SLACC: Simion-based Language Agnostic Code Clones

George Mathew, Chris Parnin, Kathryn T Stolee
North Carolina State University
{george2,cjparnin,ktstolee}@ncsu.edu

ABSTRACT
Successful cross-language clone detection could enable researchers and developers to create robust language migration tools, facilitate learning additional programming languages once one is mastered, and promote reuse of code snippets over a broader codebase. However, identifying cross-language clones presents special challenges to the clone detection problem. A lack of common underlying representation between arbitrary languages means detecting clones requires one of the following solutions: 1) a static analysis framework replicated across each targeted language with annotations matching language features across all languages, or 2) a dynamic analysis framework that detects clones based on runtime behavior.

In this work, we demonstrate the feasibility of the latter solution, a dynamic analysis approach called SLACC for cross-language clone detection. Like prior clone detection techniques, we use input/output behavior to match clones, though we overcome limitations of prior work by amplifying the number of inputs and covering more data types; and as a result, achieve better clusters than prior attempts. Since clusters are generated based on input/output behavior, SLACC supports cross-language clone detection. As an added challenge, we target a static typed language, Java, and a dynamic typed language, Python. Compared to HitoshiI, a recent clone detection tool for Java, SLACC retrieves 6 times as many clusters and has higher precision (86.7% vs. 30.7%).

This is the first work to perform clone detection for dynamic typed languages (precision = 87.3%) and the first to perform clone detection across languages that lack a common underlying representation (precision = 94.1%). It provides a first step towards the larger goal of scalable language migration tools.

CCS CONCEPTS
• Software and its engineering → Software maintenance tools; Object oriented languages; Functional languages; • Information systems → Clustering.

KEYWORDS
semantic code clone detection; cross-language analysis

ACM Reference Format:

1 INTRODUCTION
Modern programmers typically work on systems built with a cocktail of multiple programming languages [12]. A recent survey found that professional software developers have a mean of seven different programming languages in their industrial software projects [30] and open-source software projects frequently have between 2–5 programming languages [29, 43]. Programmers are also expected to continue learning multiple programming languages on a daily basis. To learn a new programming language, studies have shown that programmers attempt to use a cross-language learning strategy by reusing knowledge from a previously known language [37, 38, 49]. This means programmers often need the ability to relate code snippets across multiple programming languages.

Traditional clone detection often works with only a single programming language, meaning that typical applications and tools are not applicable to modern programming systems and contexts. These applications include bug detection in ported software [35], maintaining quality through refactoring [52], and protecting the security of products [45]. For example, security teams at Microsoft use clone detection to scan for other instances of vulnerable code that might be present in any production software [8]. In short, there is a need to extend clone detection to work in cross-language contexts, but limited support exists for them.

This paper presents Simion-based Language-Agnostic Code Clone detection technique (SLACC), a cross-language semantic clone detection technique based on code behavior. Our technique can match whole and partial methods or functions. It works in both static and dynamic languages. It does not require annotations or manual effort such as seeding test inputs. Critically, unlike any other clone detection technique, we are able to detect semantically similar code across multiple programming languages and type systems (e.g., Python and Java).

SLACC finds semantic clones by comparing the input/output (IO) relationship of snippets, called simions (short for similar input output functions), in line with prior work [20, 40]. SLACC segments a target code repository into smaller executable functions. Arguments for the functions are generated using a custom input generator inspired by grey-box testing and multi-modal distribution. Functions are executed on the generated arguments and subsequently clustered based on the generated arguments and corresponding return values. The similarity measure for clustering is based on the IO behavior of code snippets and is independent of their syntactic features. Hence, SLACC generates cross-language clusters with code snippets from different programming languages. To validate our technique, using a single, static typed language, we perform an empirical study with 19,188 Java functions derived from Google Code Jam (GJC) [15] submissions and demonstrate...
that SLACC identifies 6x more clones and with higher precision (86.7% vs. 30.7%) compared to HitoshiIO [40], a state-of-the-art code semantic clone detection technique. Using a single, dynamic typed language, we perform a study with 17,215 Python functions derived from GCJ and find that SLACC can identify true behavioral clones with 87.3% precision. For cross-language clones, SLACC finds 32 clusters with both Python and Java functions, demonstrating that detection of code clones does not depend on a common type system.

In summary, this paper makes the following contributions:

- For single-language static typed clone detection, an empirical validation demonstrating SLACC can be used to identify 6x more and better code clones clusters than the state-of-the-art code-clone detection technique HitoshiIO.
- The first exploration of clone detection for a dynamic-typed language and demonstrated feasibility in Python with precision of 87.3%.
- The first exploration of cross-language clone detection when the languages lack an underlying representation; SLACC is successful in identifying cross-language clone clusters between Python and Java with 94.1% precision.
- An open-source tool for detection of semantic code clones between different programming languages.

2 MOTIVATION

Avery is preparing for a technical interview and was given a few practice coding challenges [50] to work on. Avery is more comfortable writing code in Java during an interview setting but is worried because the company exclusively codes in Python. As practice for the interview, Avery wants to code with Python. First, Avery decides to write the code in Java to understand the solution, and then translate those solutions into Python code.

