SDAIR), vol. 161175, 1994.

[34] C. Thune. (Jul. 2020). c2nes/javalang. [Online]. Available: https://github.com/c2nes/javalang

[35] Codenames, Tags, and Build Numbers | Android OpenSource Project. Accessed: Jul. 6, 2020. [Online]. Available: https://source.android.com

N.-T. Chau, S. Jung: Enhancing Notation-Based Code Cloning Method With an External-Based Identifier Model

NGOC-TU CHAU received the M.S. degree in information and telecommunications and the Ph.D. degree in computer automation and network from Soongsil University, in 2012 and 2019, respectively. Since 2013, he has joined the Communication Network Security Laboratory, Soongsil University, where he is currently a Postdoctoral Researcher. Before joining Soongsil University, he worked for Toshiba Software Development (Vietnam) Company Ltd., for three years, as a Senior Software Engineer (two years), a Team Leader (one year), and subsequently as a network administrator position (one year). His research interests include cloud computing and security as a service. Recently, his works related to mobile security, improving Android malware analysis by combining with machine learning.

SOUHWAN JUNG received the B.S. and M.S. degrees in electronics engineering from Seoul National University, in 1985 and 1987, respectively, and the Ph.D. degree from the University of Washington, Seattle, USA, in 1996.

From 1996 to 1997, he was a Senior Software Engineer with Stellar One Corporation, Bellevue, USA. In 1997, he joined the School of Electronic Engineering, Soongsil University, Seoul, South Korea, where he currently serves as a Professor. He is the Executive Director of the Korea Institute of Information Security and Cryptology. He was also a Research and Development Program Director of the Ministry of Knowledge Economy in Korea for information security area, from 2009 to 2011. He has led the Smart Security Service Research Center funded by Korea Government for six years, since 2012. His current research interests include Android and malware security, cloud security, and the IoT security.

***