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 HitoshiIO, 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:
George Mathew, Chris Parnin, Kathryn T Stolee. 2020. SLACC: Simion-based Language Agnostic Code Clones. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380407

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380407

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 **S**imion-based **L**anguage-**A**gnostic **C**ode **C**lone 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 **sim**ilar **i**nput **o**utput functio**ns**), 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 (GCJ) [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
def fancy_interleave(l1, l2):
    from itertools import chain
    return "".join([str(x)
            for x in chain.from_iterable(zip(l1, l2))])
```

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.

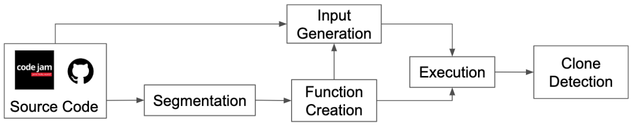

Figure 1: High level workflow for SLACC.

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

Table 1: Types of code clones. Types I, II and III are syntactic while type IV are semantic or behavioral clones [36]

Type	Description
I	Identical sans whitespace and comments
II	Identical AST but uses different variable names, types or function calls
III	Similar AST but uses different expressions/statements. For example, a) using **while** in place of **for** loops or b) using **if else if** in place of **switch** statements.
IV	Different syntax but behaviorally same. For example, an iterative stack approach or a recursive approach can be used for breadth first search of a graph.

Algorithm 1 Segmentation

1: **Input:** A_F - AST Node
2: **Output:** $\mathbb{S}$ - List of Segment
3: **procedure** SEGMENT(A_F)
4: $\mathbb{S} \leftarrow \phi$
5: stmts $\leftarrow$ $PreorderTraverse(A_F)$
6: **for all** i $\in$ range(0, **len**(stmts) − 1) **do**
7: $S_i \leftarrow \{\}$
8: $stmt_i \leftarrow$ stmts[i]
9: **for all** j $\in$ range(i, **len**(stmts)) **do**
10: $stmt_j \leftarrow$ stmts[j]
11: S_i.append($stmt_j$)
12: **if len**(S_i) $\geq$ MIN_STMTS **then**
13: $\mathbb{S} \leftarrow \mathbb{S} \cup S_i$
14: **if** $stmt_j$.**hasChildren**() **then**
15: $\mathbb{S} \leftarrow \mathbb{S} \cup$ **SEGMENT**($stmt_j$)
16: **return** $\mathbb{S}$

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.

```
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:

```
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 Shape func_s (int l, int w, int x) {
    return new Shape(l + x, w * 2, x);
}
public int func_l (int l, int w, int x) {
    return func_s(l, w, x).length;
}
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.

a generic type is assigned (i.e., for a) allowing the argument to assume any of the primitive types used in argument generation (Section 3.3).

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

```
public int divide_simple (int a, int b) {
    if (b == 0) return 0
    return a / b;
}
public int divide_complex (int divisor, int dividend) {
    // Same as dividend/divisor
    if (b == 0) return 0
    int quotient = 0;
    while (dividend >= divisor) {
        dividend = dividend - divisor;
        quotient++;
    }
    return quotient;
}
```

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.

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.

Memoization. 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[1]. 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]If 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 when there is an infinite loop, such as `while(true)` when the loop invariant is an argument. Each execution of the function is run on an independent thread. Subsequently, the return value, runtime and exception for the executed function over the input set is stored.

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 $[0.0, 1.0]$ 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 `interleave(a,b) = [2,4,3]`, `fancy_interleave(a,b) = [2,4]` and `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 `f1(int a, String b)`, `f2(long a, File b)`, `f3(File a, String b)` and `f4(String a)`. Functions `f1` and `f2` can be compared since **`int`** can be cast to a **`long`** value. But they cannot be compared to `f3` since primitive types cannot be cast to `File`. Similarly, `f1`, `f2` and `f3` cannot be compared `f4` due to the difference in number of arguments.

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*)

Algorithm 2 Clustering

