Structural and Nominal Cross-Language Clone Detection

Lawton Nichols[1], Mehmet Emre[1], and Ben Hardekopf[1]

[1]University of California, Santa Barbara, USA
{lawtonnichols,emre,benh}@cs.ucsb.edu

Abstract. In this paper we address the challenge of cross-language clone detection. Due to the rise of cross-language libraries and applications (e.g., apps written for both Android and iPhone), it has become common for code fragments in one language to be ported over into another language in an extension of the usual "copy and paste" coding methodology. As with single-language clones, it is important to be able to detect these cross-language clones. However there are many real-world cross-language clones that existing techniques cannot detect.

We describe the first general, cross-language algorithm that combines both structural and nominal similarity to find syntactic clones, thereby enabling more complete clone detection than any existing technique. This algorithm also performs comparably to the state of the art in single-language clone detection when applied to single-language source code; thus it generalizes the state of the art in clone detection to detect both single- and cross-language clones using one technique.

1 Introduction

The clone detection problem has long been recognized by the community, with many existing papers exploring different techniques for finding clones amongst code written in a single language [11, 16, 17, 22, 23]. However, in recent years an interesting twist has arisen due to the rising popularity of cross-language libraries and applications: *cross-language clones*. Consider the parser generator ANTLR [1], which has runtimes that are written in C#, C++, Go, Java, JavaScript, Python (2 and 3), and Swift. Also consider multi-platform mobile applications, which are often ported between Java and Objective-C or Swift, the languages used by Android and iPhone applications. In these kinds of settings, clones can actually cross language boundaries: a fragment of code in one language can be copied and massaged to conform to the syntax and semantics of another language. Existing single-language clone detection techniques are unable to effectively detect these sorts of cross-language clones. In this paper we propose a method to detect cross-language clones and demonstrate that it (1) finds cross-language clones that no existing method can detect; and (2) performs comparably to existing single-language clone detectors for finding clones within a corpus of single-language code sources. Therefore, our technique generalizes the current state of the art in clone

```javascript
Trees._findAllNodes = function(t, index, findTokens, nodes) {
        // check this node (the root) first
        if(findTokens && (t instanceof TerminalNode)) {
                if(t.symbol.type===index) {
                        nodes.push(t);
                }
        } else if(!findTokens && (t instanceof ParserRuleContext)) {
                if(t.ruleIndex===index) {
                        nodes.push(t);
                }
        }
        // check children
        for(var i=0;i<t.getChildCount();i++) {
                Trees._findAllNodes(t.getChild(i), index, findTokens, nodes);
        }
};
```

```cpp
template<typename T>
static void _findAllNodes(ParseTree *t, size_t index, bool findTokens, std::vector<T> &nodes) {
  // check this node (the root) first
  if (findTokens && is<TerminalNode *>(t)) {
    TerminalNode *tnode = dynamic_cast<TerminalNode *>(t);
    if (tnode->getSymbol()->getType() == index) {
      nodes.push_back(t);
    }
  } else if (!findTokens && is<ParserRuleContext *>(t)) {
    ParserRuleContext *ctx = dynamic_cast<ParserRuleContext *>(t);
    if (ctx->getRuleIndex() == index) {
      nodes.push_back(t);
    }
  }
  // check children
  for (size_t i = 0; i < t->children.size(); i++) {
    _findAllNodes(t->children[i], index, findTokens, nodes);
  }
}
```

Fig. 1: A JavaScript (top) and C++ (bottom) clone pair doing a pre-order search.

```javascript
VerletParticle2D.prototype.setWeight = function(w){
  this.weight = w;
  this.invWeight =
    (w !== 0) ? 1 / w : 0; //avoid divide by zero
};
```

```java
public void setWeight(float w) {
  weight = w;
  invWeight = 1f / w;
}
```

Fig. 2: A JavaScript (left) and Java (right) clone pair setting the weight and inverse weight of a particle in a graphics application. A bug-fix has been applied to the JavaScript clone but not the Java clone.

detection by extending it to allow for both single-language and cross-language clone detection using a single technique.

To make this problem more concrete, consider Figure 1, which shows a real-life case (found during our evaluation described in Section 6) of code clones involving C++ and JavaScript source code from the ANTLR parser generator [1]. To demonstrate the importance of finding cross-language clones, consider Figure 2, which shows another real-life case (also found during our evaluation) of code clones involving JavaScript and Java in which a bug-fix has been applied to one of the clones but not the other. In addition, a quick search of the CVE (Common Vulnerabilities and Exposures) database yields a vulnerability due to incorrect message authentication checking that exists in multiple different language implementations of the relevant code [2].

There are only four existing papers that we are aware of that introduce new techniques for cross-language clone detection (discussed in more detail in Section 2). That initial work has either focused on clones across languages that share a common intermediate representation such as .NET [7,18] or has deviated from classical clone detection and taken a more restricted, natural language-based approach, sometimes relying on assumptions that may not be met in real code [13,14]. None of that existing work would detect the clone examples given in Figures 1 and 2 without extensive modification.

The main reason for these restrictions in previous work is that the *syntactic structure* (i.e., parse trees) of different languages can be extremely different even for code that, at the source level, seems similar. We demonstrate this phenomenon later in this paper. In order to overcome this problem, previous work has either restricted itself to languages with a common intermediate representation (thus enforcing that the syntactic structure is similar for similar code) or abandoned structural matching entirely and looked only at the names of variables and other user-defined abstractions (what we call *nominal* clone detection). We observe that using purely structural or purely nominal matching is sub-optimal in a cross-language setting, in that each can yield both false positives and false negatives.

Our technique consists of (1) a method for enabling structural matching for cross-language clones even in those cases where syntactic structure is different (Section 4); and (2) a method for composing both structural and nominal matching into a singular matcher, maintaining the strengths of each while mitigating their individual weaknesses (Section 5). We have implemented our technique in a tool called FETT[1] that works at the granularity of function pairs; we use FETT to empirically compare our proposed technique against existing techniques (Section 6). We begin by describing related work and background information in Section 2 and giving a high-level overview of our technique in Section 3.

2 Background and Related Work

The concept of clone detection is not new, and the different techniques involved have been surveyed extensively [11, 22]. Most existing non-semantics-based techniques can be categorized into the classes of "structural," "nominal," or "hybrid," which we define below.

