srcClone: Detecting Code Clones via Decompositional Slicing

Hakam W. Alomari and Matthew Stephan
{alomarhw,stephamd}@miamioh.edu
Miami University
Oxford, Ohio, USA

ABSTRACT

Detecting code clones is an established method for comprehending and maintaining systems. One important but challenging form of code clone detection involves detecting semantic clones, which are those that are semantically similar code segments that differ syntactically. Existing approaches to semantic clone detection do not scale well to large code bases and have room for improvement in their precision and recall. In this paper, we present a scalable slicing-based approach for detecting code clones, including semantic clones. We determine code segment similarity based on their corresponding program slices. We take advantage of a lightweight, publicly available, and scalable program slicing approach to compute the necessary information. Our approach uses dependency analysis to find and measure cloned elements, and provides insights into elements of the code that are affected by an entire clone set/-class. We have implemented our approach as a tool called srcClone. We evaluate it by comparing it to two semantic clone detectors in terms of clones, performance, and scalability; and perform recall and precision analysis using established benchmark scenarios. In our evaluation, we illustrate our approach is both relatively scalable and accurate. srcClone can also be used by program analysts to run on non-compilable and incomplete source code, which serves comprehension and maintenance tasks very well. We believe our approach is an important advancement in program comprehension that can help improve clone detection practices and provide developers greater insights into their software.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Code clone, Clone detection, Program slicing, Semantic clones

ACM Reference Format:
Hakam W. Alomari and Matthew Stephan. 2020. srcClone: Detecting Code Clones via Decompositional Slicing. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389271

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389271

1 INTRODUCTION

Comprehending and maintaining evolving large software systems is an ongoing challenge. Clone detection is one approach that can assist by identifying recurring patterns of use. There are many approaches for clone detection [6, 30]. The clones detected by each approach are fundamentally constricted by their respective underlying clone definitions, the similarity measures they use, and the level of analysis they apply to the source code during the detection process.

Textual-based techniques use three levels of analysis during the clone detection process: *textual*, *lexical*, and *syntactical*, which results in Type-1, Type-2, and Type-3 clones [33]. Textual-based techniques are capable of finding clones only within a program's contiguous and structured syntax. They are not able to find semantically equivalent clones that differ significantly in structure since they are not sensitive to non-contiguous, reordered, and intertwined clones [19, 23]. Functional-based techniques use a semantic analysis level to analyze source code, and their clones are known as Type-4 clones.

Semantic clone identification is challenging as it can be difficult to determine precisely semantically similar components, and is, in general, undecidable [11, 15]. As such, semantic approaches generally use a less strict definition of semantically similar code segments. Most approaches explore abstractions of the program semantics to guide the clone detection process, such as Control Flow Graphs (CFGs) [1] or Program Dependence Graphs (PDGs) [10]. The clone detection process is thus turned into the problem of finding isomorphic sub-graphs, which is NP-hard, requiring existing algorithms use approximative solutions [11, 19, 22].

As a motivating example, consider the code snippet in Figure 1a compared to the similar code snippet in Figure 1b from Roy et al. [33]. Both perform the same overall computation, but Figure 1b replaces a control statement. Both are implemented by different syntactic variants. Current textual-based techniques, and most existing functional-based techniques, are unable to detect the semantically similar clone scenario proposed in Figure 1b [32, 33].

To help understand semantic similarities, program slicing [40] is a widely-used and well-known approach for comprehending and

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod =1.0;
4 for (int i=1;i<=n;i++)
5     {sum=sum + i;
6     prod = prod * i;
7     foo(sum, prod); }}
```

(a) Example Code Snippet