```
1:  Input: 𝔽 - List of Functions with Input and Output
2:  Output: ℂ - List of clusters
3:  procedure CLUSTER(𝔽)
4:      ℂ ← ϕ
5:      for all F ∈ 𝔽 do
6:          for all C ∈ ℂ do
7:              O ← GetRepresentive(C)
8:              if Similarity(O, F) ≥ SIM_T then
9:                  C ← C ∪ F
10:                 break
11:         if ∀C ∈ ℂ, F ∉ C then
12:             C_{|C|+1} ← F
13:             SetRepresentative(C_{|C|+1}, F)
14:             ℂ ← ℂ ∪ C_{|C|+1}
15:     return ℂ
```

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 HitoshiIO [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?

Prior research has already shown that semantic clones can be found in static typed languages [10, 20, 40] like C and Java. In our literature search, we failed to find techniques that identified semantic code clones in dynamic typed languages. Therefore, we use an AST based comparison approach as an alternative baseline to benchmark SLACC. This leads us to the next research question:

Table 2: Projects used in this study with the number of valid submissions in both Java and Python.

Year	Problem	ID	Java	Python
2011	Irregular Cake	Y11R5P1	48	16
2012	Perfect Game	Y12R5P1	47	24
2013	Cheaters	Y13R5P1	29	19
2014	Magical Tour	Y14R5P1	46	18
	Total		170	77

Research Question 2

How effective is SLACC on semantic clone detection in dynamic typed languages?

Prior work identified code clones between languages by mapping APIs between similar languages (e.g., Java and C#) using predefined rules [11] or using an embedded API translations [4, 33]. As a result, these code clones are syntactic rather than semantic. Therefore:

Research Question 3

How effective is SLACC at cross-language semantic clone detection?

4.2 Data

We validate this study on four problems from Google Code Jam (GCJ) repository and their valid submissions in Java and Python. GCJ is an annual online coding competition hosted by Google where participants solve the programming problems provided and submit their solutions for Google to test. The submissions that pass Google's tests are considered valid and are published online. We use the first problem from the fifth round of GCJ from 2011 to 2014[2]. The details about the problem and submissions are in Table 2. Overall in this study, we consider 247 projects; 170 from Java and 77 from Python. The 170 Java GCJ submissions contain 885 methods and generated 19,188 Java functions. The 77 Python submission contains 301 methods and generated 17,215 Python functions.

The code, projects and execution scripts for the project can be found in our GitHub Repository [28].

4.3 Experimental Setup

The experiments were run on a 16 node cluster with each node having a 4-core AMD opteron processor and 32GB DDR3 1333 ECC DRAM. Our experiments have four hyper-parameters

- Minimum size of snippet (`MIN_STMT` - Section 3.1): We set this to 2 to capture snippets with interesting behavior.
- Maximum number of arguments (`ARG_MAX` - Section 3.2): This value is set to 5. Hence if a snippet has more than 5 arguments, it is omitted from the experiments.
- Number of executions (Section 3.4): We execute each snippet with 256 generated inputs (Section 3.3); see Section 6.1 for details on this choice.
- Similarity Threshold (`SIM_T` - Section 3.5): We set this to 1.0 for our experiment. This implies that two functions are only considered to be clones if for ***all*** inputs they generate the same outputs.

Sensitivity to the number of executions and `ARG_MAX` is explored and discussed in Sections 6.1 and 6.2 respectively.

4.4 Metrics

Our study uses three metrics primarily to address the research questions we pose.

- **Number of Clusters**: A cluster is a collection of functions with a common property (i.e., type I-IV similarity). This metric is the number of clusters generated by a clone detection algorithm. This is represented as |Clusters|, # Clusters or #C.
- **Number of Clones**: A function that belongs to a cluster is called a clone. This metric is the total number of functions in all the clusters generated by a clone detection algorithm. This is represented as |Clones|, # Clones or #M.
- **Number of False Positives**: A false positive is a cluster which contains one or more functions which does not adhere to the similarity measure of the cluster. This is represented as |False Positive|, # False Positives or #FP.

4.5 Baselines

To answer RQ1, RQ2, and RQ3, we use baseline techniques to illustrate the capabilities of SLACC.

4.5.1 RQ1: HitoshiIO. As a baseline, we use the closest technique to ours, HitoshiIO [40]. This tool identifies functional clones for Java Virtual Machine (JVM) based languages such as Java and Scala. It uses in-vivo clone detection and inserts instrumentation code in the form of control instructions [47] in the application's bytecode to record input and output values at runtime. Inputs and outputs are observed using the existing workloads, which allows it to observe behavior and identify clones in code for which input generators cannot generate inputs. The methods with similar values of inputs and outputs during executions are identified as functional clones. HitoshiIO considers every method in a project as a potential functional clone of every other method and returns pairs of clones. For comparison against SLACC, we group the pairs into clusters as follows: two pairs of clones are grouped into a cluster if both the pairs have a common function between them (i.e., for pairs (A,B) and (B,C), a clone cluster is created with (A,B,C)).

Like the similarity threshold `SIM_T` in SLACC, HitoshiIO has a similar parameter that provides a lower bound on how similar two methods must be to be considered a functional clone. As with SLACC, HitoshiIO also has a parameter for an upper bound on the number of IO profiles considered for each method.

We used an existing and public implementation of HitoshiIO.[3] The workload used to benchmark HitoshiIO with GCJ are the sample test input files. GCJ provides only two sample input files for

[2]Early rounds have many submissions to create a reasonably scoped experiment. Thus, we chose submissions from the quarterfinals in round five.

[3]github.com/Programming-Systems-Lab/ioclones; Commit hash: aa5b5b3; Dated: 05/06/2018

a validating a submission. However, in SLACC each method was executed 256 times. To create a balanced benchmark, we randomly fuzzed the test input files 32 times before sending it to HitoshiIO. Note that we tried fuzzing the files 256 times but the clone-detection phase of HitoshiIO crashed for large numbers of inputs.

4.5.2 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.

Table 3: Number of whole method clones identified by HitoshiIO(H), SLACC(S) and both the approaches, after accounting for false positives.

Problem	HitoshiIO(|H|)	SLACC(|S|)	|H∩S|
Irregular Cake	3	44	3
Perfect Game	4	35	4
Cheaters	4	21	4
Magical Tour	9	35	9
Total	20	135	20

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

[4]False 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.

SLACC$_{stmt}$