Before we begin, there is a bit of misleading terminology in the literature: there exist many clone detection tools that are considered language-generic or language-agnostic (e.g., [23]), but can only be configured to work for programs written in a single language at a time. CCFinder [17], for example, can detect clones for six different programming languages; however, the user cannot (outside of naive text-only modes) truly cross language boundaries during a "language-generic" clone detection phase.

2.1 What Exactly Is a Cross-Language Clone?

Intuitively, we consider a cross-language clone to be the same as any same-language clone—two pieces of code that implement similar functionality—the only difference is the setting. We highlight here what kinds of clones our tool is able to find, and what kinds of clones we include in our evaluation based on their classification (i.e., Type I, II, III or IV [25]).

[1] Our implementation is located at `http://www.cs.ucsb.edu/~pllab` under the "Downloads" link.

The usual code clone hierarchy does not translate well to a cross-language setting: type I and type II clones [25] may not exist across languages because of syntactic differences between languages (e.g., switch statements exist in C but not in Python). In this paper, we present methods that discover syntactic clones modulo the differences in language syntax, and we do this by creating a correspondence between related but different constructs. We do not consider semantic (type IV) clones that implement the same functionality in a different way (e.g., quicksort vs. selection sort). Readers familiar with the standard clone hierarchy can think of the clones that we find as type III clones generalized across languages.

2.2 Structural Program Similarity

Intuitively, two programs (or subprograms) can be considered similar if they look the same, disregarding identifier names—i.e., if their syntax trees have roughly the same shape. We refer to structural clone detection as the process of taking advantage of this similarity.

Same-language clone detection tools usually also consider identifier data, and we are not aware of any purely structural cross-language clone detector. A notable same-language tool that operates via structural similarity is Deckard, which converts syntax trees into vectors for fast comparison [16].

Structural similarity is useful in all settings, but it is a hard problem in a multi-language setting—all the hybrid structural/nominal methods we describe below make some restriction on the languages involved. A major part of the novelty of our technique is a method for purely structural matching across languages (though the final algorithm then combines structural with nominal (i.e., identifier-based) techniques for greater accuracy).

2.3 Nominal Program Similarity

Whereas structural similarity disregards identifiers and instead looks at code shape, nominal similarity does the exact opposite. Nominal similarity relies on the insight that similar code, especially copied and pasted snippets, will have the same identifier names throughout, regardless of code structure.

Notable same-language clone detection tools that operate via nominal similarity are CCFinder and SourcererCC, which compare program tokens [17, 26].

Across Languages. Cheng et al. describe CLCMiner [14], the first cross-language clone detection tool that does not require the languages involved to translate to the same intermediate form. It compares revision histories (diffs) in repository logs for cross-platform C# and Java programs; the tokens inside commits are used to compute similarity scores. CLCMiner is the basis for the Nominal algorithm defined in Section 5.1.

Cheng et al. study a different notion of nominal similarity in [13], where they measure the effectiveness of token distributions in finding clones among cross-platform mobile applications; they obtain a negative result for identifier

names alone. Flores et al. use natural language processing techniques to discover cross language clones at the function level.

2.4 Hybrid Program Similarity

It is logical to combine structural and nominal similarity methods, as the results they provide are complementary. A notable same-language, hybrid clone detection tool is NiCad, which performs its comparisons at the parse tree level [24]. Syntax tree-based comparison is quite common [10, 27].

Tree similarity is computationally expensive [12], and it is more efficient to linearize programs in some way; sequence similarity algorithms can then do the comparison. Existing same-language work compares the tokens in the order in which they appear in the parse tree [15], and we also take advantage of linearization of full parse trees in this work.

Across Languages. Kraft et al. present C2D2 [18], the first cross-language clone detection tool, for C# and Visual Basic programs. This work requires that the languages involved be compiled to the same intermediate representation (IR)—.NET IR in this case. From a graph derived from that IR, they create sequences of tokens for subgraphs and use a Levenshtein distance-based token similarity algorithm to compare them.

Al-Omari et al. build on Kraft et al.'s work and find clones by comparing CIL intermediate code text [7]. Again, they are restricted to .NET languages.

This work. Our method is a hybrid method, works on any language with a grammar definition, and relies on just the source code (in contrast to, e.g., CLCMiner which requires the existence of revision history). We linearize preprocessed parse trees at the function level and compare the linearized sequences in a novel way that generalizes Kraft et al.'s work and incorporates features of Cheng et al.'s work.

2.5 CLCMiner

Our main comparison is with the only tool designed for cross-language clone detection and capable of handling arbitrary languages: CLCMiner [14]. We provide further background on it here. CLCMiner is based on having the source code in a version control system, and requires a revision history by design. Section 5.1 gives a detailed explanation of our adaptation of CLCMiner. The original CLCMiner algorithm works on diffs and lexes them, whereas our version works on function parse trees.

We were not able to obtain access to the original CLCMiner source code from the authors. In order to compare against this method, we implement our own version which adapts CLCMiner to work with the entire text of a function and have it calculate the distance metric above when given a function pair. Our new implementation may perform better or worse than the original (which uses revision history rather than function pairs) in certain cases.

We incorporate CLCMiner's distance metric in a novel way in FETT, and show that our combination of structural and nominal information produces better

results. As we have adapted CLCMiner's algorithm to work on functions instead of diffs, it relies on having a parser to extract the functions and does not rely on a version control system. We refer to our nominal-only adaptation of CLCMiner's algorithm as "Nominal" for the rest of the paper.

3 Overview

In this section we provide a high-level overview of FETT and provide justification for some of our steps. We give an end-to-end example of our clone detection process in Appendix B. FETT's pipeline is:

1. Take as input a corpus of source code (which may exist in multiple languages);
2. Using existing ANTLR grammars, parse and create a separate parse tree for each function (we currently handle C++, Java, and JavaScript);
3. Simplify parse trees that have an unnecessarily large depth;
4. Abstract the multilingual parse trees into a common representation to facilitate comparison;
5. Linearize the resulting trees using a preorder traversal;
6. Compare all linearized function pairs using a Smith-Waterman local sequence alignment algorithm; and finally
7. Present the pairwise similarity scores to the user.