```
1 void sumProd_E(int n) {
2 float sum=0.0; //C1
3 float prod = 1.0;
4 int i=0;
5  while (i<=n)
6   { sum=sum + i;
7     prod = prod * i;
8     foo(sum, prod);
9     i++; }}
```

(b) Semantically Similar Code

Figure 1: Example Motivation Proposed by Roy et al. [33].

detecting semantic properties of software [3, 11, 19, 41]. Generally, program slicing is time consuming since it is based on PDG. Building the PDG is quite costly in terms of computational time and space. Thus, PDG-based approaches generally do not scale well. Generating slices for a very large system can often take days of computing time [2]. However, one available program slicer by Alomari et al., srcSlice [2, 28], is able to generate slices much faster as it is not PDG-based.

This paper describes our design and initial implementation of a slice-based clone detection method and corresponding tool, srcClone, for large scale C/C++ software systems. We employ srcSlice to compute specifically tailored numerical vectors to represent the structural information within slices, which we then hash and cluster. We consider two code segments semantically similar if their slices are similar and they represent a candidate clone pair.

This work advances the knowledge and practice of program comprehension through clone detection in different ways. Our approach (1) is scalable and robust for detecting Type-4 clones, (2) is more resilient to differences in syntax and able to find non-contiguous clones, (3) finds clones that are relevant and meaningful computations through means of slicing variable, and (4) discovers data and control dependencies that exist within slices to detect not only cloned code but also the code that is impacted by clones.

We organize the remainder of this paper as follows. Section 2 introduces background information on program slicing and srcSlice. Section 3 describes our slice-based clone detection process. Section 4 discusses our evaluation. Section 5 presents and contrasts related work. Finally, Section 6 summarizes the paper and discusses our ideas for future work.

2 PROGRAM SLICING AND SRCSLICE

This section provides the necessary background information on program slicing and srcSlice's slicing results. We focus specifically on srcSlice's properties to assist in our description of how we employ it to detect clones.

Weiser originally defined a "slice" as an executable program that preserves the behavior of the original program [40]. Their algorithm traces data and control dependencies for determining the direct and indirect relevant variables and statements. Slicing techniques are distinguished broadly according to the type of slices they can compute [36, 39]. A common limitation of these techniques is using PDGs to compute slices, which are quite computationally costly and space intensive. Thus, slicing generally does not scale well.

The slicing tool we employ for srcClone, srcSlice [28], addresses this limitation by eliminating the time and effort needed to build entire PDGs. It combines a text-based approach with a lightweight static analysis XML infrastructure, srcML [8]. This format provides direct access to abstract syntactic information to support static analysis. This information is used by srcSlice to identify program dependencies as needed when computing the slice.

srcSlice implements a forward, decomposition, inter-procedural, static slicing technique. The forward slice from a specific program point includes all program points in the forward control flow affected by the computation at that point. srcSlice uses the initial variable declaration as the starting point. This approach varies from traditional definitions in that it does not require a precise reference to a location or statement number in the source, rather only a variable. Specifically, the slicing criterion specified as a triple of the file, function, and variable names. This is a forward decomposition slice [2], which can be viewed as the union of a collection of static forward slices taken at a set of statements that (re)define a variable.