One of the practice questions asks the coder to interleave the results of two arrays. Avery quickly writes this solution in Java:

```java
public String interleave(int[] a, int[] b) {
    String result = "";
    int i = 0;
    for (i = 0; i < a.length && i < b.length; i++) {
        result += a[i];
        result += b[i];
    }
    int[] remaining = a.length < b.length ? b : a;
    for (int j = i; j < remaining.length; j++) {
        result += remaining[j];
    }
    return result;
}
```

While one approach is to directly translate the code into Python, Avery wonders if there are other ways to take advantage of idioms and capabilities in Python. After spending a few hours searching Stack Overflow [42] and GitHub Gists [41], Avery finds a few code snippets that seem to do the same thing.

The first one seems a bit too complex and relies on another dependency.

```python
import chain
```

This other solution is similar to the Java solution, but is using something new, a zip function. Avery is excited to learn some new Python tricks!

```python
def problem2(l1, l2):
    result = ""
    for (e1, e2) in zip(l1, l2):
        result += str(e1)
        result += str(e2)
    return result
```

Avery found the strategy of writing code in Java and translating that code into Python helpful. However, the process of manually searching and translating the code between languages was time-consuming. Avery’s unfamiliarity with Python made it difficult to verify whether these snippets were truly the same.

At the interview, Avery was relieved to be asked to solve the same interleave problem from the practice set! However, while coding up a solution in Python, the interviewer asked, does this handle interleaving uneven lists? The original Java-based solution handled this case, but the Python translation did not. Because searching for code took so long, Avery never had the chance to fully verify that the Python solution worked the same as the Java solution. Avery’s assumption that the new zip function would work on uneven lists was wrong! Had there been a better way for Avery to find semantically related snippets in other programming languages, this issue may have been avoided.

In this work, we introduce SLACC, which could detect that these functions are not equivalent. From a corpus of code, it could instead find this semantically identical snippet—just one of many applications enabled by cross-language clone detection.

```python
def valid_interleave1(l1, l2):
    result = ""
    a1, a2 = len(l1), len(l2)
    for i in range(max(a1, a2)):
        if i < a1:
            result += str(list1[i])
        if i < a2:
            result += str(list2[i])
    return result
```

3 SIMION-BASED LANGUAGE-AGNOSTIC CODE-CLONE DETECTION

Code clones can be broadly classified into four types [36] as described in Table 1. Types I, II and III represent syntactic code clones where similarity between code is estimated with respect to the structure of the code. On the other hand, type-IV indicates functional similarity. Syntactic code clone detection techniques are impractical for cross-language code clone detection as it would require an explicit mapping between the syntax of the languages. This is feasible for syntactically similar languages like Java and C# [11] but much harder for different languages like Java and Python.

On the other hand semantic approaches for cross-language code detection [33] rely on large number of training examples between the languages and was yet again tested on similar programming languages.
We propose Simion-based Language-Agnostic Code_Clone detection (SLACC), a semantic approach to code similarity that is predicated on the availability of large repositories of redundant code [2]. Instead of mapping API translations using predefined rules [5, 11], or using embedded API translations [4, 33], SLACC uses IO examples to cluster code based on its behavior. Further, it relaxes the bounds of the datatypes across programming languages, which helps dynamic typed code snippets (e.g., Python) to be clustered alongside static typed code snippets (e.g., Java).

In SLACC, we build on the ideas pioneered by EQMiner [20] for using segmentation and random testing for clone detection. SLACC starts by identifying snippets from a large code base and involves a multi-step process depicted in Figure 1, which starts with a) Segmentation of the code base into smaller fragments of code called snippets, b) Function creation from the snippets, c) Input generation for the functions, d) Execution of the functions, and e) Clone detection based on clustering functions arguments and execution results.

### 3.1 Segmentation

In the first stage, code from all the source files in a project is broken into smaller code fragments called snippets. Consecutive statement blocks of threshold MIN_STMT or more are grouped into a snippet. A statement block can be

1. Declaration Statement. e.g., `int x;`
2. Assignment Statement e.g., `x = 5;`
3. Block Statement e.g., `static (x = 10;)`
4. Loop statements. e.g., `for, while, do-while`
5. Conditional statements. e.g., `if, if-else-if, switch`
6. Try Statement. e.g., `try, try-catch`

Algorithm 1 illustrates the segmentation phase. For an AST $A_F$ of a function, the algorithm performs a pre-order traversal of all the nodes in the AST (line 5) and then uses a sliding window to extract segments of size greater than a minimum segment size MIN_STMT (lines 12-13). Further, for statements like Block, Loop, Conditional and Try which have statements in its nested scope, the algorithm is called recursively on them (lines 14-15).

### 3.2 Function Creation

Next, snippets are converted into executable functions. This section describes how arguments, return variables, and types are inferred.

**Inferring arguments and return variables.** We adapt a dataflow analysis similar to that used by Su et al. [40]. For each method, potential return variables are identified as variables that are defined or modified within the scope of the snippet. If the last definition of a variable is a constant value, that variable is removed from the set of potential return variables. Arguments are variables that are 1) used but not defined within the scope of the snippet, and 2) not declared as public static variables for the class. For each potential return variable in a snippet, a function is created.