The following sections fill in the details of the structural and nominal aspects of FETT's cross-language clone detection process.

4 Structural Clone Detection

One key insight of our structural algorithm is that *abstract* syntax trees (ASTs), which eliminate details in the concrete parse trees about how exactly the input was parsed or what language it came from, tend to look more similar for similar code even across languages. Unfortunately, ASTs are not part of a language's specification, and AST grammars and formats are implementation dependent. We are not aware of any single compiler that has frontends for the variety of languages that we compare. Our structural clone detection algorithm processes *reduced parse trees* (Section 4.1) to eliminate nonessential details about parsing and obtain a structure similar to ASTs.

Another source of disparity between trees generated by two grammars is that the nonterminals are different. The other key insight of our structural algorithm is that abstracting reduced parse trees by putting nonterminals in *equivalence classes* (Section 4.2) strikes a balance between preserving necessary information and smoothing out differences across languages.

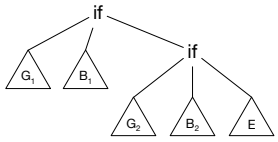
Our structural algorithm proceeds by extracting functions from an abstracted parse tree and then computes similarity scores between functions using the Smith-Waterman local sequence alignment algorithm.

Flattening a tree using a preorder traversal helps smooth out most remaining inconsistencies between inter-language reduced parse trees. To demonstrate the

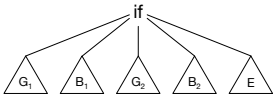
dissimilarities due to grammatical differences that preorder traversal removes, see Figure 3: a grammar that uses nested `if` statements will have a parse tree like Figure 3b, while a grammar that uses unnested `if` statements will look more like Figure 3c. As the `else if` cases become more numerous in the first grammar the nesting becomes more severe, emphasizing the differences in the resulting parse trees.

`if` `(` exp `)` block [*`else` block*] (G1)
`if` exp `:` block [*`elif` exp `:` block*]* [*`else` block*] (G2)

(a) Two different kinds of grammars for if statements.



(b) An example parse tree using the nested `if` grammar (G1).



(c) An example parse tree using the unnested `if` grammar (G2).

Fig. 3: Grammars and parse trees for nested vs. unnested `if` statements.

4.1 Precedence woes

Some grammar definitions encode operator precedence into the grammar[2], whereas others use facilities provided by the parser generators to encode the precedence. Direct encoding of precedence causes spurious chains of nonterminals in the resulting parse tree, which would be removed when the parse tree is converted to an AST. We collapse the chains of nonterminals encountered in a parse tree for the direct encoding case to remove the chains and mitigate this disparity between different styles of grammars. Figure 4 demonstrates the kinds of issues that are apparent when a grammar hard-codes precedence—because precedence in this case appears in the form of nested productions, we always see "AdditiveExpression" even when there is only a multiplication expression present; this will throw off any clone detector that is working directly on plain parse trees.

If precedence is handled indirectly through the parser generator, then the resulting parse tree is much closer to an AST. This is an example of an issue that only arises in a cross-language setting, and which makes cross-language clone detection strictly more difficult than same-language clone detection. We condense any chains of nonterminals, and we refer to the parse trees after this stage as *reduced parse trees*.

4.2 Abstracting Parse Tree Nonterminals

Consider the two reduced parse trees for the expression `binarySearch(array, mid+1, high, x)` in Figures 5a and 5b. Although they

[2] We encountered this only in the C++ grammar during our evaluation.

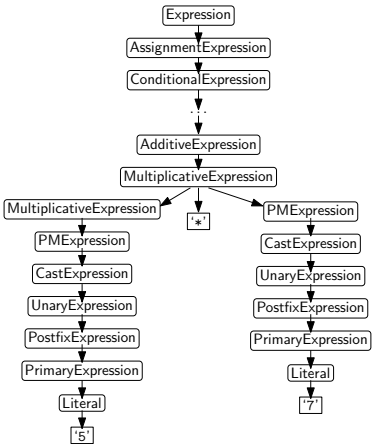
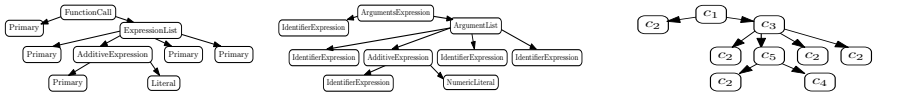


Fig. 4: A subtree of the original C++ parse tree for the text "`5*7`".

look similar to the naked eye, because the node names are different, even a tree edit distance algorithm would say that the trees are not similar at all. We thus need to abstract the nonterminal names while preserving essential information about the tree structure. After performing this abstraction, we call the resulting parse trees *abstracted parse trees*.



(a) Reduced parse tree from a Java parser.

(b) Reduced parse tree from a JavaScript parser.

(c) Abstraction of the trees in Figures 5a and 5b.

Fig. 5: Reduced parse trees for expression `binarySearch(array, mid+1, high, x)` in Java and JavaScript, and their abstraction. The terminals are omitted for simplicity.

Our method instead groups node types with similar meanings across languages, so that node types that "mean" similar things are in the same group. To do this, we *manually* categorize node types into equivalence classes *once per pair of languages*. For example, consider the equivalence classes $c_1 = \{$FunctionCall, ArgumentsExpression$\}$, $c_2 = \{$Primary, IdentifierExpression$\}$, $c_3 = \{$ArgumentList, ExpressionList$\}$, $c_4 = \{$NumericLiteral, Literal$\}$, $c_5 = \{$AdditiveExpression$\}$ and the set $C = \{c_1, c_2, c_3, c_4, c_5\}$. After replacing each node in Figures 5a and 5b with its equivalence class in C, we end up with trees that are exactly the same (Figure 5c). In this specific example the abstracted trees are the same, though this is not always the case in practice.

We define the abstraction algorithm in two parts: $\mathsf{EqClassMapOf}(C)$ produces a map from each node to a symbol corresponding to its equivalence class. $\mathsf{Abstract}(\mathit{tree}, \mathit{map})$ does the abstraction by traversing the given tree bottom up and applying the map. It removes the *nonterminals* which do not belong

to any equivalence class. When the abstraction algorithm removes a node, it connects any children of the removed node to the removed node's parent.