```
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;
}
```

SLACC$_{method}$

```
import Y12R5P1.xiaowuc.A // Parent Class A
public static int func_b(Scanner in) {
  // Y12R5P1.xiaowuc.A.next()
  while (A.tok == null || !A.tok.hasMoreTokens()) {
    A.tok = new StringTokenizer(in.readLine());
  }
  return Integer.parseInt(A.tok.nextToken());
}
```

HitoshiIO

```
public static int func_c(StreamTokenizer in) {
  // Y11R5P1.burdakovd.A.nextInt()
  in.nextToken();
  return (int) in.nval;
}
public static int func_d(StreamTokenizer in) {
  // Y11R5P1.Sammarize.Main.next()
  in.nextToken();
  return Integer.parseInt(in.nval);
}
```

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

```
public static int func_f(Scanner sc) {
  // Snipped from Y11R5P1.dooglius.A.go()
  int next = sc.nextInt();
  return next;
}
```

Figure 4: Semantic clusters detected by HitoshiIO, SLACC on method level (SLACC$_{\text{method}}$) and SLACC on statement level (SLACC$_{\text{stmt}}$). The cluster contains functions that take an object that reads a file and returns the next Integer token.

contained within the valid clusters for SLACC, ($H \equiv H \cap S$), demonstrating that among the valid clones, SLACC subsumes HitoshiIO for this experiment. However, the low precision for HitoshiIO may be due to the use of limited inputs or the execution context, so further investigation is needed for generalization of this result.

An example of a cluster that contains methods from both SLACC and HitoshiIO is shown in Figure 4. 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$_{\text{method}}$ additionally identifies two more method level clones that were not detected by HitoshiIO: `func_b` and `func_e`.

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

	Java		Python		Java + Python	
	SLACC	AST	SLACC	AST	SLACC	AST
# Clusters	632	6122	482	3971	34	616
# Valid	584	226	421	181	32	25
Precision	92.4	3.7	87.3	4.6	94.1	4.1

5.1.2 Statement level clones. Additionally, SLACC identifies 624 clusters with 4,038 statement level code clones. Of these, 48 clusters are false positives (precision=92.3%). The large number of code clones is intuitive because each method can contain multiple modular functionalities. That said, it should be noted that the higher precision for statement level clusters would lead us to believe that detecting clones for succinct behavior is more accurate.

Statement level clones can be clustered with whole method clones. For example, in Figure 4, SLACC$_{\text{stmt}}$ represents a SLACC cluster based on partial methods: `func_a` and `func_f` are functions segmented from the `main` method in class `Y14R5P1.stolis.MMT3` and the go method in `Y11R5P1.dooglius.A`, respectively.

> **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.

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

```
def func_db8e(a):
    n = len(a)
    sum0 = [0] * (n + 1)
    for i in xrange(n):
        sum0[i + 1] = sum0[i] + a[i]
    allv = sum0[-1]
    return allv
```

```
def func_43df(items):
    _sum = sum(items)
    j = len(items) - 1
    return _sum
```

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

```
static long func_3b0e (Long[] x2) {
    Long res = null;
    Long[] arr = x2;
    int len = arr.length;
    for (int i = 0; i < len; ++i) {
        long xx = arr[i];
        if (xx >= res)
            continue;
        res = xx;
    }
    return res;
}
```

```
def func_6437 (y):
    ymin = min (y)
    count = 0
    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.