**Inferring types.** In the case of static typed languages, argument types and return values can be inferred via static code analysis. For dynamic typed languages, the parameters can take multiple types of input arguments. This increases the possible values of the arguments generated (see Section 3.3) to identify its behavior. In many cases, the possible types for the arguments can be inferred by parsing the code and looking for constant variables [7] in its context. This technique has been used in inferring types in other dynamic languages like JavaScript [18]. For example, in the following Python function, the type of n can be assumed to be an integer since it is compared against an integer.

```python
def fib(n):
    if n <= 1: return n
    return fib(n-1) + fib(n-2)
```

In cases where the types of the parameters could not be inferred at compile time, such as:

```python
def main(a):
    print a
```
class Shape {
    public int length;
    int width;
    private int height;
    public Shape(int l, int w, int h) {
        length=l; width=w; height=h;
    }
    public int func_s (int l, int w, x).w, int divisor, int if (b == int l, Shape {
        w, public int x) {
            w, int 0 public x) {
                width; divide_simple (int l, w, a, int a / b;
            return new Shape(l + x, w * /f_irst function divides
            return quotient; 
            public int func_w (int l, int w, int x) {
                return func_s(l, w, x).width;
            }
        }
    }
}

Figure 2: An example depicting conversion of a function with object as a return type to multiple functions with non-primitive members of the object’s class.

Figure 3: An example illustrating the need for reordering arguments. The two functions perform integer division but do not return the same return value for the same set of inputs due to the order of arguments in the function definition.

Converting object return types into functions. If a snippet returns an object, the object is simplified into multiple functions returning each of its non-private members independently. For example, in Figure 2, func_s has a return type of Shape. Shape has two members, length and width. Hence, func_s is broken down into two functions, func_l and func_w, which return the length and width of the shape object independently. Note that a third function for height is not created since it is a private member.

Permuting argument order. For each of the snippets, we generate different permutations based on the input of arguments since order matters for capturing function behavior. Consider the two functions in Figure 3; the first function divides a with b using the division (/) operator while the second divides dividend with divisor using the subtract (-) operator recursively. For the inputs (5, 2) the two functions would produce the values 2 and 0 respectively. But if the arguments for the second function was reversed, it would produce the same output 2. Thus, for every function, we create duplicates in different permutations of the arguments, ARGS, resulting in |ARGS| different functions. To limit the creation of this exploding space, we set an upper limit on the number of arguments per function that is included in the analysis (ARGS_MAX).

3.3 Input Generation

A set of inputs are required to execute the created functions. Following this, clustering is performed.

Input creation. Inputs are generated based on argument type and using a custom input generator inspired by grey-box testing [23] and multi-modal distribution [20]. First, the source code is parsed and constants of each type are identified. Next, a multi-modal distribution is declared for each of the types with peaks at the constants. Finally, values for each type are sampled from this multi-modal distribution. Our experiments create 256 inputs per function, as justified in Section 6.1.

Memorization. For every function with the same argument types, a common set of inputs have to be used to compare them. This is ensured using a database and the input generator. The generator is used to create sample inputs for the given argument types and stored in the database. For subsequent functions with the same signature for the arguments, the stored input values are reused.

Supported argument types. SLACC currently supports four types of arguments.

(1) Primitive. The multi-modal distribution for the argument type is sampled to generate the inputs. This includes integers (and longs, shorts), floats (and double), characters, booleans, and strings.

(2) Objects. Objects are recursively expanded to their constructor with primitive types; inputs are generated for the types.

(3) Arrays. A random array size is generated using the input generator for integers. For each element in the array, a value is generated based on the array type (Primitive or Object).

(4) Files: Files are stored as a shared resource pool of strings in the database. If a seed file(s) is provided, it is randomly mutated and stored as a string in the database. In the absence of a seed, constants from the multi-modal distribution are sampled and stored as strings. For an argument with a file type (or its extensions), a temporary (deleted on termination) file object is created using the stored strings.

Type size restrictions. Comparing code snippets requires compatible sizes of types across programming languages. For example, Java has 4 integer datatypes byte, short, int and long which occupy sizes of 1, 2, 4 and 8 bytes, respectively. On the other hand, 1

1If a negative integer is sampled, the distribution is re-sampled.
Python has two integer datatypes: int which is equivalent to the long datatype in Java and long which has an unlimited length. Thus, we make a restriction when generating inputs for functions across different languages: inputs are generated from the smaller bound of the two programming languages. For example, in the case of Java and Python function that has an int, inputs are generated within the bounds of Java.

### 3.4 Execution

In the next stage, the created functions are executed over the generated input sets and the subsequent return values are stored. Each function is assigned an execution time limit of $T_L$ seconds, after which a Timeout Exception is raised. This occurs most frequently within the bounds of Java.

Consider the functions from Section 2, `interleave`, `fancy_interleave`, and `valid_interleave`. For values $a = [2, 3]$ and $b = [4]$, we see that $\text{interleave}(a, b) = [2, 4, 3]$, $\text{fancy_interleave}(a, b) = [2, 4, 3]$ and $\text{valid_interleave}(a, b) = [2, 4, 3]$. Functions `interleave` and `valid_interleave` are similar since they have the same output for the same input but `interleave` and `fancy_interleave` are not similar. In contrast, for $a = [2, 3]$ and $b = [4, 5]$, all three functions would return the same output $[2, 4, 3, 5]$. Based on these two inputs, `interleave` and `fancy_interleave` have a similarity of 0.5, `interleave` and `valid_interleave` have a similarity of 1.0, and `fancy_interleave` and `valid_interleave` have a similarity of 0.5. This process is repeated for many such inputs $a$ and $b$ to compute similarity scores between each pair of functions.