4.3 Sequence Alignment for Clone Detection

Linearizing the trees via a preorder traversal of the nodes will remove most traces of the structural differences demonstrated in Figure 3. Moreover, the state of the art tree edit distance algorithms are not as scalable as sequence alignment algorithms[3]. These observations led us to explore sequence alignment algorithms as an alternative to tree-edit distance. Levenshtein distance is a popular choice in this category. Smith-Waterman is strictly more general than Levenshtein distance, and it supports assigning weights to different elements in the sequence. Hence, we use the Smith-Waterman algorithm on preordered trees to compute similarity scores. We evaluate the precision and recall of both Smith-Waterman and tree edit distance in Section 6 and observe that sequence alignment performs better in terms of precision and scalability.

We convert function subtrees to sequences by computing the preorder traversal. Finally, we execute Smith-Waterman using custom weights on each sequence pair and normalize the resulting score using the normalization factor Z described below. We chose the weights based on the hypothesis that certain nodes like conditionals indicate important program structure, and should generally appear in the same order in a cloned pair of functions; therefore, we assign higher weights to penalize the function pairs in which this alignment does not occur. In the algorithm, the function $\mathsf{SmithWaterman}(a, b, M, g)$ computes a similarity score between two sequences a and b using the Smith-Waterman algorithm with substitution matrix M and linear gap penalty coefficient g; a detailed explanation of these parameters can be found in [9].

Normalizing Smith-Waterman results. The result of the Smith-Waterman algorithm depends on the size of the input, and longer sequence pairs have higher scores. In order to find both short and long clones, we normalize the resulting similarity score from the Smith-Waterman algorithm to neutralize the bias towards longer clones.

We define the *self-similarity score* of a sequence a as the score assigned to the pair (a, a) by the *unnormalized* Smith-Waterman algorithm; denote this score $\mathsf{S}(a)$. We normalize score assigned to a pair (a, b) by $\frac{1}{Z}$ where $Z = \max\{\mathsf{S}(a), \mathsf{S}(b)\}$. Note that Z is an upper bound for the score obtained by Smith-Waterman, and the score is equal to Z if and only if $a = b$. Thus, using the normalization factor $\frac{1}{Z}$ is useful if one is looking for similar whole functions rather than looking for a small snippet in a larger piece of code.

[3] APTED, the state of the art tree edit distance algorithm has a time complexity of $O(n^3)$ [21] whereas the variant of Smith-Waterman algorithm we use is $O(n^2)$ [9].

5 Hybrid Algorithm

Combining nominal and structural clone detection in a cross-language setting provides the best of both worlds, and mitigates any issues that running just one detection method might have.

Identifier names carry some meaning about the programmer intent and give a code snippet context. On the other hand, structure of code (conditionals, loops, function calls etc.) also carry information about programmer intent. Without this structural information, we might misidentify two pieces of code as clones. Our hybrid algorithm is guided by structural information while consulting the Nominal algorithm to use local context within structurally similar pieces of code.

5.1 Our Nominal Algorithm

We have adapted CLCMiner's algorithm to work on functions as our purely Nominal algorithm. For a given pair of functions (f_1, f_2), our nominal matching algorithm consists of two parts.

The first part takes a function f, removes the comments and splits the tokens on each non-letter character (such as underscores or dashes). It then splits the camel case tokens into words and converts them to lowercase—each function becomes a bag of words that is represented by a characteristic vector, which holds the number of occurrences of each word. We denote the resulting characteristic vector as $\mathsf{v}(f)$.

The second part of the algorithm computes a normalized distance between the two characteristic vectors v_1, v_2 according to the formula $\mathrm{d}(v_1, v_2) = \frac{\|v_1 - v_2\|_1}{\|v_1\|_1 + \|v_2\|_1}$ where $\|\cdot\|_1$ is the ℓ_1 norm (i.e., the sum of the absolute values of every entry in the vector). This algorithm computes a distance between two given functions; to make it comparable to the other algorithms, we use $1 - \mathrm{d}(v_1, v_2)$ as a similarity score.

5.2 Full Algorithm

Our full algorithm is shown in Appendix A. It is a combination of the structural and nominal algorithms: we linearize the parse trees, and consecutive terminal nodes become bags of words. Nonterminals are compared using our structural method, and bags of words are compared using our nominal method.

6 Evaluation

In this section we compare our work against existing work on both cross-language and same-language clone detection.

6.1 Implementation and Environment

We have implemented our tool FETT in Scala and used the ANTLR parser framework as its front end, so that any language with an ANTLR grammar can be easily connected.

To test whether FETT can handle same-language clone detection with similar accuracy as specialized, language-specific tools, we configured NiCad 4.0 [24] to work at the function-level granularity and experimented with configurations until we found the best-performing one for our tests[4].

Because we are comparing parse *trees*, we also want to determine how well we compete against the state-of-the-art tree edit distance algorithms, thus we compare one data set with APTED [20, 21]. We normalize the similarities using the method described in [19], and, as this normalization method requires a metric distance, we could not introduce weights for matches. We can still weight mismatches, though. We found that the parameters $mismatch = 1$, $deletion = insertion = 5$, $match = 0$ gave us the best results overall.

We chose the threshold for ignored functions (defined in Section 4.3) to be $\theta = 35$ for every experiment, and the exact tolerance parameters are given below for each case. We used the same set of equivalence classes with the same weights for all cases: conditional, loop, return, and function call were all weighted 5; assignments were weighted 2; and all other considered nodes were weighted 1.

Our experiments were run on a computer with an Intel i7 4790 3.6 GHz processor. FETT, Structural, Tree Edit Distance, and Nominal were given 8 GB maximum heap size and were set to use 4 threads.

6.2 Methodology

We used the standard statistical metrics of precision, recall, and F-measure to quantitatively assess the effectiveness of our different techniques.

Due to the sheer amount of possible clone candidates in large projects, it is difficult to manually obtain complete ground truth for clones in real-world programs. Hence, we created two separate data sets for evaluation:

Manual programs set (handwritten set). We implemented a set of small programs in different languages to create a setting in which we have complete knowledge of whether a pair of functions are clones. Statistics about the code are in Table 1.

