Clone Detection Techniques for JavaScript and Language Independence: Review

Danyah Alfageh[1], Hosam Alhakami[2], Abdullah Baz[3], Eisa Alanazi[4], Tahani Alsubait[5]
College of Computer and Information Systems
Umm Al-Qura University, Makkah, Saudi Arabia

***Abstract*—Code clone detection is an active field of study in computer science. Despite its rich history, it lacks focus on web scripting languages. Due to the expansion of web applications and web development amongst developers of varying education and experience levels, they inevitably resort to cloning through out the web. The spread of code clones is further increased by websites like StackOverflow and GitHub. In this paper, we will be focusing on clone detection research done to target clones in JavaScript code and discuss its areas of concern. Also, we will summarize language independent research done and possibility of its application on JavaScript and web applications.**

***Keywords*—Clone detection; code clones; JavaScript; language independent clone detection; web applications**

I. Introduction

Code cloning is one of the most common practices in software development. Over the years many researchers studied the phenomenon and attempted to categorize and solve the problem. Code cloning can cause an issue for software systems as it leads to bug propagation which causes a huge technical debt for stakeholders. Clone detection techniques aim to detect all types of cloned code but differ in their results and coverage ability.

Clones are generally grouped based on their similarity to the source code to four categories as follows [1], [2]:

1) Type-I Exact clones duplicates of the source code except for white space, layout and comments changes.
2) Type-II Renamed clones are similar to the source code both in functionality and syntax and only differ in variables names.
3) Type-III Gapped clones that include type-I and type-II clones but with added or deleted parts of code. They are also referred to as near miss clones.
4) Type-IV Semantic clones or logical clones are only logically similar to source code but differ syntactically.

Clones can be further grouped into two main groups based on their similarity to the source as [3]: clones of textual similarity which includes type-I, type-II and type-III Or functional similarity which applies to type-IV.

Various techniques have been developed to detect any and all of the aforementioned clone types and these techniques fall under the following categories as demonstrated by [4] and [5]:

1) Textual based techniques where the source code is compared to the cloned code to find a sequence of the same text. These techniques best detect type-I clones as they look for exact duplicates.
2) Lexical or token based techniques in which a lexer is used to parse the entire source code into a series of tokens which are scanned to return duplicated tokens. This approach is better as it will not be affected by white spaces and comments, but introduces the possibility of false detection.
3) Tree based techniques where the program is parsed into an AST (Abstract Syntax Tree) then the tree is traversed to find duplicated code.
4) Metric based techniques where the code is parsed into an AST or CFG (Control Flow Graph) on which a number of metrics are calculated to detect clones.
5) Semantic approaches use static program analysis to provide more information regarding logical clones rather than focusing on syntactical similarity.
6) Hybrid approaches provide a mix of the other approaches in attempt to deliver a more accurate detection.

Clone detection accuracy is measured using standard information retrieval metrics specifically precision and recall [3]. Precision measures how well the tool can detect an actual clone. Recall measures the ratio of total number of clones detected by a tool to the actual number of existing clones in the source code.

In this research, we will partially focus on clones in JavaScript which is one of the most used languages in web development. It has also been dominating StackOverflow's most popular programming languages for seven years as shown in their website [6]. Therefore, web developers must inevitably resort to cloning code snippets found in StackOverflow in their daily programming activities. Research done by Ragkhitwetsagul et al. [7] shows that there is a toxic nature of some code snippets found on code sharing sites like StackOverflow such as being out-dated or harmful to the software that they are used in. Furthermore, Baltes and Treude [8] stated that even within StackOverflow, the habit of cloning has even spread through out developer threads i.e. answers to varying posts have been cloned within the platform. Yet, research in areas of clone detection has been lacking in developing tools and techniques specifically geared toward this language. Furthermore, efforts to refactor code clones suffer greatly from the lack of language diversity as noted by Mondal, Roy and Schneider [9] where they have noticed weaknesses in code refactoring closely relating to lack of language diversity. Hence, we will also analyse language independent research done in clone detection and its applicability to JavaScript.

Current uses of JavaScript are unique due to the popularity of libraries and frameworks such as React, Angular, Vue, Node.js and so on. These tools enable developers to create full stack website by only writing a few lines of JavaScript code. These frameworks also come with tons of built-in code such as node modules for Node.js which means if regular code detection is used it will detect such codes and classify them as duplicates between two projects which is a waste of time for a developer looking for a minimum amount of actual clones. Framework and library specific files will never affect the quality of the website and will only waste valuable time in being detected by any clone detection tool. Furthermore, These frameworks allow for the developer to write JavaScript code that will compile into HTML and CSS code. This means there is a necessity to find techniques that help developers to do the following:

- detect duplicate JavaScript code without digging through framework and library files
- detect duplicate JavaScript, HTML and CSS code without having to configure each language specifications.