Functions are only compared if they have the same number of arguments and cast-able argument types. For example, consider the four functions $f_1(\text{int } a, \text{String } b)$, $f_2(\text{long } a, \text{File } b)$, $f_3(\text{File } a, \text{String } b)$ and $f_4(\text{String } a)$. Functions $f_1$ and $f_2$ can be compared since `int` can be cast to a `long` value. But they cannot be compared to $f_3$ since primitive types cannot be cast to `File`. Similarly, $f_1$, $f_2$ and $f_3$ cannot be compared to $f_4$ due to the difference in number of arguments.

### 3.5 Clone Detection

The last stage of SLACC is identifying the clones, where the executed functions are clustered on their inputs and outputs. SLACC uses a representative based partitioning strategy [36, 40] to cluster the executed functions.

**Similarity Measure.** In this work, a pair of functions have the highest semantically similarity if for any given input, the functions return the same output. The similarity measure between two functions is computed as the number of inputs for which the methods return the same output value divided by the number of inputs, same as the Jaccard index. This creates a similarity value between two functions with a range of $[0, 1]$ with 1.0 being the highest.

Consider the functions from Section 2, `interleave`, `fancy_interleave`, and `valid_interleave`. For values $a = [2, 3]$ and $b = [4]$, we see that $\text{interleave}(a, b) = [2, 4, 3]$, $\text{fancy_interleave}(a, b) = [2, 4, 3]$ and $\text{valid_interleave}(a, b) = [2, 4, 3]$. Functions `interleave` and `valid_interleave` are similar since they have the same output for the same input but `interleave` and `fancy_interleave` are not similar. In contrast, for $a = [2, 3]$ and $b = [4, 5]$, all three functions would return the same output $[2, 4, 3, 5]$. Based on these two inputs, `interleave` and `fancy_interleave` have a similarity of 0.5, `interleave` and `valid_interleave` have a similarity of 1.0, and `fancy_interleave` and `valid_interleave` have a similarity of 0.5. This process is repeated for many such inputs $a$ and $b$ to compute similarity scores between each pair of functions.

**Clustering.** A function is compared to a cluster by measuring its similarity with the first function added to the cluster (called representative). The clustering algorithm is briefly described in Algorithm 2. An empty set of clusters is first initialized (line 4). Each function (line 5) is compared against each cluster (line 6). If the similarity between the representative (line 7) and the function is greater than a predefined similarity threshold, $SIM_T$ (line 8), the function is added to the cluster (line 9). If the function does not belong in any cluster (line 11), a singleton cluster is created for the function (line 12) and the function is set as the cluster’s representative (line 13). The singleton cluster is added to the set of clusters (line 14).

### 4 EVALUATION

Our goal is to evaluate the effectiveness of SLACC. There is a three-phase evaluation, first to compare SLACC to a comparable technique in a single, static typed language. Next, we apply SLACC to a single, dynamic typed language (Python) and then to a multi-language context; in both cases SLACC is compared to type-III clones.

#### 4.1 Research Questions

SLACC is benchmarked against HitoshiLO [40] with respect to coverage and precision of code-clone detection. This leads us to our first research question:

**Research Question 1**

How effective is SLACC on semantic clone detection in static typed languages?
We validate this study on four problems from Google Code Jam APIs between similar languages (e.g., Java and C#) using prede-
We use the JavaParser [44] tool and Python AST [34] module to detect code clones in dynamic languages. Hence we benchmarked SLACC for dynamic and cross-language clones by matching the Abstract Syntax Trees (ASTs) as a proxy for similarity. This technique has been adopted by many graph-based (an example of type-III clone) code clone detection techniques in C [3, 19, 51] and Java [19, 25].

Like SLACC, the first phase of the AST comparison segments the code into snippets. Next an AST is generated for the snippets. We use the JavaParser [44] tool and Python AST [34] module to construct the ASTs in the respective languages. We measure similarity by matching the ASTs. For clones in the same programming language (RQ1, RQ2), we match the ASTs and consider them to be type-III clones if the ASTs are equivalent or have a difference of at most one node.

4.3.2 RQ2: Automated AST Comparison. To the best of our knowledge we could not find a prior work to detect semantic code clones in dynamic languages. Hence we benchmarked SLACC for dynamic and cross-language clones by matching the Abstract Syntax Trees (ASTs) as a proxy for similarity. This technique has been adopted by many graph-based (an example of type-III clone) code clone detection techniques in C [3, 19, 51] and Java [19, 25].

Like SLACC, the first phase of the AST comparison segments the code into snippets. Next an AST is generated for the snippets. We use the JavaParser [44] tool and Python AST [34] module to construct the ASTs in the respective languages. We measure similarity by matching the ASTs. For clones in the same programming language (RQ1, RQ2), we match the ASTs and consider them to be type-III clones if the ASTs are equivalent or have a difference of at most one node.