Randomly sampled program set (large set). We chose four libraries that have implementations in different languages and set the tolerance parameters[5]

[4] NiCad: threshold=0.5, minsize=4, maxsize=2500, rename=blind, filter=none, abstract=none, normalize=none

[5] For FETT: $\mu = 6$ (match coefficient) and $g = -4$ (gap penalty) for the case of comparing Java and JavaScript, and $(\mu, g) = (9, -1)$ for Java/C++ and JavaScript/C++, and $(8, -3)$ for Java/Java. The nominal multiplier was set to 2 for all but the Java/C++ and JavaScript/C++ cases, where it was set to 3. For the Structural algorithm: $(7, -1)$ for JavaScript/Java, $(8, -4)$ for Java/C++, $(0.5, -2)$ for Java/Java, and $(9, -4)$ for JavaScript/C++.

Table 1: Statistics of handwritten clones.

Language Pair	LoC	#Functions	#Pairs	#Clones
Java JavaScript	201 177	12 11	132	11
Java C++	201 195	12 12	144	12
JavaScript C++	177 195	11 12	132	11

defined in Algorithm 1 to give the best results on a per-language pair basis. We randomly sampled functions from the files with the same names (ignoring extensions) and manually checked the pairs to create a sample with ground truth—this is essentially the sampling strategy used by Cheng et al. [14] applied to functions instead of diffs. We chose to reuse this sampling strategy due to the manual nature of our evaluation, and because we only possess finite human resources; it does not reflect the true distribution of clones, as function clone pairs are unlikely to be chosen in a standard uniform random sample—had we gone that route, our precision and recall scores would not have been meaningful. We are not aware of a better solution to this problem.

The first three libraries considered for this set are: the ANTLR parser framework, version 4 [1]; the toxiclibs computational design library [5]; and the ZXing barcode image processing library [6]. We also considered two ports of the LAME MP3 encoding library in different languages that were ported by different developers to assess the efficacy of clone detection tools in such a scenario: lamejs, a JavaScript port [4]; and java-lame, a Java port [3]. Statistics about the libraries are in Table 2.

Table 2: Statistics of libraries considered for evaluation. LoC: non-blank non-comment lines of code, Fun's: # of functions found in each project, Nont'l (Nontrivial) Fun's: # of functions whose reduced parse trees are $> \theta$ (the chosen threshold), Pairs: the # of possible fun. pairs, Same-File Pairs: # of pairs of functions coming from files with the same name (ignoring extensions), Sel'd: # of selected pairs, Runtime: total time (H:M:S) to run our method.

Data set	Library	Lang. Pair	LoC	Fun's	Nont'l Fun's	Pairs	Same-File Pairs	Sel'd	Runtime	Clones
antlrj	ANTLR	Java Java	13,770 13,770	1,393 1,393	694 694	240,471	4,942	505	0:56:18	14
antlrjsj	ANTLR	Java JavaScript	13,770 7,323	1,393 728	694 405	281,070	6,240	663	0:25:01	45
antlrcppjs	ANTLR	C++ JavaScript	15,766 7,323	1,222 728	480 405	194,400	3,762	752	0:17:11	17
toxic	toxiclibs	Java JavaScript	36,178 36,976	3,734 4,108	2,156 2,321	5,004,076	11,637	1,060	3:01:12	63
zxing	ZXing	Java C++	38,968 22,784	2,659 866	1,689 405	684,045	1,388	254	2:10:51	45
lame	java-lame lamejs	Java JavaScript	20,950 11,112	575 285	436 232	101,152	4,645	873	0:27:37	34

6.3 Results

For our main set of tests, we compare FETT against (1) our purely Structural algorithm (i.e., no token similarity), and (2) our Nominal algorithm. We also apply the APTED tree edit distance algorithm combined with our abstraction method on our handwritten data set; tree edit distance takes at least an order of magnitude longer than the other tools, and we did not evaluate the large data set using tree edit distance because of this and due to its poor performance on the handwritten tests. We use NiCad on the Java-Java same-language case of our large data set.

Cumulative clone ratios. We look at the graphs of cumulative clone distributions to choose a good cut-off point for each of the three techniques. These graphs were originally used in [14], and they are meant to give an intuition about where a clone detector separates clones from non-clones.

Similarity vs. cumulative clone ratio graphs track the ratio of clones to non-clones as the similarity score varies from 1.0 to 0. For example, at point 0.4 on the similarity axis, we plot the ratio of clones to non-clones of all samples with similarity scores > 0.4. A successful clone detector would have a similarity value at which there is a significant drop in this ratio, and that would create the optimal cutoff point. A clone detector may not assign very high scores to any pairs based on its similarity metric; in such cases, we start the plot from the first nonempty bin. Figure 7 shows the cumulative clone ratios for antlrj and toxic; graphs of other test cases are omitted because of space constraints, but they are of similar overall shape. We chose a cutoff point for each clone detector based on the drops from these graphs (e.g. we chose the cutoff point of 0.4 for FETT's Java/Java case). The relative shape of the graph is more important than absolute scores—squishing or stretching the similarity scores only affects the choice of the optimal cutoff point.

Handwritten test set. When evaluating the manually created (handwritten) data set, we used the same parameters $\mu = 7$, $g = -2$ overall for all pairs of functions in the data set and considered the combined results for both FETT and the Structural algorithm. FETT had its nominal multiplier set to 2. Figure 6 shows the clone distributions of different clone detection methods for the handwritten program set; and precision, recall, and F-measure (harmonic mean of precision and recall) for this set are given in Table 3. FETT and the Structural algorithm had a cutoff of 0.5, and the Nominal algorithm's cutoff was 0.6.

Handwritten test set discussion. The table and the figures paint a similar picture. Both FETT and the Structural algorithm seem to perform the best on this data set—the graphs for the higher similarity scores have a high clone ratio, and there is a sharp decline visible in both graphs as the similarity score is allowed to lower. The Nominal algorithm has a less sharp drop, and this indicates that it is assigning mid-range similarity scores with low precision. It is also notable that tree edit distance does so poorly; we believe that this is because we are not allowed to give weights to matches, as described above.

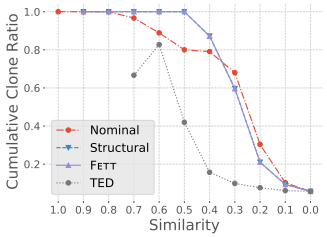


Fig. 6: Cumulative clone ratio distribution for handwritten programs. Results of FETT and structural coincide.