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.

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

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

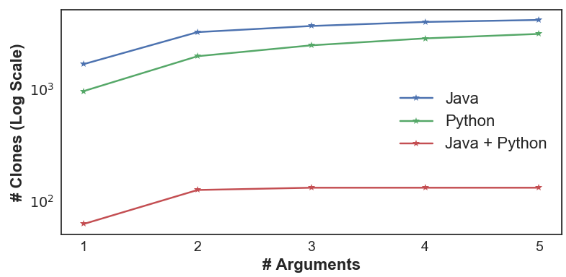

Figure 7: Cumulative # clones with # arguments varying between 1-5.

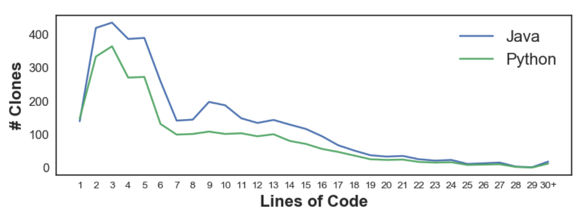

Figure 8: # clones for lines of code between ranging from 1-29. Clones with 30 or more lines are grouped into 30+

a given number of inputs, each row represents the mean and variance (in parenthesis) of the number of clones, clusters and false positives. For low numbers of inputs, we see more functions being marked as clones and fewer clusters. As the number of inputs increases, the number of clones reduces and the number of clusters increases, demonstrating that the additional inputs are critical at differentiating behavior between functions. The counts of clones, clusters, and false positives appear to plateau after 64 inputs. This highlights that 10 inputs used by Jiang and Su would not be sufficient for optimally identifying true functional clones and will lead to a large number of false positives, as suggested in prior work [9].

6.2 Influence of arguments in clones

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 modular 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 predefined 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

[1] Brenda S Baker. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*. IEEE, 86–95.

[2] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 306–317.