4.5.3 RQ3: Manual Cross-language AST Comparison. The automated AST comparison approach cannot be adopted for cross-language clones (RQ3) due to the difference in format of the ASTs for both the languages. In this case, conservatively, we sampled cross-language snippets with extremely similar outputs and manually verified the ASTs for similarity. To do this, we randomly sample 1 million pairs of a Java function and a Python function. If the input and output types are compatible, and the outputs are the same for the same inputs or off by a consistent value, then we manually evaluate the ASTs for similarity. Consistency is determined based on the output type. Values of primitive types are consistent if they have a constant difference (for Boolean or Numeric values), constant ratio (for Boolean or Numeric values) or constant Levenshtein distance [48] (for Strings) between the outputs. Objects are consistent if each member of the object is consistent. Finally, two arrays are consistent, if all the corresponding members of the array are consistent.

For example, given two methods, int A(int x) and def B(y), if A(1) = 1, B(1) = 9, A(2) = 2, and B(2) = 18, then A() and B() are similar since their outputs have a constant ratio (9). Of the 616 similar pairs, all had identical ASTs or had a difference of at most one node, making them type-III clones.

4.6 Precision Analysis

SLACC and HitoshiIO are both clustered using IO relationships of the functions. However, given a different set of inputs, some functions in a cluster might produce a different set of outputs such that they are not clones; such clusters are marked as false positives and considered invalid. We identify false positives at the cluster-level in keeping with prior work [20].

To detect false positives, SLACC clusters are re-executed on a new set of 256 inputs generated using random fuzzing [20] based on a triangular distribution, and clustered. If any method in a cluster is not grouped into the same cluster using the new input set, the whole cluster is marked as a false positive. We observe that the number of clusters and false positives is relatively stable above 64 inputs (Section 6.1).

To detect false positives in HitoshiIO, we randomly fuzz the test input files 32 times (Section 4.5) to generate a new test file that is 32x the size of the original, and then re-execute HitoshiIO. Clone pairs are clustered and false-positives are detected when a new cluster does not match an original cluster, as done for SLACC.

False positives in clusters generated by AST comparisons are identified in a similar manner to SLACC. ASTs in the clusters are first converted to functions (as described in Section 3.2). The functions are re-executed on 256 inputs like SLACC clusters and checked for false positives. Any cluster that contains a different method after execution is marked as a false positive.

5 RESULTS

The results show that SLACC identifies more method level clones compared to prior work and with higher precision (RQ1), successfully identifies clones in dynamic typed languages (RQ2), and successfully detects clones between Java and Python (RQ3).

5.1 RQ1: Static Typed Languages

The 885 Java methods generated 19,188 Java functions for analysis. SLACC was able to support 691 of the 885 Java methods. From the 691 whole methods, 18,497 functions are derived into partial method snippets. Of the total generated functions, 4,180 (22%) are clones resulting in 632 clusters. These 4,180 clones derive from 4,038 partial-method snippets and 142 whole methods. We call them statement level clones and method level clones, respectively.

5.1.1 Method level clones. We benchmark SLACC against HitoshiIO by comparing clones detected by SLACC at a method level granularity. We provide all 885 Java methods to HitoshiIO, which groups 43 of the methods into 13 clusters. False positives were identified for 9 of the 13 clusters (precision=30.7%).4 The remaining valid clusters from HitoshiIO contain 20 methods. From the 691 Java methods, SLACC detected 142 methods, grouped into 15 clusters. False positives were identified for 2 of the 15 clusters (precision = 86.7%). The remaining valid clusters for SLACC contain 135 methods.

Table 3 shows the numbers of valid clusters for each approach, as well as their intersection. All valid clusters from HitoshiIO are

<table>
<thead>
<tr>
<th>Problem</th>
<th>HitoshiIO(H)</th>
<th>SLACC(S)</th>
<th>H∩S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irregular Cake</td>
<td>3</td>
<td>44</td>
<td>3</td>
</tr>
<tr>
<td>Perfect Game</td>
<td>4</td>
<td>35</td>
<td>4</td>
</tr>
<tr>
<td>Cheaters</td>
<td>4</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>Magical Tour</td>
<td>9</td>
<td>35</td>
<td>9</td>
</tr>
<tr>
<td>Total</td>
<td>20</td>
<td>135</td>
<td>20</td>
</tr>
</tbody>
</table>

4False positive rates in the original HitoshiIO paper [40] are computed at the pair-level rather than cluster level and used student opinions rather than code behavior, which may account for the relatively low precision reported here.
import Y14R5P1.stolis.MMT3 // Parent Class MMT3
public static int func_a(BufferedReader br)
    // Snipped from Y14R5P1.stolis.MMT3.main()
    if (!MMT3.in.hasMoreTokens())
        MMT3.in = new StringTokenizer(br.readLine());
    int a = Integer.parseInt(MMT3.in.nextToken());
    return a;
}

import Y12R5P1.xiaowuc.A // Parent Class A
public static int func_b(Scanner in) {
    return next;
    while (!Y14R5P1.stolis.MMT3.in.hasMoreTokens()) {
        a = Integer.parseInt(Y12R5P1.xiaowuc.A.in.nextToken());
        return a;
    }
}

public static int func_d(StreamTokenizer in) {
    return next;
    if (!MMT3.in.hasMoreTokens()) {
        MMT3.in = new StringTokenizer(in.readLine());
        int a = Integer.parseInt(A.in.nextToken());
        return a;
    }
}

public static int func_e(Scanner in) {
    return next;
    while (in.nextToken();
        return Integer.parseInt(A.in.nextToken());
    }
}

import Y14R5P1.eatMore.A // Parent Class A
public static int func_e(Scanner in) {
    // Y14R5P1.eatMore.A.nextInt()
    A.in = in;
    return Integer.parseInt(A.in.nextToken());
}

import Y14R5P1.dooglius.A // Parent Class A
public static int func_f(Scanner sc) {
    // Snipped from Y11R5P1.dooglius.A.go()
    int next = sc.nextToken();
    return next;
}