This paper is structured in the following order. Firstly, we discuss general concepts of clone detection and the necessity of clone detection for JavaScript. Secondly, the related work section discusses related research done in areas of plagiarism detection of source code and cross language detection. Furthermore, we present a literature review which is a summary of the research for JavaScript clones detection. Then, we present a section to exclusively discusses JavaScript specific research and tools. Afterwards, a section for language independent research is presented which is further divided into sub sections based on tool and related research of the tool. Followed by a section of comparison between the tools mentioned in the papers. Lastly, we conclude with a summary of the paper and thoughts for future research.

II. Related Work

In this section we will discuss related works that include levels of language independence which are related to plagiarism detection for university source code assignments. Also, we will discuss a tool for cross-language detection in where clones are detected between different programming languages.

A. Source Code Plagiarism Detection

Similar to clone detection, source code plagiarism detection aims to detect clones in assignments submitted by students. It maintains language independence due to the varied languages taught at universities.

YAP was a tool developed by Wise [10] to detect plagiarised source code submitted by students. It detects multiple languages to support multiple courses. In later versions, it even detects English language. It has two phases, a generation phase in where comments are removed, upper to lower case transformations is applied, removing letters that are not identifiers and removing multiple changes based on student tactics of text modification to generate token files. Then, a comparison phase where the generated pair of token files are compared.

Brixtel et al. [11] developed a language independent framework for plagiarism detection based on clone detection research. They compare plagiarised homework to type-II and type-III clones as students tend to modify the code ever so slightly to camouflage syntax similarity yet maintain same functionality. Their approach sets a threshold of similarity which compares two files at a time and if the result is higher than the threshold it means the code is plagiarised.

Pandit and Toksha [12] have said that future directions in the field are towards machine learning use in plagiarism detection which is a promising direction for improving source code plagiarism detection.

B. Cross-Language Clone Detection

Cross language clone detection aims to detect clones that perform similarly in two different programming languages. A tool named LICCA (Language Independent Code Clone Analyzer) was developed by Raki, Cardozo and Budimac [13] to detect cross language clones. LICCA aims to test source code similar in structure and syntax.

This is possible as source code is transformed using Set of Software Quality Static Analysers (SSQA) introduced by Rakic [14] which produces a unified representation of the source code as enriched Concrete Syntax Tree (eCST) which allows for all languages to be transformed into a unified format. The tree is composed of universal nodes which are nodes that give a grouping to tokens in the syntax tree. An example for a universal node is the universal FUNCTION_DECL which means that it can be a function, a method or a block in the original source code. Comparison of the the trees is based on these nodes.

LICCA is integrated into SSQA and the clone detection is based on the eCST produced by SSQA. LICCA calculates similarity on levels of the syntax tree starting from the higher statement level to lower levels. They compare five programming languages with each having a sample of the same source code Java, JavaScript, C, Modula-2, and Scheme using LICCA. Their research shows many limitation of cross-language comparison but it does provide a unique approach as it is the only one that covers a wide range of languages.

III. Literature Review

Research specifically focusing on clones in JavaScript alone is sparse yet there is plenty of research that includes JavaScript in its analysis. Thung et al. [15] show the degree of cloning of JavaScript code based on source code in Github. They use a tool called Dejavu which detects file level duplication in Github repositories for C++, Java, Python and JavaScript. They found out that the highest duplication amongst the four was found in JavaScript where almost 94% of the dataset being tested turned out to contain duplicates.

Further research on JavaScript and HTML clones was done in a paper by Choi et al. [16] in which they focus on inter-language clones. They found out that the highest co-used languages on Github are HTML and JavaScript. Also, they have found the nature of JavaScript code written inside an HTML file's script tags to be particularly challenging since it will not allow for a straightforward application of a language dependent detection tool.

JavaScript is also a language that can manipulate CSS through jQuery and research on this area is very limited. Yet, Mazinanian [17] has developed a tool to detect CSS clones. However, research on jQuery and JavaScript code clones that manipulate CSS is almost non-existent.

IV. Overview of JavaScript Clone Detection Techniques

Cheung, Ryu and Kim [18] developed JSCD (JavaScript Clone Detector) which is a tool to detect clones in JavaScript. Their research also provides a comprehensive review of tools and research done for clone detection specifically focused on JavaScript. They have segmented their test data to three categories: web applications that include JavaScript and HTML, stand-alone JavaScript applications and libraries and Java systems.

They have found out that web applications have unique characteristics that separate them from stand-alone JavaScript projects as they include clones in HTML, CSS and DOM Manipulation. They have also reported high percentage of clones in JavaScript web segment in particular and they also suggested that lack of studies on the area has led to low assessment of the risks of such phenomenon. Furthermore, they concluded that stand-alone JavaScript projects share further similarity to Java projects rather than with JavaScript used in web development.