srcSlice computes a slice profile for each identifier line by line as they are encountered. The slice profile for an identifier contains all data gathered about that identifier during the slicing process. Since srcSlice uses a forward decomposition slicing definition, these data include all lines of code transitively affected by the value of the identifier along data and control dependencies. The following is a list of that data,

- `File`, `function`, and `type`, `variable`: names of the file/function the variable is in and its type, respectively.
- `Def`: list of lines a variable is defined or redefined on. Def is used to differentiate between variables with the same name but in differing scopes.
- `Use`: list of lines a variable is used. This refers to a variable's value being used in a computation with no modification to its value. Can be used to construct def-use chains.
- `Dvars`: list of variables that are data dependent on slicing variable.
- `Ptrs`: list of aliases of slicing variable. The elements of this list are variables to which the slicing variable is a pointer.
- `Cfuncs`: list of functions called using the slicing variable.

srcSlice produces a system dictionary of all the slice profiles of all variables. It is three-tiered and consists of three maps: files to functions, functions to variable names, and variable names to slice profiles. srcSlice is very memory efficient and fast. On the Linux kernel version 4.06 with $\approx$ 13 MLOC, srcSlice computes around 2M slice profiles within 7 minutes [28]. More formally, each slice profile, denoted by *SP*, is constructed based on the following definition of forward decomposition slice:

Definition 2.1 (Forward Decomposition Slice). A forward decomposition slice, denoted by ds, of a program P is constructed with respect to a given file f, a given function/method m in f, and a given variable v in m. It consists of the union of all static forward slices, denoted by fs, constructed for the criteria $\{(\{v\}, s_1), ..., (\{v\}, s_k)\}$, where $s_1, ..., s_k$ is the set of statements in P that define/redefine v. It is defined as:

$$SP(v) = ds(f, m, v) = \bigcup_{s \in \{s_1 \dots s_k\}} fs_{(v,s)}(P). \tag{1}$$

This definition can be generalized to cater to a set of variables and functions. For a given function m with a total number of variables equal to d, the slice profile is given as:

$$SP(m) = ds(f, m) = \bigcup_{i=1}^{d} ds(f, m, v_i) \tag{2}$$

Finally, for a given file f with a total number of functions/methods equal to y, the slice profile is given as:

$$SP(f) = ds(f) = \bigcup_{i=1}^{y} ds(f, m_i) \tag{3}$$

Eqs. (2) and (3) are what we use to make larger code segments in the context of clone detection. Table 1 shows an example of using Eq.(1) to compute the *SPs* for code snippets in Figure 1a and Figure 1b. Table 2 shows the *SPs* computed for the same snippets

Table 1: Slice Profiles for Code Snippets in Figure 1a and Figure 1b. Profiles are Derived Using Eq.1.

Fig.	Vars	Def	Use	Dvars	Ptrs	Cfuncs
1a	i	{4, 4}	{4, 5, 6}	{prod, sum}	{ }	{ }
	prod	{3, 6}	{7}	{ }	{ }	{foo{2}}
	sum	{2, 5}	{7}	{ }	{ }	{foo{1}}
	n	{1}	{4}	{ }	{ }	{ }
1b	i	{4, 9}	{5, 6, 7}	{prod, sum}	{ }	{ }
	prod	{3, 7}	{8}	{ }	{ }	{foo{2}}
	sum	{2, 6}	{8}	{ }	{ }	{foo{1}}
	n	{1}	{5}	{ }	{ }	{ }

Table 2: Slice Profiles for Code Snippets in Figure 1a and Figure 1b. Profiles are Derived Using Eq.2.

Fig.	Vars	Def	Use	Dvars	Ptrs	Cfuncs
1a	*all*	{1-6}	{4-7}	{prod,sum}	{ }	{foo{1,2}}
1b	*all*	{1-4,6,7,9}	{5-8}	{prod,sum}	{ }	{foo{1,2}}

at the function level using Eq.(2). Note, we do not use Eq.(3) in this particular example since each file contains just one function.

3 CLONE DETECTION USING SLICING

In this section, we present the main steps of our slice-based clone detection approach. The main idea is to detect code clones based on the similarity between their slicing information. To achieve this goal, we first convert the source code into srcML format, which allows srcSlice to gather data about every file, function, and variable throughout the system. It stores this information in a three-tier dictionary and represents that dictionary as a set of slice profiles for all variables in the system. We then encode slice profiles into slicing vectors by means of its embedded slicing information. We search and identify clones at the function and file levels based on these vectors, and compute them at the variable level. We follow this with an efficient hashing and near-neighbor algorithm to cluster these vectors with respect to distances between them. By categorizing vectors with hash values, only those vectors with the same hash value need be compared. In this way, we minimized the number of comparisons by a factor of the hash values generated.