Figure 4: Semantic clusters detected by HitoshiIO, SLACC on method level (SLACC_method) and SLACC on statement level (SLACC_stmt). The cluster contains functions that take an object that reads a file and returns the next Integer token. Functions func_c and func_d are clones detected by HitoshiIO. Within the same cluster, SLACC_method additionally identifies two more method level clones that were not detected by HitoshiIO: func_b and func_e.

5.2 RQ2: Dynamic Typed Languages
SLACC identified that 3,135 (18.2%) of the 17,215 extracted Python functions had clones which resulted in 482 clone clusters. Of these 482 clusters, 421 are valid, resulting in precision of 87.3%. As a baseline, using the same Python functions, we systematically looked for type-III clones. There exists 3,971 clusters, of which 181 are valid (4.6% precision); these results are shown in the Python column of Table 4, where AST shows the type-III clones. For sake of comparison, the experiment was repeated for Java clones; a similar differential between SLACC and AST precision was observed (92.4% vs. 3.7%).

When these clusters are validated, 61 of the 482 SLACC clusters (12.8%) were deemed to be false positive. This is more than the percentage of false positives in Java (7.3%), but we suspect that by executing the functions over a larger set generated arguments, the subsequent clustering could yield more robust results.

An example of Python clones identified by SLACC can be seen in Figure 5. Both the functions in this example compute the sum of an array. func_db8e uses a loop that maintains the running sum where each index in the array contains the array sum until that index. The last index of the array would contain the array sum and

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Python</th>
<th>Java + Python</th>
</tr>
</thead>
<tbody>
<tr>
<td># Clusters</td>
<td>632</td>
<td>6122</td>
<td>482 3971 34 616</td>
</tr>
<tr>
<td># Valid</td>
<td>584</td>
<td>226</td>
<td>421 181 32 25</td>
</tr>
<tr>
<td>Precision</td>
<td>92.4</td>
<td>3.7</td>
<td>87.3 4.6 94.1 4.1</td>
</tr>
</tbody>
</table>

Table 4: # of Java, Python and Cross language clusters detected by SLACC compared against AST (Type-III) clusters.

RQ1: Method level clones: SLACC identifies more method level clones compared to HitoshiIO at higher precision. Statement level clones: Segmentation of code increases the precision of SLACC and yields a higher number of semantic clones.
1. **def** func_db8e(a):
2.     n = len(a)
3.     sum0 = [0] * (n + 1)
4.     for i in xrange(n):
5.         sum0[i + 1] = sum0[i] + a[i]
6.     allv = sum0[-1]
7.     return allv

1. **def** func_43df(items):
2.     _sum = sum(items)
3.     j = len(items) - 1
4.     return _sum

Figure 5: Semantic cluster of Python functions detected by SLACC. The cluster contains functions that returns the sum of an input array.

1. **static** long func_3b0e (Long[] x2) {
2.     Long res = null;
3.     Long[] arr = x2;
4.     int len = arr.length;
5.     for (int i = 0; i < len; ++i) {
6.         long xx = arr[i];
7.         if (xx >= res)
8.             continue;
9.         res = xx;
10.     }
11.     return res;
12. }

1. **def** func_6437 (y):
2.     ymin = min (y)
3.     count = 0
4.     return ymin

Figure 6: Semantic cluster of a Java function and a Python function detected by SLACC. The cluster contains functions that returns the minimum value in an input integer array.

is eventually returned. In contrast, func_43df uses the sum library function to perform the same task.

RQ2: SLACC can successfully identify code clones for dynamic typed languages with high precision (87.3%).

5.3 RQ3: Across Programming Languages

We execute SLACC on the Java and Python projects from GCJ. From 36,403 extracted snippets, SLACC identified 131 Java and 48 Python functions clustered into 34 cross-language clusters (single-language clusters are omitted from the RQ3 analysis). On validation, we find that 2 of these 34 (5.8%) clusters are false positives which is better than the percentage of false positives found in Java and Python independently. That said, SLACC would produce more clusters when support for the languages is broadened.

We discover 616 type-III clusters by comparing the ASTs of Java and Python snippets (Table 4), of which 25 clusters are valid (4.1% precision). It should be noted that this is a conservative precision estimate; the baseline was created by starting with close behavioral matches, hence giving the AST analysis a slight edge on precision (Section 4.5.2).

An example of a pair of Java-Python clones can be seen in Figure 6. func_3b0e is a Java function that uses a loop to find the minimum in an array while func_6437 is a Python function uses the inbuilt min function in Python.