JSCD was based on a popular code detection tool called DECKARD demonstrated by Jiang et al. [19]. It parses JavaScript files into ASTs then approximates the resulting structural information into integer vectors that represent the occurrence of a node in the ASTs. It then uses LSH (Locality Sensitive Hashing), a technique which hashes similar vectors into one hash with high probability and distant vectors into a hash with low probability, to cluster similar vectors based on their euclidean distance into clone groups.

In addition, Letic [20] did research which uses JSInspect tool developed by Danielstjules [21] that detects structurally similar code clones in JavaScript code. The tool accepts .js and .jsx files and returns the result in either a JSON (JavaScript Object Notation) or XML (Extensible Markup Language) format. A sample result is shown in Fig. 1 includes the path of the file containing the clones, the number of lines of clones locations and the exact text of the clones.


```
[{
  "id":"6ceb36d5891732db3835c4954d48d1b90368a475",
  "instances":[
    {
      "path":"spec/fixtures/intersection.js",
      "lines":[1,5],
      "code":"function intersectionA(array1, array2) {\n  array1.filter
        (function(n) {\n    return array2.indexOf(n) != -1;\n  });\n}"
    },
    {
      "path":"spec/fixtures/intersection.js",
      "lines":[7,11],
      "code":"function intersectionB(arrayA, arrayB) {\n  arrayA.filter
        (function(n) {\n    return arrayB.indexOf(n) != -1;\n  });\n}"
    }
  ]
}]
```


Fig. 1. Sample result of a comparison by JSInspect tool in [21]

JSInspect uses abstract syntax trees to break the source code to nodes where each node represents a block of the source code. In the results, the paper provides a detailed comparison between the JSInspect tool and another tool called JSCPD (JavaScript Copy/Paste Detector) which is a language independent tool developed by Kucherenko [22] in JavaScript. JSCPD implements Karp and Rabin algorithm [23] to find copy/paste duplicates across 150 programming languages. The algorithm converts a predefined string value to its hash value then iterates through existing strings to match the hash of any of the sub string hashes found in the file.

Data used for testing the tools was taken from three open source projects on Github containing a total 4,1600 and 6977 JavaScript files per project. She noted that both tools perform equally when dealing with a small project but JSCPD had better performance and results with larger projects.

V. Language Independent Approaches

Language independent clone detection tools can detect clones in multiple programming languages. These tools can be helpful for researchers to test on JavaScript as they are not reliant on a single language rules and limitations for clone detection. They manage to abstract their clone detection process based on many techniques and algorithms to fulfil language independence. We will discuss the most popular language independent tools and the languages tested in their research in chronological order.

A. Duploc

Ducasse, Rieger and Demeyer in [24] have introduced a solution that requires no parsing and can detect variety of languages. Firstly, they transform the code very minimally by using basic string manipulation. Then, they use string matching techniques to detect clones but in order to optimize this process they use hashing to store the lines of string in buckets so lines of the same hash are thrown in the same bucket hence eliminating false negatives. Lastly, they provide a visualization of the distribution of the clones using scatter plots.

For their testing, they try four source code projects written in C, Smalltalk, Cobol and Python. They have implemented the algorithm in a tool called Duploc which is developed in SmallTalk and runs on Unix, Mac and Windows. Duploc reads the source code, removes white spaces and comments. Resulting lines of output are then compared with basic string matching algorithm. Duploc produces a matrix to show the source code with matches. Also, it produces a report called map to show exact occurrences per line of duplication in the source code as shown by Rieger and Ducasse[25]. Due to its straightforward approach this tool might provide a good solution to detect JavaScript clones as it is truly language independent.

B. DuDe: Duplication Detector

DuDe (Duplication Detector) developed by Wettel and Marinescu [26] is a tool to detect language independent clones. The tool detects syntactically similar clones and also chunks of similar codes that might be related to the same source. They define what they refer to as “Duplication Chain”, shown in Fig. 2, and is composed of two chunks of similar clones found close to each other in the source code. Furthermore, they breakdown the duplication chain to what they call exact

chunks, which are non-altered duplicate code blocks, that are represented as chunks in a scatter plot matrix in the tool. Also, non-matching gaps are blank areas in the scatter plot occurring between two exact chunks which may indicate they are code lines that have been altered from a larger duplication block. They have specified two characteristics of a duplication chain which are type and signature. The type characteristic defines the type of changes done to the clone code block and is divided into:

1) Exact: the code has suffered no change at all.
2) Modified: the duplication chain is made of exact chunks with modified code lines between them.
3) Insert/Delete: the code chain is linked by insert/delete gaps.
4) Composed: exact chunks linked by various gap types.

Additionally, they define the signature as a characteristic similar to a map that stores the locations of the exact chunks and the non-matching gaps.