3.1 Formal Definitions

Many works define code clones in various ways [29]. Baxter et al. [5] notes "*clones are segments of code that are similar according to some definition of similarity*". To identify a segment as a clone, we should allow modifications to some degree. However, if the degree of modifications allowed is too large, then at some point everything will be a clone. Thus, the similarity measure used determines the types of clones we detect.

Another issue we consider is the size of the code segment, or more precisely, the minimum size of the code segment that we should consider to be worthwhile to examine. This minimal size is considered differently in different contexts. For example, Nicad [31] uses a structured block that is at least 6 LOC, Deckard [15] uses a structured sub-tree, and others uses the whole function, PDG subgraphs, or begin-end blocks of a certain minimum size [6, 29, 33]. This is important, since very small or very large code segments may end up with a high rate of false positives.

To explicate and help clarify, this paper uses the following definitions of code segments and clone types.

Definition 3.1 (Code Segment). A code segment, *CS*, is a set of lines of code, not necessarily contiguous, as computed by $SP(v)$, $SP(m)$, or $SP(f)$ using Eqs. (1), (2), or (3), respectively.

Definition 3.2 (Clone Types). We follow the standard definition used in the literature [33]:

(1) `Type-1`: *CSs* that are exactly identical, except for minor differences in whitespace, layout and comments.
(2) `Type-2`: *CSs* that have similar syntactic structures, except for differences in identifiers, literals, types, whitespace, layout and comments.
(3) `Type-3`: *CSs* with further modifications, including added, modified and/or removed statements, in addition to differences in identifiers, literals, types, whitespace, layout and comments.
(4) `Type-4`: *CSs* that perform the same computation but have different syntactic structures/implementations.

The first three types are commonly detected based on the similarity of their program text. Type-1 is often referred to as an *exact* clone, whereas both Type-2 and Type-3 are known as *near-miss* clones. Type-4 does not require the *CSs* to have any similar code, but only the same computation, which can be detected using their functional similarity. This type of *semantic* clone is generally out of scope of much of the clone detection research, with existing approaches struggling to detect this type effectively [11, 19, 22]. In this paper, we view clones as semantically similar code segments. We detect these semantic aspects using program slicing. Thus, it is important to define our notion of similar slices.

Definition 3.3 (Slice Profiles Similarity). Two slice profiles, SP_1 and SP_2, are similar if their corresponding slicing vectors, sv_1 and sv_2, are σ-similar, for a specified threshold σ.

Definition 3.4 (Slicing Vectors Similarity). Two slicing vectors, sv_1 and sv_2, are σ-similar for a given threshold σ, if $\mathcal{S}(sv_1, sv_2) \leq \sigma$.

$\mathcal{S}$ refers to a similarity function. Deckard [15] uses tree edit distance as a similarity function, CloneDr [5] uses a size-sensitive definition on trees called *Similarity*, and Gabel [11] uses a mapping function that maps a sequence of syntax to a PDG subgraph. Since our approach is not a PDG- nor Tree-based, we use a Hamming distance metric on d-dimensional slicing vectors. We identify two *CSs* as a *clone pair* if their corresponding *SPs* are similar. A group of similar *CSs* form a *clone class*.

3.2 Slice Profile Vectorization