RQ3: SLACC succeeds in identifying clones between programming languages irrespective of their typing.

6 DISCUSSION

We have demonstrated how SLACC can successfully identify clones in single-language, multi-language, static typed language, and dynamic typed language environments. Compared to prior art (HitoshiIO), SLACC identifies a superset of the clusters and with higher precision. Compared to type-III clone detection, SLACC achieves a much higher precision in Python and in cross-language situations. This would lead us to believe that traditional methods that detect syntactic type-III clones cannot be used for cross-language clone detection, despite successful applications in single languages for identifying libraries with reusable code [6], detecting malicious code [45], catching plagiarism [1] and identifying opportunities for refactoring [31].

Next, we explore the sensitivity of code clones to the number of inputs, the number of arguments, and the size of the snippets.

6.1 Impact of input sizes

Prior studies have shown that varying the number of inputs can alter the accuracy of clone detection techniques [20, 24, 46]. This was particularly evident in the earliest clone detection techniques by Jiang and Su [20] where the authors limited the number of inputs to 10 with a maximum of 120 permutations of the input due to the need for large computational resources and the corresponding runtime.

We test the impact on clones, clusters, and false positives by varying the number of inputs from 8 to 256 in powers of 2 and repeating SLACC using the generated Java functions. Each experiment is repeated 20 times on a set of randomly generated inputs. For each set of input, we record the mean and variance for the number of clones, clusters and false positives, as shown in Table 5. For

<table>
<thead>
<tr>
<th># Inputs</th>
<th># Clones</th>
<th># Clusters</th>
<th># False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4461(85)</td>
<td>218(16)</td>
<td>184(19)</td>
</tr>
<tr>
<td>16</td>
<td>4297(49)</td>
<td>355(17)</td>
<td>142(19)</td>
</tr>
<tr>
<td>32</td>
<td>4221(23)</td>
<td>412(13)</td>
<td>101(5)</td>
</tr>
<tr>
<td>64</td>
<td>4194(4)</td>
<td>623(6)</td>
<td>71(3)</td>
</tr>
<tr>
<td>128</td>
<td>4180(0)</td>
<td>630(1)</td>
<td>52(0)</td>
</tr>
<tr>
<td>256</td>
<td>4180(0)</td>
<td>632(0)</td>
<td>50(0)</td>
</tr>
</tbody>
</table>

Table 5: Mean and variance (in parenthesis) of # clones, # clusters and # false positives for 20 repeats when # inputs varying between 8-256. The mean (and variance) are reported.
We use our engineering judgment to set ARGS_MAX = 5 (Maximum number of Arguments) to limit the number of functions generated from snippets. Figure 7 represents the cumulative number of clones with arguments varying from 1 to 5 and can be used to justify our choice of ARGS_MAX. Most clones detected by SLACC have two arguments or less. In Java functions, 3252 of 4180 clones detected have less than three arguments. Cross-language functions are fewer in number and typically contain functions with 2 arguments or less (125 out of 131). This would seem intuitive as modules functions are more frequent compared to complex functionalities. As ARGS_MAX increases, it begins to plateau around 3. Hence, a larger value of ARGS_MAX may not yield significantly larger number of code clones but would incur more computational resources (ARGS_MAX! function executions).

6.3 Clones vs Lines Of Code
Prior work suggests there is more code redundancy at smaller levels of granularity [40]. Aggregating all the cloned functions identified by SLACC in RQ1, RQ2, and RQ3, we have 6,536 total, valid cloned functions in Java and Python (duplicates removed, as the same function could be included in an RQ1 and an RQ3 cluster, for example).

Figure 8 represents the number of clones with lines of code varying from 1 to 29. Clones with 30 or more lines are denoted as “30+”. More than 50% of the valid Java clones have 6 lines of code or less (2037/3845), while the median of valid Python clones have 5 lines or less (1372/2691). This implies that snippets with more lines of code are more unique and harder to clone functionally. On the contrary, smaller snippets are more likely to contain clones in a code base. The greater median for Java clones compared to Python clones can be attributed to the verbosity in Java compared to the succinct nature of Python [17].

7 RELATED WORK
In keeping with the survey on code clones by Roy et al. [36], research on code clones can broadly be classified as syntactic [3, 14, 19, 21, 26, 27], which represent structural similarities, and semantic [10, 20, 39, 40], which represent behavioral similarities.

EQMiner [20] is the closest related work with respect to our methodology. They examined the Linux Kernel v2.6.24 by using a similar segmentation procedure, used 10 randomly generated inputs to execute them, and cluster based on IO behavior. Compared with SLACC, EQMiner crucially ignores cross-language clone detection. Furthermore, the implementation of EQMiner contains several limitations, noted by Deissenboeck [9], that make cross-language detection infeasible and even replication itself impractical. As a result, we build on the ideas pioneered by EQMiner, while overcoming limitations in its original design. We introduce novel contributions, such as using grey-box analysis to overcome the limitations of simple random random testing, scale the input generation phase from 10 to 256 inputs, which drastically reduces false positives, introduce several steps and components to support complex language features, such as lambda functions, and handle differences arising from cross-language types. Finally, SLACC introduces flexibility in clustering as it permits a tolerance on similarity due to the SIM_T hyper-parameter.