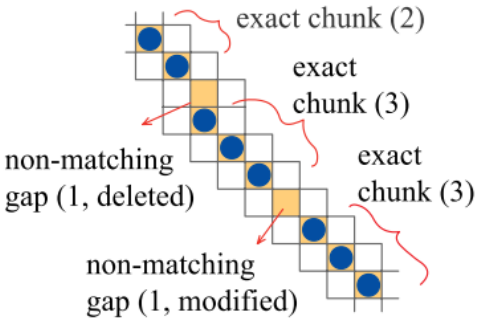

Fig. 2. DuDe duplication chain as shown in [26]

Furthermore, they define the following metrics based on LOC (lines of code) to evaluate their tool:

1) SEC (Size of Exact Chunks) is the length of the duplicate code.
2) LB (Line Bias) the distance between two consecutive exact chunks. The lower the distance is the more likely the space is occupied by modified duplicate code.
3) SDC (Size of Duplication Chain) the size of a meaningful duplication area. It should be equal to SEC in case of exact duplicate chain.

Additionally, thresholds were imposed to manage the proportions of the duplication chain. Firstly, a minimum SDC to ensure the length of the chain is significant. Secondly, a minimum SEC to guarantee exclusion of very small duplicates. Thirdly, a maximum LB to ensure consecutive exact chunks. Finally, minimum SEC should always be greater than maximum LB as it is not preferable to detect chains with gaps longer than exact chunks.

Moreover, they have broken down their detection process into three phases as follows:

1) Code processing: where the original source code files are read to eliminate white spaces and pieces of text that do not affect the detection. This step is language independent as they treat these files as basic strings with no language specific parsing done on them.
2) Populating the scatter-plot: each line of code is compared against the entire project to populate a matrix with the comparison result.
3) Build duplication chains: by looking at the exact chunk of the matrix they attempt to follow along the diagonal to find a pattern to qualify for duplication chain.

They have conducted two experiences one on 4 Java and 4 C projects the other is on an open-source software system called JHotDraw. They have carried out quantitative, qualitative, reliability and scalability validations using their tool.

For the first experiment, they conducted the detection by DuDe based on two configurations for both of those they set the SDC to 7 LOC. For the first configuration they set LB to zero to simulate a detection without duplication chains. As for the second configuration they set the maximum LB and minimum SEC to a threshold of 2 to enable detection of duplication chains. More duplication was detected with the second configuration hence validating the quantitative request.

In a quest to fulfil qualitative validation they carried out the second experiment on JHotDraw where they have tested two configurations one enabled duplication chain and the other did not enable duplication chain. This led to show that when chain detection was enabled they were able to detect 72 clones of varying types vs only 42 clones of exact duplication when chain detection was not enabled.

The percentage of recall of their method is 89% for detecting type-I exact clones. Finally, their tool can process 800,000 LOC over four hours which assures scalability.

C. SDD: Similar Data Detection

Lee and Jeong [27] have introduced SDD (Similar Data Detection) which offers high performance code detection for large software systems. Their algorithm defines a distance measure called n-neighbour distance which represents the distance between two exact chunks of exact clones and based on its length you can judge the type of the clone. Furthermore, they define an inverted index that includes code chunks and their positions, and an index which stores information of the position and corresponding chunk reversely.

Then, a chunk of code can be found easily when tracing adjacent indices by simply looking for it in the inverted index. Then if duplicated indices are found their n-neighbor distance will be evaluated to define a leading chain and duplicated chains are made accordingly. Finally, they have tested their algorithm on five projects of varying languages which are: JDK-1.5 com2, httpd-2.0.54, lucene-1.4.3, Phpwiki-1.2.9 and Ruby-1.8.2. They did not measure recall nor precision for the tool but they state that the tool manages to analyze Java project of size 37.65MB in 67 seconds.

D. PALEX: Parsing Actions and Lexical Information in XML

Another language independent algorithm was presented by Maeda [28] and implemented in a tool called PALEX which

stands for (PArsing actions and LExical information in XML). Their approach works with two independent tools. First, it uses a parser generator such as YACC or Bison to read user-defined syntax rules with action codes to be invoked when the syntax rules are recognized, and they generate LALR parsers. The LALR parser excute two actions: shift and reduce. The LALR parser is built in debug mode and they provide an example of how the parser works by analyzing the arithmetic expression: 1 + 2 * 3. The LALR parser will display debug results to the user output as shown in Fig. 3 which includes state transitions.

```
Reading a token: Next token is token NUM
Shifting token NUM
Entering state 1
Reducing stack by rule 5
Stack now 0 2 6
Entering state 4
Reducing stack by rule 4
Stack now 0 2 6
Entering state 8
Reading a token: Next token is token MLT
Shifting token MLT
Entering state 7
Reading a token: Next token is token NUM
Shifting token NUM
```

Fig. 3. LALR parser resulting debug information [28]

Then when the compiler is finished, it records the list of parsing actions: shift, reduce and reading a token. PALEX then represents these information and additionally includes lexical information such as white spaces and comments in XML. Moreover, this breakdown of the approach into two steps further ensures language independence because PALEX XML elements, which are shown in Fig. 4 below, have no language association. The elements are defined as follows:

- wsc: white space element or comment.
- lex: reading a token.
- sft: shift action.
- rdc: reduce action.
- cst: change from state to state.