[3] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 368–377.
[4] Jonathan Beit-Aharon. 2002. Source code translation. US Patent App. 15/894,096.
[5] Stephen W Bowles and George E Bethke Jr. 1983. Multi-pass system and method for source to source code translation. US Patent 4,374,408.
[6] Elizabeth Burd and John Bailey. 2002. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 36–43.
[7] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 238–252.
[8] Yingnong Dang, Dongmei Zhang, Song Ge, Ray Huang, Chengyun Chu, and Tao Xie. 2017. Transferring Code-clone Detection and Analysis to Practice. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track* (Buenos Aires, Argentina) *(ICSE-SEIP '17)*. IEEE Press, Piscataway, NJ, USA, 53–62. https://doi.org/10.1109/ICSE-SEIP.2017.6
[9] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Stefan Wagner. 2012. Challenges of the dynamic detection of functionally similar code fragments. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 299–308.
[10] Rochelle Elva and Gary T Leavens. 2012. *Jsctracker: A semantic clone detection tool for java code*. Technical Report. University of Central Florida, Dept. of EECS, CS division.
[11] Alexandre Fau and Reinhold Bihler. [n.d.]. Java2CSharp. http://sourceforge.net/projects/j2cstranslator/. Accessed: 2018-09-25.
[12] Hans-Christian Fjeldberg. 2008. *Polyglot programming*. Ph.D. Dissertation. Master thesis, Norwegian University of Science and Technology, Trondheim/Norway.
[13] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 416–419.
[14] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*. ACM, 321–330.
[15] Google. [n.d.]. Google Code Jam. code.google.com/codejam. Accessed: 2018-09-25.
[16] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 72–82.
[17] Diwaker Gupta. 2004. What is a good first programming language? *Crossroads* 10, 4 (2004), 7–7.
[18] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*. Springer, 238–255.
[19] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.
[20] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 81–92.
[21] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.
[22] Marcus Kessel and Colin Atkinson. 2019. On the Efficacy of Dynamic Behavior Comparison for Judging Functional Equivalence. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 193–203.
[23] Mohd Ehmer Khan, Farmeena Khan, et al. 2012. A comparative study of white box, black box and grey box testing techniques. *Int. J. Adv. Comput. Sci. Appl* 3, 6 (2012).
[24] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. 2011. MeCC: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 301–310.
[25] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*. IEEE, 253–262.
[26] Jingyue Li and Michael D Ernst. 2012. CBCD: Cloned buggy code detector. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 310–320.
[27] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code.. In *OSdi*, Vol. 4. 289–302.
[28] George Mathew, Chris Parnin, and Kathryn T Stolee. [n.d.]. SLACC. github.com/DynamicCodeSearch/SLACC/tree/ICSE20. [Online; accessed 06-February-2020].
[29] Philip Mayer and Alexander Bauer. 2015. An Empirical Analysis of the Utilization of Multiple Programming Languages in Open Source Projects. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering* (Nanjing, China) *(EASE '15)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/2745802.2745805
[30] Philip Mayer, Michael Kirsch, and Minh Anh Le. 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development* 5, 1 (19 Apr 2017), 1. https://doi.org/10.1186/s40411-017-0035-z
[31] Narcisa Andreea Milea, Lingxiao Jiang, and Siau-Cheng Khoo. 2014. Scalable detection of missed cross-function refactorings. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 138–148.
[32] Kawser Nafi, Tonny Sheka Kar, Banani Roy, Chanchal K. Roy, and Kevin Schneider. [n.d.]. CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation. ([n. d.]).
[33] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE, 438–449.
[34] Python Community. [n.d.]. Python AST. docs.python.org/3/library/ast.html. [Online; accessed 23-August-2019].
[35] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. 2013. Detecting and Characterizing Semantic Inconsistencies in Ported Code. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Silicon Valley, CA, USA) *(ASE'13)*. IEEE Press, Piscataway, NJ, USA, 367–377. https://doi.org/10.1109/ASE.2013.6693095
[36] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming* 74, 7 (2009), 470–495.
[37] Jean Scholtz and Susan Wiedenbeck. 1990. Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction* 2, 1 (1990), 51–72. https://doi.org/10.1080/10447319009525970
[38] N. Shrestha, T. Barik, and C. Parnin. 2018. It's Like Python But: Towards Supporting Transfer of Programming Language Knowledge. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 177–185. https://doi.org/10.1109/VLHCC.2018.8506508
[39] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code relatives: detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 702–714.
[40] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. 2016. Identifying functionally similar code in complex codebases. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 1–10.
[41] Team GitHub. [n.d.]. GitHub Gist. https://gist.github.com/discover. [Online; accessed 23-August-2019].
[42] Team Stack Overflow. [n.d.]. Stack Overflow. https://stackoverflow.com. [Online; accessed 23-August-2019].
[43] Federico Tomassetti and Marco Torchiano. 2014. An Empirical Assessment of Polyglot-ism in GitHub. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) *(EASE '14)*. ACM, New York, NY, USA, Article 17, 4 pages. https://doi.org/10.1145/2601248.2601269
[44] Danny van Bruggen. 2015. Javaparser - For processing Java code. github.com/javaparser/javaparser. [Online; accessed 23-August-2019].
[45] Andrew Walenstein and Arun Lakhotia. 2007. The software similarity problem in malware analysis. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
[46] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.
[47] Wikipedia Contributors. [n.d.]. Java bytecode instruction listings. en.wikipedia.org/wiki/Java_bytecode. [Online; accessed 23-August-2019].
[48] Wikipedia contributors. 2019. Levenshtein distance — Wikipedia, The Free Encyclopedia. en.wikipedia.org/wiki/Levenshtein_distance. [Online; accessed 23-August-2019].
[49] Quanfeng Wu and John R. Anderson. 1990. *Problem-solving transfer among programming languages*. Technical Report. Carnegie Mellon University.
[50] Marvin Wyrich, Daniel Graziotin, and Stefan Wagner. 2019. A theory on individual characteristics of successful coding challenge solvers. *PeerJ Computer Science* 5 (Feb. 2019), e173. https://doi.org/10.7717/peerj-cs.173
[51] Wuu Yang. 1991. Identifying syntactic differences between two programs. *Software: Practice and Experience* 21, 7 (1991), 739–755.
[52] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler. 2018. Automatic Clone Recommendation for Refactoring Based on the Present and the Past. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 115–126. https://doi.org/10.1109/ICSME.2018.00021