We introduce *slicing vectors* to capture the structural information of slice profiles. This is an important and a key step in our approach. The slicing vector of a slice profile is a point of uniform dimensions in the space. This is similar to the *characteristic vector* introduced by Deckard's [15] approach. A characteristic vector is a numerical approximation of a particular sub-tree. The dimensions of a characteristic vector are determined by the total number of possible types of complete binary trees needed to approximate a given tree. Deckard generates vectors with a post-order traversal of the parse tree. We extend this work in multiple ways. Firstly, each slicing vector has a fixed size that is based on the number of slicing fields we

Table 3: Slicing Vectors for Variable and Function Levels for the Slice Profiles (*SPs*) Computed in Table 1 and Table 2.

Function	Variable	Slicing Vectors (*svs*)	
		Variable-level	Function-level
sumProd	i	⟨2, 3, 2, 0, 0⟩	⟨**6**, 4, 2, 0, 1, 4⟩
	prod	⟨2, 1, 0, 0, 1⟩	
	sum	⟨2, 1, 0, 0, 1⟩	
	n	⟨1, 1, 0, 0, 0⟩	
sumProd_E	i	⟨2, 3, 2, 0, 0⟩	⟨**7**, 4, 2, 0, 1, 4⟩
	prod	⟨2, 1, 0, 0, 1⟩	
	sum	⟨2, 1, 0, 0, 1⟩	
	n	⟨1, 1, 0, 0, 0⟩	

calculate using srcClone. Secondly, there is no tree to traverse in order to generate these vectors and there is no need to approximate the values of the vector's dimensions. The slicing vector, denoted by *sv*, for a given variable's slice profile, has following dimensions:

$$sv(v) = \langle |Def|, |Use|, |Dvars|, |Ptrs|, |Cfuncs| \rangle \quad (4)$$

Each dimension represents the size of one of the slicing fields. For example, the $|Def|$ dimension represents the number of lines of code an identifier is defined or redefined on. If two code segments are cloned, their vectors will be very similar. Intuitively, even if a clone modified a small part of the original copy, their slicing vectors will not change greatly from a slicing perspective. This is the only necessary information that remains about the variable after this encoding step. Thus, neither the structure of the code nor other information, such as the names of the variables, will remain. We are still able to detect Type-2 clones effectively despite this however, because if two variables have the same *sv*, but their names are different, we still detect these clones via the slice profile.

As an example, Table 3 shows the slicing vectors of the slice profiles we computed in Table 1 and Table 2. At the variable level, the *sv* for each variable in function *sumProd* is equal to its corresponding *sv* for the same variable in function *sumProd_E*.

We have thus far reduced slice profiles to a set of vectors. We compose a method of comparing *n* slicing vectors, where *n* is the number of variables. In the process of comparing two methods, instead of comparing *n* slicing vectors we can use Eq.(2) to create a unique slice profile for each method, then encode the slice profile in the same way we did for a variable. We lastly then add one more dimension at the end to count the number of variables in that method. The *sv* for $SP(m)$, has six dimensions:

$$sv(m) = \langle |Def|, ..., |Cfuncs|, |SP(v)| \rangle \quad (5)$$

We repeat the same process for each file using Eq.(3) by adding a new dimension to represent the number of methods in a file. The slicing vector for a given $SP(f)$ has seven dimensions:

$$sv(f) = \langle |Def|, ..., |Cfuncs|, |SP(v)|, |SP(m)| \rangle \quad (6)$$

For example, as shown in Table 3, the slicing vectors for functions *sumProd* and *sumProd_E* are ⟨**6**, 4, 2, 0, 1, 4⟩ and ⟨**7**, 4, 2, 0, 1, 4⟩, respectively. There is no difference in the number of variables in each function. However, there are different numbers of definition statements: 7 in *sumProd_E* instead of 6 in *sumProd*. This is due to us using Eq.(2) to compute the slice profiles. The union between the Def sets in both functions eliminate repeated lines of code, in this case statement number 4 in the $SP(i)$, as shown in Table 1.

Given a system dictionary, we perform a single pass to generate vectors for its slice profiles. Algorithm 1 shows how we generate vectors for the three granularity levels of slice profiles. srcClone returns the $\mathcal{G}$ set with all *svs* in the system.