Table 3: Precision, recall, and F-measure for handwritten program set.

Data set	Method	Precision	Recall	F-measure
Handwritten	FETT	1.000	0.970	0.985
	Structural	1.000	0.970	0.985
	Nominal	0.886	0.939	0.912
	Tree Edit Dist.	0.821	0.697	0.754

Large test set. We now present and discuss all the cross-language results for our large test set. The same-language case is different from the cross-language cases, so the reader is asked to consult Figure 7b, which is indicative of all the cross-language cases, and not Figure 7a.

Cutoffs were chosen on a per-language pair basis that maximized a given tool's score. For FETT, for the three JavaScript/Java test cases and the Java/C++ test case, we used a cutoff of 0.4, and the rest used a cutoff of 0.5. For the Structural algorithm, we used a cutoff of 0.6 for JavaScript/Java, 0.5 for Java/C++ and JavaScript/C++, and 0.4 for Java/Java. For the Nominal algorithm, we used a cutoff of 0.5 for JavaScript/C++, and 0.6 for the rest.

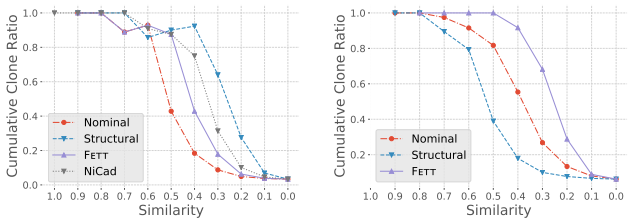
Figure 8 shows precision, recall and F-measure of all the tools we compared for each data set and provides a visual and quantitative assessment of efficacy of all the techniques.

Large test set discussion. Clone ratios relate most closely to the precision scores for each data set, and from the results it appears that the Structural algorithm generally has the upper hand in this area—applying the intuition described above, we see that the Structural algorithm seems to cut off at the sharpest angle in most cases. It makes sense why this is the case, as pieces of code that look similar across languages are generally prime candidates for clones.

Precision is of course not the whole story. It is clear that FETT is able to take the best of both the nominal and structural worlds, and the F-measure is always the highest. When it comes to Structural's results, the toxiclibs case is an outlier, where we found that there were more cases of the structural differences; FETT's hybrid structural/nominal algorithm was able to make up for this, though.

Same-language test case. To assess performance on same-language clones, we compared our tool with NiCad on the Java version of ANTLR. Returning to the same figures, the antlrj case is quite similar to the other language pairs in terms of precision, recall, and F-measure, which demonstrates that our tool is capable of holding its ground in a same-language setting.

FETT performs slightly worse (by one percentage point in terms of F-measure) than NiCad. This result is not surprising because NiCad uses more information about the code whereas we deliberately discard some information by abstracting parse trees to work in a cross-language setting. Even with our filtering of parse



(a) antlrj (b) toxic

Fig. 7: Similarity vs. cumulative clone ratio for the samples from the large open-source program set.

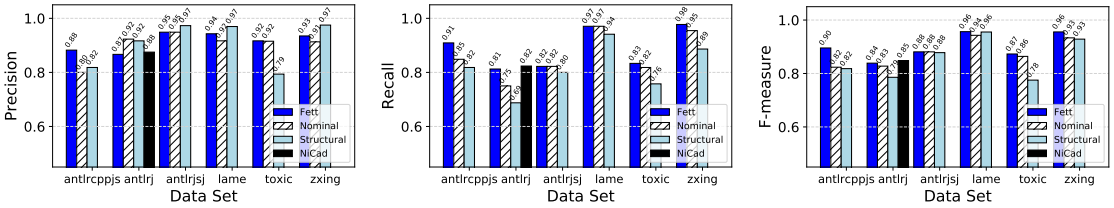


Fig. 8: Precision, recall and F-measure of clone detection tools on the large program set.

trees, Fett's F-measure score is very close, and this shows that our tool is capable of producing similar results to a dedicated same-language tool.

Overall results. We observe that the Fett's hybrid algorithm, in terms of F-measure, outperforms both the Nominal algorithm and the Structural algorithm consistently in our large test set experiments.

Limitations. Fett may have difficulty scaling to repositories with large numbers of large functions—a run of Fett on the entire toxiclibs library (comparing every function pair, not just same file pairs) takes 5.13 hours—and so further improvements will be required to enable such a target. One possible future direction for improvements could be to develop semi-automated solutions where we have the user use her domain knowledge and pick out the files or functions to compare beforehand, or the user can prune the search space by telling the tool which modules are unrelated.

7 Conclusion

We have presented Fett, a hybrid structural/nominal clone detection method that is capable of operating across programming languages and that is generic in the sense that it does not require any languages involved to belong to the same language family. It is syntax-based, uses ready-made grammar specifications, and requires minimal manual effort—the keys to the process are syntax abstraction and sequence alignment. We have provided a two-part evaluation of Fett, and we empirically demonstrate on multiple test sets that Fett is accurate in terms of the standard metrics of precision and recall. We also confirm that our method is on a par with previous work when it comes to same-language clone detection, thus proving that it is strictly more general than single-language methods.

Acknowledgments

This work was supported by NSF CCF-1319060.