The approach was tested on Java, C# and Ruby two of which are dynamically typed languages which further proofs its applicability to JavaScript.

In order to achieve seamless transition from parsing and maintain language integrity bison was modified to write out PALEX code and it is called MoBison in the paper. This provides ability to parse the sorce code to PALEX syntax and transform PALEX XML back to the language syntax.

The clone detection is done using PALEX and implements a suffix-tree matching algorithm. A suffix tree is a data structure that represents suffixes of a string. For example, the string "ABCDABC" will be broken into seven suffixes and the tree will be built in the following order:

- ABCDABC
- BCDABC
- CDABC

```
<?xml version="1.0" encoding="us-ascii"?>
<parseFiles lang="ruby" pg="bison" ver="0.5">
<parse name="ruby.rb">
<rdc st="0" ru="1" />
<cst  fr="0" to="2" />
<lex st="2" tk="tIDENTIFIER" va="include" li="1"
                                        co="7" />
<sft fr="2" to="34" />
<wsc va=" " />
<lex st="34" tk="tCONSTANT" va="Math" li="1"
                                        co="0" />
<rdc st="34" ru="477" />
<cst  fr="2" to="94" />
<rdc st="94" ru="253" />
<cst  fr="94" to="246" />
<sft fr="246" to="38" />
<lex st="38" tk="'\n'" va="&#xA;" li="2" co="1" />
    ...(skip)...
```

Fig. 4. PALEX XML Representation of Ruby code snippet [28]

- DABC
- ABC
- BC
- C

Clones can be detected as sub strings of the tree in the previous example ABC, BC and finally C are clones detected by the algorithm.

E. The NiCad Clone Detector

The NiCad clone detector is a tool developed by Cordy and Roy [29] based on an approach they developed previously and it loosely an acronym of Accurate Detection of Near-Miss Intentional Clones. Their method is broken down to three stages: parsing, normalization and comparison.

- First phase: the source code is parsed to extract fragments based on user specified granularity such as function or block.
- Second phase: the parsed text is then normalized like renaming transformation of the source code.
- Third phase: after the fragments are extracted and normalized they are compared line-wise using a longest common sub-sequence algorithm.

NiCad tool runs through the command line where users can specify the granularity, the language of the source and the path of the directory to be analyzed. At the time of the research the tool only supported function and block granularities and five languages C, C#, Java, Python and WSDL.

However, the tool mainly accomplishes language independence due to its use of TXL which was introduced by Cordy [30] as a programming language designed to transform and manipulate source code. They use it to parse any programming language based on its rules into a normalized text that is then abstracted based on the specified language's rules.

The tool is designed on a plugin based architecture. New languages, normalization rules and granularities can be added easily to customize the tool. It can handle over 60 million lines of code on a single processor computer with only 2 GB of memory.

Their approach is based on text line comparison of lines of code. Similar to many clone detection techniques they define a minimal clone to reduce the the amount of work detectors have to do. Therefore, they use TXL to extract and enumerate potential clones. Extracted clones are only captured once and saved to a text file by the name of their source file and the numbers of the first and last line of the chunk.

Moreover, clones are detected twice if they are inside a parent clone chunk so they will be extracted once alone and with their parent chunk. Throughout extraction extracted clones are stripped of all formatting and comments and are pretty printed by TXL according to adapted grammar rules. Furthermore, TXL's agile parsing explained by Dean et al. [31] allows to control flexibility of pretty printing to be specified for clone detection.

TXL's agile parsing overrides the original grammar to allow for more specific parsing according to each application. Thus, TXL includes a language specific grammar file and with agile parsing it can "override" non terminals with a definition more appropriate to the current task.

Hence, this allows to customize pretty typing, allowing to break the source code into multiple lines and to carry out line by line textual comparison. An example of the grammar of C language is shown in Fig. 5 below.

```
define for_head
    'for ([opt expr]'; [opt expr]'; [opt expr])
end define
```

Fig. 5. TXL grammar definition for for headers in C [32]

Also, by using pretty typing overrides we can break down the grammar and add new lines as shown in Fig. 6 below. This is achieved simply by adding "[NL]" which will add a new line after each part.

```
redefine for_head
     'for(          [NL]
     [opt expr] '; [NL]
     [opt expr] '; [NL]
     [opt expr] )  [NL]
     [statement]
end redefine
```

Fig. 6. TXL grammar override to add new line [32]

The user can further specify the breaking level by changing TXL grammar override by hand to add customized granularities.