HitoshiIO [40] by Su et al. also performs simion-based comparisons to identify clones. It uses existing workloads like test-cases or ‘main’ function calls to collect values for the behavior rather than the random testing approach proposed in EQMiner or the grey-box analysis approach used in SLACC. Research shows that existing unit tests do not attain complete code coverage [16] and as a result, the application of such a technique to open source repositories might not be produce a comprehensive set of clones. This conjecture can be observed in RQ1 where SLACC identifies more clones to HitoshiIO by an order of magnitude. Further, HitoshiIO operates at a method level granularity while SLACC can operate at method or statement level granularity. Naturally, this ensures a greater number of code clones since SLACC can identify succinct behavior in complex code snippets.
LASSO [22] by Kessel and Atkinson, like HitoshiIO, is another clone detection technique for method level clones from large repositories using test cases. But unlike HitoshiIO, it does not use pre-defined test cases; LASSO generates test cases using random generation via Evosuite [13]. That said, LASSO has many deviations compared to HitoshiIO and SLACC. Firstly, LASSO identifies only clones that have the same signature and method name (excluding case). Secondly, it detects clones only in methods where the arguments are primitive datatypes, boxed wrappers of primitives, strings, and one dimensional arrays of these datatypes. It fails to support objects; SLACC supports objects that can be initialized recursively using constructors of its members(Section 3.3). Finally, LASSO supports only strongly typed languages as it does not have a type inference engine like SLACC does.

Most clone detection techniques [3, 14, 19, 21, 26, 27] have been proposed for single language clone detection. With respect to cross language clone detection, we failed to find any techniques based on semantic behavior of code. A small number of techniques have been proposed on syntactic code features [32, 33]. API2Vec [33] detects clones between two syntactically similar languages by embedding source code into a vector representation and subsequently comparing the similarity between vectors to identify code clones. CLCDSA [32], identifies nine features from the source code AST and uses a deep neural network based model to learn the features and detect cross language clones.

Segmentation used in SLACC is inspired by methods that parse ASTs of the source code [3, 19]. These methods encode the ASTs into intermediate representations and do not account for the semantic relationships. For example, DECKARD [19] characterizes sub-trees of the AST into numerical vectors and clusters them based on the Euclidean distance which fails to capture the behavior of code in the clusters [20]. This limitation has been observed in other syntactic methods as well and is a reason for adoption of semantic techniques to detect code clones [20].

8 LIMITATIONS AND THREATS

Threats to external validity include the focus on two languages as instances of static and dynamic typing, so results may not generalize beyond Java and Python. The use of GCJ code may not generalize to more complex code bases. Threats to internal validity include that for RQ3, where we “help” the AST matching by starting with behavioral clusters and then determining if the ASTs are similar, which overestimates the precision of cross-language AST matching.

Our implementation of SLACC has the following limitations: Dynamic Typing. SLACC does not support two primitive types long and complex for Python. That being said, we verified that the GCJ projects used in this study, do not explicitly use these values in the source code and they are not present in the input file used by the baseline HitoshiIO. Further, in case of a failure to identify the type of a function argument, the function was fuzzed with arguments of all supported types. In this study, we supported primitive types and the simple data-structures tuple, set, list and dict. Support for other sophisticated data-structures can be incorporated by extending the existing SLACC API with instructions in the wiki [28].

Unsupported Features. Although SLACC supports Object-Oriented features such as inheritance and encapsulation, it is limited to objects derived from primitive types. Hence, the current version of SLACC cannot scale to more sophisticated objects like Threads and Database Connections. Similarly, for Python we do not support modules like generators and decorators. Nevertheless, it would be possible to support these features with more engineering effort.

Dead Code Elimination. In the code-clone examples of Figure 5 and Figure 6, we see the presence of lines of code that do not influence the return value i.e., Dead Code. At the moment, the functions do not fail due to dead code but eliminating them would make the functions more succinct and comprehensible. This will be an avenue for future work for specific applications of SLACC.

9 CONCLUSION

In this paper, we present SLACC, a technique for language-agnostic code clone detection that precisely yields semantic code clones across programming languages. This is the first research to identify semantic code clones in a dynamic typed language and also across differently-typed programming languages. SLACC identifies clones by comparing the IO relationship of segmented snippets of code from a target repository. Input values for the segmented code are generated using multi-modal grey-box fuzzing. This results in fewer false positives compared to current state of the art semantic code clone detection tool, HitoshiIO. In our study, we identify code clones between Java and Python from Google Code Jam submissions. Compared to HitoshiIO, SLACC identifies significantly (6x) more code clones, with greater precision (86.7% vs. 30.7%). SLACC also detects code clones in a multi-language code corpora. The number of clones detected was fewer and the number of false positives was slightly more compared to code clones within the same language. However, future work that broadens language support is likely to improve these metrics. These results have implications for future applications of behavioral code clones, such as enabling robust language migration tools or mastery of a new programming language once one is known.

SLACC is open-source and the data used in this study is publicly available [28].

ACKNOWLEDGMENTS

Special thanks to Fang-Hsiang Su, Jonathan Bell, Gail Kaiser and Simha Sethumadhavan for making HitoshiIO publicly available. We would also like to thank the anonymous reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation under Grant No. 1645136 and Grant No. 1749936.

REFERENCES