References

1. ANTLR (2017), http://www.antlr.org/
2. Cve-2013-1624 : The tls implementation in the bouncy castle java library before 1.48 and c# library before 1.8 does not properly conside (2017), http://www.cvedetails.com/cve/CVE-2013-1624/
3. java-lame (2017), https://github.com/nwaldispuehl/java-lame
4. lamejs (2017), https://github.com/zhuker/lamejs
5. toxiclibs (2017), http://toxiclibs.org/
6. Zxing (2017), https://github.com/zxing/zxing
7. Al-Omari, F., Keivanloo, I., Roy, C.K., Rilling, J.: Detecting clones across microsoft .net programming languages. In: 19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012. pp. 405–414 (2012). https://doi.org/10.1109/WCRE.2012.50, http://dx.doi.org/10.1109/WCRE.2012.50
8. Altschul, S.F.: Global and local sequence alignment. Lecture Notes (2011)
9. Altschul, S.F., Erickson, B.W.: Optimal sequence alignment using affine gap costs. Bulletin of Mathematical Biology **48**(5), 603–616 (1986). https://doi.org/10.1007/BF02462326, http://dx.doi.org/10.1007/BF02462326
10. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Software Maintenance, 1998. Proceedings., International Conference on. pp. 368–377. IEEE (1998)
11. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Transactions on software engineering **33**(9) (2007)
12. Bille, P.: A survey on tree edit distance and related problems. Theor. Comput. Sci. **337**(1-3), 217–239 (Jun 2005). https://doi.org/10.1016/j.tcs.2004.12.030, http://dx.doi.org/10.1016/j.tcs.2004.12.030
13. Cheng, X., Jiang, L., Zhong, H., Yu, H., Zhao, J.: On the feasibility of detecting cross-platform code clones via identifier similarity. In: Proceedings of the 5th International Workshop on Software Mining. pp. 39–42. ACM (2016)
14. Cheng, X., Peng, Z., Jiang, L., Zhong, H., Yu, H., Zhao, J.: Mining revision histories to detect cross-language clones without intermediates. In: Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on. pp. 696–701. IEEE (2016)
15. Gitchell, D., Tran, N.: Sim: a utility for detecting similarity in computer programs. In: ACM SIGCSE Bulletin. vol. 31, pp. 266–270. ACM (1999)
16. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: 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)
17. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28**(7), 654–670 (2002)
18. Kraft, N.A., Bonds, B.W., Smith, R.K.: Cross-language clone detection. In: SEKE. pp. 54–59 (2008)

19. Li, Y., Chenguang, Z.: A metric normalization of tree edit distance. Frontiers of Computer Science in China **5**(1), 119–125 (2011). https://doi.org/10.1007/s11704-011-9336-2, http://dx.doi.org/10.1007/s11704-011-9336-2
20. Pawlik, M., Augsten, N.: Efficient computation of the tree edit distance. ACM Trans. Database Syst. **40**(1), 3:1–3:40 (2015). https://doi.org/10.1145/2699485, http://doi.acm.org/10.1145/2699485
21. Pawlik, M., Augsten, N.: Tree edit distance: Robust and memory-efficient. Inf. Syst. **56**, 157–173 (2016). https://doi.org/10.1016/j.is.2015.08.004, http://dx.doi.org/10.1016/j.is.2015.08.004
22. Rattan, D., Bhatia, R., Singh, M.: Software clone detection: A systematic review. Information and Software Technology **55**(7), 1165–1199 (2013)
23. Rieger, M.: Effective clone detection without language barriers. Ph.D. thesis, University of Bern (2005)
24. Roy, C.K., Cordy, J.R.: 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 (June 2008). https://doi.org/10.1109/ICPC.2008.41
25. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Tech. Rep. 541, Queen's School of Computing (2007)
26. Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., Lopes, C.V.: Sourcerercc: Scaling code clone detection to big-code. In: Proceedings of the 38th International Conference on Software Engineering. pp. 1157–1168. ICSE '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2884781.2884877, http://doi.acm.org/10.1145/2884781.2884877
27. Yang, W.: Identifying syntactic differences between two programs. Software: Practice and Experience **21**(7), 739–755 (1991)

A The Full Algorithm

The hybrid algorithm shown in Algorithm 1 works mostly the same as the structural algorithm as described in Section 4.3, but on a tree with terminals with two differences: (1) in the hybrid algorithm, the Preprocess function also merges consecutive terminals and converts terminals into bags of words, and (2) whenever the hybrid algorithm is comparing two bags of words, it runs the nominal algorithm on the two terminals to compute the similarity between them; then, it multiplies this score with the nominal multiplier ν.

This method for introducing nominal information works at a finer granularity than our Nominal algorithm, and the token information is added in order (so the results are still highly structural); thus it should be less prone to false positives if the same tokens appear out of order. The nominal multiplier exists for tuning purposes, just like the other parameters, and intuitively it should be lower when an implementation has more name mismatches (e.g., for language pairs where the standard libraries look vastly different, or in an educational setting to check student code duplication, a smaller multiplier would be more suitable).

Algorithm 1 Algorithm to find cross-language function clones.

procedure SIMILARITY($s, t, C, W, \mu, g, \theta$)
input:
Two sets of parse trees s and t, a set of equivalence classes between nodes C, an equivalence class weight function W, a match coefficient μ to control overall tolerance, a gap penalty coefficient g, a threshold θ on function subtree size
output:
A similarity score map S from pairs of functions to scores between 0 and 1.
$m \leftarrow \mathsf{NodeToEqClassMap}(C)$
$\hat{s} \leftarrow \{\mathsf{Abstract}(\mathsf{Preprocess}(\tau), m) \mid \tau \in s\}$
$\hat{t} \leftarrow \{\mathsf{Abstract}(\mathsf{Preprocess}(\tau), m) \mid \tau \in t\}$
$S \leftarrow \mathsf{EmptyMap}$
for all $f \in \{\mathsf{Functions}(\tau) \mid \tau \in \hat{s} \wedge |f| \geq \theta\}$ **do**
  for all $g \in \{\mathsf{Functions}(\tau) \mid \tau \in \hat{t} \wedge |g| \geq \theta\}$ **do**
    $a \leftarrow \mathsf{Preorder}(f)$
    $b \leftarrow \mathsf{Preorder}(g)$
    $M \leftarrow \lambda i, j. \begin{cases} \nu\,(1 - \mathrm{d}(\mathsf{v}(i), \mathsf{v}(j))) & i, j \text{ are bags of words} \\ \mu W(i), & i = j \\ -\max(W(i), W(j)), & i \neq j \end{cases}$
    Compute Z according to Section 4.3
    $S \leftarrow S[(f, g) \mapsto \frac{\mathsf{SmithWaterman}_{(a,b,M,g)}}{Z}]$
  end for
end for
return S
end procedure

B End-to-End Example

B.1 Parsing Is Such Sweet Sorrow

For our running example, consider the two functions in Figure 9. Both are implementations of binary search (one in C++, the other in Java), and they have some structural, nominal, and semantic differences.

```cpp
template <class T>
int binsearch(const T array[], int left, int right, T what) {
  if (right < left) return -1;
  int mid = (right + left) / 2;
  if (array[mid] > what)
    return binsearch(array, left, mid-1, what);
  if (array[mid] < what) {
    return binsearch(array, mid+1, right, what);
  }
  return mid;
}
```

(a) A generic C++ binary search implementation.

```java
public static int binarySearch(int[] nums, int check, int low, int high) {
  if (high < low) return -1;
  int center = (high + low) / 2;
  if (array[center] > check)
    return binarySearch(nums, check, low, center-1);
  else if (array[center] < check)
    return binarySearch(array, check, center+1, high);
  else
    return center;
}
```

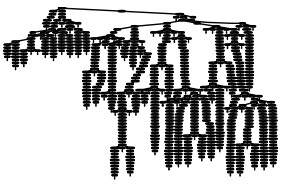
(b) A non-generic Java binary search implementation.