Additionally, TXL allows for partial normalization of code which coupled with pretty printing allows NiCad to detect up to 100% of near-miss code clones. Furthermore, TXL allows to choose normalizing only certain parts of a statement, types of statement or levels of nesting. Moreover, selective normalization allows user to normalize the statement if(x <(n + y)) by either normalizing the entire control part so it becomes if(AnyControl) but such normalization will not detect clones in the control part. Also, the same statement can be normalized to maintain the control part as if(id <(id + id)).

It also allows for stricter normalization where only a part of the statement is normalized like if(x <(id + id)) or if(id <(rightControl)) the flexibility in normalization allows the user to choose the level of clones they aim to detect. Furthermore, filtering out code that doesn't include any suspect clones is provided by TXL's agile parsing. Such cases include replacing initialization and declaration with empty lines as they are not significant for detection.

Lastly, after clone extraction and processing it proceeds to apply longest common sub-sequence algorithm to compare the clones line by line. The algorithm in the simplest terms takes two strings A: "SDEXWRL" and B: "CSDRZEKL" and results in C: "SDERL" which represents the shared literals amongst the strings and is obtained by deleting non common literals of the two string. The comparison is done by comparing the processed and broken clones line by line and giving every line a score of 1 in case of similarity and 0 in case of uniqueness.

Then, the percentage of unique items is measured against all items and compared against a clone threshold, which can be set by the user, specified to qualify clones. This threshold is called UPI (Unique Percentage of Items) and it is size sensitive depending on the lines of code in each clone. Subsequently, as longest common sub-sequence algorithm can only compare two clones at a time this means each clone has to be compared with all other clones which leads to high run time. In order to improve the performance comparisons are done based on UPI, if UPI equals 0% this indicates the clones requested to detect are exact clones hence only a cluster of clones of the same size are compared.

However, if UPI is greater than 0% then a clone x is compared to another clone y if size(y) is between of minimum size *size(x) - size(x) * UPIT/100* and maximum size clones *size(x) + size(x) * UPIT/100*. Clones are maintained in a database ordered by their size which makes generating classes for comparison an applicable job. An exemplar of a class is a possible option for optimization i.e. if x and y are similar clones x is considered an exemplar of the two and will go to be compared based on size to find it's class while y will not be compared hence enhancing the overall performance. Overall, the high level of abstraction provided at extraction level with the textual comparison makes NiCad a competent candidate for true language independence and support of any language.

F. Clone Detection Using Fingerprinting

Next, Mythili and Sarala [33] developed a language independent cloning approach based on Rabin-Karp string matching algorithm and fingerprinting. To start they explain fingerprinting as the process of breaking down a document into chunks or k-grams which are then converted into a numeric value making a document's fingerprint.

Moreover, their method first starts be pre-processing the source code in which white spaces, tabs and all extra formatting is removed. Secondly, they look for lexical meaning of text by searching for words in WordNet as introduced by Miller [34], which is a database for English language that links nouns, verbs, adjectives and adverbs to synonym sets linked by semantic relationships based on word definitions,

to find similar words and replace them all with a common name for example two methods *sum(x,y)* and *add(x,y)* will be both identified as clones based on the previous check. Then, the source code variable names and data types are normalized and converted to general format. Next, fingerprint generation is carried out using Karp's [23] algorithm and fingerprints are stored in an array for comparison. After comparison, if fingerprints of the source code and the pattern are similar a character by character comparison is done to the codes associated with the fingerprints. Finally, based on a results matrix lines of duplicate codes will be highlighted if they correspond with a value of 1 in the comparison matrix meaning they are clones.

G. Clone Detection using JSON String Parsing

An approach by Singh, Kaur and Sohal [35] converts all source code into JSON format and compares the resulting JSON files to detect clones. First, the user inputs the source code files to be compared. Second, the code is converted in JSON to remove all formatting and white spaces. Then, string matching is done by using a C# function called equals() to compare the two JSON strings and detect type-I clones. To detect type-II and type-III clones they use Google's "Google Diff Match and Patch Library" in where diff compares lines of texts to return the differences between them, then match looks for the best match of a string. Finally, patch applies the needed fixes and report their success or failure. Furthermore, they apply a size by size comparison of the converted JSON files to determine if they are size clones. Finally, results are displayed graphically based on the calculated percentage of detected clones.

H. CCFinderSW Clone Detector

The CCFinderSW was developed by Semura et al. [36] to fulfill the need for easy addition of multiple programming languages. CCFinderSW is a token based approach that identifies Type-II and Type-III clones. First step is lexical analysis where comments are removed. Then, the source code is tokenized in the following order:

1) Characters and string literals are mapped to a token.
2) White spaces and new lines are delimiters.
3) Each symbol is mapped to a token.
4) The remaining strings are mapped to token.

Next, tokens are transformed by replacing all variable and function names with a common value but reserved words are not replaced. Furthermore, clone detection is carried out using n-gram algorithm. Based on a user defined threshold for the value of n the n-grams are them extracted from the token sequence where CCFinderSW will only detect clones longer than the n threshold. Lastly, results are displayed with lines of detected clone.

1) Multilingual Clone Detection using ANTLR Grammar: The research by Semura et al. [37] extends the CCFinderSW tool to extract lexical information using ANTLR Grammer. The module is added at extraction phase and it uses regular expressions to extract: comments, string literals and reserved words.

VI. Results and Comparison

In this section we will be comparing between the tools mentioned in this paper based on the ability to detect clones and the algorithms most popular amongst them. These comparisons will be divided into the following sections.

1) Classification based on Clone Detection Criteria: The varied tools followed varying comparison measures some did not measure the success rates of their tool. Hence, some of these tools are not included below despite their mention previously because their original paper did not follow any clear quantifiable measures of their tools. In Table I the tools are compared based on Line of Code (LOC), precision, recall and Clone measure of the tool provided in the paper such as lines of clones detected per minute. Also, a final column named notes will include short specific details regarding each entry. The last two columns are added to allow for inclusion of tools that did not measure precision nor recall but provided a clear measure unique to their paper.

Many of the papers do not measure the results of their finding by known clone detection criteria and some do not measure the clone detection at all.

2) Classification of tools based on Approach: All the papers explicitly state what approaches they follow in their detection process. The majority of the tools follow a textual approaches to compare clones as it is a straightforward way to achieve language independence. Most tools use either Rabin-Karp algorithm, basic string matching, N-neigbhor or Longest Common Subsequence (LCS) algorithm. However, JavaScript specific tools JSCD and JSInspect use tree based approaches for detection. PLAEX also uses a tree based approach despite being language independent. Only CCFinderSW uses a lexical approach in its implementation. In Fig. 7 the distribution of the tools is displayed based on their approaches.

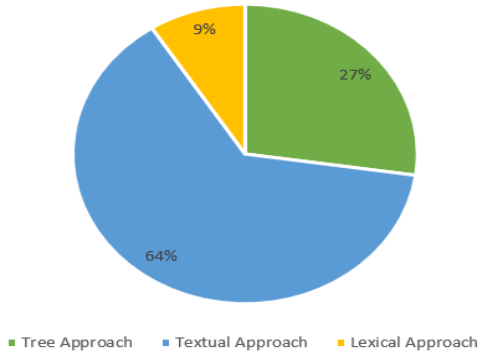

Fig. 7. Tool count by approach

VII. Discussion and Recommendations

In this paper we have compared different tools to detect code duplication. Hence, due to the complexity of modern web applications we are led to believe that straightforward language specific detection is not enough to handle their complexity. Therefore, a language independent approach is the best way to handle such situation.

TABLE I. COMPARISON BASED ON CLONE DETECTION

Tool	LOC	Precision	Recall	Clone Measure	Notes
JSInspect	198.031	-	-	03:49 minutes	threshold = 10 and Project 3
JSCPD	260.195	-	-	03:01 minutes	minLines = 5, minTokens = 5 and Project 3
JSCPD	41,092	-	-	5 -7%	This is for JSWeb for function level clones
JSCPD	582,091	-	-	3 -6%	This is for JSProj for function level clones
Duploc	6500	-	-	17.4%	Case of Message Board in Python
DuDe	148,000	-	89%	-	Case of eclipse-jdtcore project
SDD	245,000	-	-	200 clones found in 28 sec	Case of ruby-1.8.2
PALEX	-	-	-	-	The paper did not provide any measures
NiCad	12,500	-	-	-	The total of the LOC is made of two projects

A solution like NiCad in particular can be excellent as it offers the addition of language semantics to it's language independent structure. The challenge mainly lies in knowing which clones to avoid detecting like node modules in a React application are wasteful to detect as they are part of the framework.

VIII. CONCLUSION

JavaScript is a wildly used and popular programming language and its use is not exclusive to the web. The current status of code clone detection techniques of JavaScript are scarce. Furthermore, reuse of code fragments from the web causes issues that are undetected in the field of clone detection studies.

The majority of the mentioned research papers agree that there is a need to do further research for the language alone. Also, research to understand the consequences of the risks of code cloning on web languages is highly needed. In this paper we have summarized the most prominent JavaScript specific and language independent papers in an attempt to support researchers to get an understanding of the current state of research on JavaScript code clone tools and techniques.

We have also provided a list of language independent tools that have established support and capability to detect a multitude of language independent clones with the most straightforward configurations.

REFERENCES

[1] 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.

[2] A. N. Runwal and A. D. Waghmare, "Code clone detection based on logical similarity: A review," *IEEE Access*, 2017.

[3] C. K. Roy and J. R. Cordy, "Benchmarks for software clone detection: A ten-year retrospective," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 26–37, IEEE, 2018.