Fig. 9: Two cross-language clones for binary search.

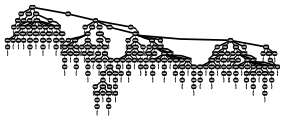
We would like to systematically figure out that these two functions are indeed similar—to do so we must get them into a common form. We begin by parsing them.

B.2 A Tale of Two Parse Trees

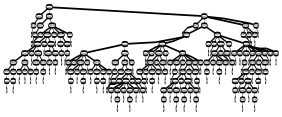
We use the ANTLR parser generator and open-source grammar definitions to parse the two files and generate concrete parse trees. The original parse trees generated by ANTLR are shown in Figures 10a and 10b.



(a) Original C++ parse tree for the code in Figure 9a.



(b) Original Java parse tree for the code in Figure 9b.



(c) Reduced C++ parse tree obtained from the parse tree in Figure 10a.

Fig. 10: Different forms of parse trees for the code snippets in Figure 9.

At a glance, it is obvious that the parse trees are vastly different; even though the two functions look similar textually, the parse trees do not reflect this. Our goal is to make these parse trees look more similar.

Some ANTLR grammars use hard-coded precedence—i.e., they have explicit precedence levels defined using nonterminals. A common example of this is the standard arithmetic expression grammar:

$$\begin{aligned} \mathit{AddE} &::= \mathit{AddE} + \mathit{Term} \\ &\mid \mathit{AddE} - \mathit{Term} \\ &\mid \mathit{MulE} \end{aligned} \qquad \begin{aligned} \mathit{MulE} &::= \mathit{MulE} * \mathit{Number} \\ &\mid \mathit{MulE} \,/\, \mathit{Number} \\ &\mid \mathit{Number} \end{aligned}$$

Parse trees for multiplication expressions would contain addition expression nonterminals, and this would confuse a cross-language clone detector when the other language's grammar does not encode precedence. We propose a simple technique that converts ANTLR parse trees that have hard-coded precedence into what we call *reduced parse trees*, resembling an abstract syntax tree. We then abstract upon all trees to arrive at a good common approximation. We refer to the process of creating reduced parse trees as *parse tree reduction*, and it is described in Section 5. After reduction, we are left with a new, smaller parse tree represented in Figure 10c. It is now both drastically smaller and more similar to the Java parse tree in terms of structure and number of nodes.

B.3 Score and Peace

We now have trees that superficially look similar, but they are still arranged differently and come from grammars with different nonterminals and levels of granularity. We address this issue by categorizing nonterminals into *equivalence classes* and comparing those classes. We call this process *parse tree abstraction*, and the resulting trees *abstracted parse trees*.

We must also find a way to compare the terminals. Before abstraction, we combine any consecutive terminal nodes into bags of words, correcting for camelcase, underscores, and capitalization. We can then compare the bags of words using characteristic vector similarity.

Finally, we linearize the abstracted trees by performing a *preorder traversal* of the two abstracted trees to minimize the remaining dissimilarities between the two languages.

For our example function pair, we place each node in the parse tree into its equivalence class, remove any "uninteresting" nonterminal nodes that do not belong to any equivalence class, and split the terminals on word boundaries. After linearization, this process yields the two final sequences to compare:

- C++: `FunDef, {"int"}, Id, {"binsearch"}, {"const"}, {"t"}, Id, {"array"}, {"int"}, Id, {"left"}, {"int"}, Id, {"right"}, {"t"}, Id, {"what"}, If, {"if"}, Relational, Id, {"right"}, Id, {"left"}, Return, {"return"}, Unary, Literal, Decl, {"int"}, Decl, Id, {"mid"}, Multiply, Add, ...`
- Java: `FunDef, {"int"}, Id, {"binary", "search"}, {"int"}, Id, {"nums"}, {"int"}, Id, {"check"}, {"int"}, Id, {"low"}, {"int"}, Id, {"high"}, If, {"if"}, Relational, Id, {"high"}, Id, {"low"}, Return, {"return"}, Unary, Id, Literal, Decl, {"int"}, Decl, Id, {"center"}, Multiply, Id, Add, ...`

We can now use a sequence alignment algorithm to compute the similarity of the two sequences. A pair of bags of words is compared by converting to a characteristic vector and calculating characteristic vector similarity, and these terminal sets never match against nonterminals. A pair of nonterminals is compared by checking the equivalence classes for equality. To incentivize aligning similar kinds of nonterminal statements with each other, we assign higher weights to some equivalence classes (such as those representing `if` statements). We also allow for a weight to be applied to the bags of words.

We feed the combined terminal/nonterminal sequences and weights to the Smith-Waterman local sequence alignment algorithm; this contrasts with Levenshtein distance, a global alignment algorithm that was used in Kraft et al.'s and Al-Omari et al.'s work. Local alignment algorithms are better suited for finding small pockets of similarity in sequences, whereas global alignment must consider the entire length of each sequence pair [8]. Thus, we believe that local alignment is a better choice for cross-language clone detection.

We normalize the result to get a similarity score between 0 and 1. These similarity scores do not have an absolute meaning but instead have a meaning relative to each other. In order to turn a score into a binary decision one can apply a threshold score such that only function pairs with scores over the threshold are considered possible clones.

For this particular example, when we run our scoring algorithm with the parameters we chose for C++ and Java in Section 6, we get a score of 0.607; this is greater than the cutoff value of 0.5 we chose in our evaluation section, so we come to the conclusion that these two snippets are clones.