[4] K. Solanki and S. Kumari, "Comparative study of software clone detection techniques," in *2016 Management and Innovation Technology International Conference (MITicon)*, pp. MIT–152, IEEE, 2016.

[5] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[6] Stackoverflow, "Developer survey results 2019," Nov 2019.

[7] C. Ragkhitwetsagul, J. Krinke, M. Paixao, G. Bianco, and R. Oliveto, "Toxic code snippets on stack overflow," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 02 2019.

[8] S. Baltes and C. Treude, "Code duplication on stack overflow," *arXiv preprint arXiv:2002.01275*, 2020.

[9] M. Mondal, C. K. Roy, and K. A. Schneider, "A survey on clone refactoring and tracking," *Journal of Systems and Software*, vol. 159, p. 110429, 2020.

[10] M. J. Wise, "Detection of similarities in student programs: Yap'ing may be preferable to plague'ing," in *Acm Sigcse Bulletin*, vol. 24, pp. 268–271, ACM, 1992.

[11] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes, "Language-independent clone detection applied to plagiarism detection," in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pp. 77–86, IEEE, 2010.

[12] A. A. Pandit and G. Toksha, "Review of plagiarism detection technique in source code," in *International Conference on Intelligent Computing and Smart Communication 2019*, pp. 393–405, Springer, 2020.

[13] T. Vislavski, G. Rakić, N. Cardozo, and Z. Budimac, "Licca: A tool for cross-language clone detection," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 512–516, IEEE, 2018.

[14] G. Rakić, *Extendable and Adaptable Framework for Input Language Independent Static Analysis, Novi Sad Faculty of Sciences, University of Novi Sad*. PhD thesis, doctoral dissertation, 2015.

[15] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *2013 IEEE 37th Annual Computer Software and Applications Conference*, pp. 303–312, July 2013.

[16] Y. Nakamura, E. Choi, N. Yoshida, S. Haruna, and K. Inoue, "Towards detection and analysis of interlanguage clones for multilingual web applications.," in *IWSC@ SANER*, pp. 17–18, 2016.

[17] D. Mazinanian, *Eliminating Code Duplication in Cascading Style Sheets*. PhD thesis, Concordia University, 2017.

[18] W. T. Cheung, S. Ryu, and S. Kim, "Development nature matters: An empirical study of code clones in javascript applications," *Empirical Software Engineering*, vol. 21, no. 2, pp. 517–564, 2016.

[19] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*, pp. 96–105, IEEE Computer Society, 2007.

[20] T. Letić, "Detection and analysis of duplicated javascript code usingjsinspect tool," *Zbornik radova Fakulteta tehničkih nauka u Novom Sadu*, vol. 34, no. 06, pp. 1116–1119, 2019.

[21] Danielstjules, "danielstjules/jsinspect," Aug 2017.

[22] Kucherenko, "kucherenko/jscpd," Nov 2019.

[23] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM journal of research and development*, vol. 31, no. 2, pp. 249–260, 1987.

[24] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pp. 109–118, IEEE, 1999.

[25] M. Rieger and S. Ducasse, "Visual detection of duplicated code," in *ECOOP Workshops*, pp. 75–76, Citeseer, 1998.

[26] R. Wettel and R. Marinescu, "Archeology of code duplication: Recovering duplication chains from small duplication fragments," in *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, pp. 8–pp, IEEE, 2005.

[27] S. Lee and I. Jeong, "Sdd: high performance code clone detection system for large scale source code," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 140–141, ACM, 2005.

[28] K. Maeda, "Syntax sensitive and language independent detection of code clones," *World Academy of Science, Engineering and Technology*, vol. 60, pp. 350–354, 2009.

[29] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *2011 IEEE 19th International Conference on Program Comprehension*, pp. 219–220, IEEE, 2011.

[30] J. R. Cordy, "The txl source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[31] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider, "Agile parsing in txl," *Automated Software Engineering*, vol. 10, no. 4, pp. 311–336, 2003.

[32] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th iEEE international conference on program comprehension*, pp. 172–181, IEEE, 2008.

[33] S. Mythili and S. Sarala, "A language independent approach for method level clone detection using fingerprinting.," *International Journal of Advanced Research in Computer Science*, vol. 3, no. 2, 2012.

[34] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[35] G. Singh, S. Kaur, and B. Sohal, "Language independent code clone detection approach using json string parsing," *International Science Press*, 2016.

[36] Y. Semura, N. Yoshida, E. Choi, and K. Inoue, "Ccfindersw: Clone detection tool with flexible multilingual tokenization," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 654–659, IEEE, 2017.

[37] Y. Semura, N. Yoshiday, E. Choi, and K. Inoue, "Multilingual detection of code clones using antlr grammar definitions," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 673–677, IEEE, 2018.