Algorithm 1: Slicing Vectors Generation

Input: $\mathcal{S}_\mathcal{D}$: system dictionary
Output: $\mathcal{G}$: slicing vectors set
/* $sv(v), sv(m), sv(f)$ generation */
1 **begin**
2 $\mathcal{F} \longleftarrow$ Set of files in $\mathcal{S}_\mathcal{D}$
3 $\mathcal{M} \longleftarrow$ Set of methods for each file
4 $\mathcal{V} \longleftarrow$ Set of variables for each method
5 **for** $\forall sp(f) \in \mathcal{F}$ **do**
6 **for** $\forall sp(m) \in \mathcal{M}$ **do**
7 **for** $\forall sp(v) \in \mathcal{V}$ **do**
8 $sv(v) \leftarrow \langle |Def|, ..., |Cfuncs| \rangle$
9 $sv(m) \leftarrow \langle |Def|, ..., |Cfuncs|, |\mathcal{V}| \rangle$
10 $sv(f) \leftarrow \langle |Def|, ..., |Cfuncs|, |\mathcal{V}|, |\mathcal{M}| \rangle$
11 $\mathcal{G} \leftarrow \mathcal{G} \bigcup (sv(f), sv(m), sv(v))$

3.3 Similarity and Matching

To find clones using slicing vectors, we compare every vector to every other vector for equality. Several issues arise however, such as near-miss clone detection, dimension similarity, and scalability.

We handle near-miss clones by comparing vectors looking for similarity rather than exact equality. Specifically, rather than matching entire slicing vectors, we compare instead dimensions for similarity. Pairwise comparisons are computationally infeasible for similarity detection, especially, in large software systems. Thus, we focus on pairs that are likely to be similar only rather than investigating every pair. There is a general theory of how to provide such focus, called Locality Sensitive Hashing (LSH) [14], which we employ in our approach and corresponding tool. LSH is a method for determining which items in a given set are similar. Rather than comparing all pairs of items within a set, items are hashed into buckets such that similar items will be more likely to hash into the same buckets. As a result, the number of comparisons are reduced. LSH has been used successfully by some AST-based clone detection approaches [11, 15] to cluster a large number of sub-trees vectors, and we employ it analogously for srcClone.

In our case, we specifically adapt LSH to hash the slicing vectors, find the near neighbor sets, and generate the clone reports. Slicing vectors similar to each other are located in the same buckets with high probability, while dissimilar vectors are likely to be in different buckets. This makes it easier to identify clones with various degrees of similarity. Our method is capable of enumerating clones from millions of vectors in a few minutes, with extremely low false positive rate, as we present in Section 4. We define our LSH implementation as follows,

Definition 3.5 ((r_1, r_2, p_1, p_2)**-Locality Sensitive Hashing).** Let $\mathcal{D}$ be a distance measure, and let $r_1 < r_2$ be two distances in this

measure. Given a set of slicing vectors $\mathcal{G}$, and a collision probability values $p_1 > p_2$, then a family of hash functions $\mathcal{H}$ is said to be (r_1, r_2, p_1, p_2)-sensitive, if for every $v_i, v_j \in \mathcal{G}$ the following two conditions hold;

$$\begin{cases} \text{if } \mathcal{D}(v_i, v_j) \leq r_1, \text{then Prob}_{h\in\mathcal{H}}\,[h(v_i) = h(v_j)] \geq p_1, \\ \text{if } \mathcal{D}(v_i, v_j) \geq r_2, \text{then Prob}_{h\in\mathcal{H}}\,[h(v_i) = h(v_j)] \leq p_2. \end{cases}$$

In our implementation, a smaller distance between vectors corresponds to a higher probability of similarity. Recall that a distance should satisfy three properties: non-negativity, symmetry, and triangle inequality, for example, Euclidean distance and Hamming distance. Note that not every distance measure may have a corresponding LSH family. We use the Hamming distance as a metric over the d-dimensional vectors. We define Hamming distance in our context as follows,

Definition 3.6 (Hamming Distance). Given a set of vectors $\mathcal{G}$, let $v_1 = \langle x_1, ..., x_d \rangle$ and $v_2 = \langle y_1, ..., y_d \rangle$ be two d-dimensional vectors $\in \mathcal{G}$. The Hamming distance of v_1 and v_2, $\mathcal{D}_{\mathcal{H}}(v_1, v_2) = \sum_{i=1}^{d} \delta(x_i, y_i)$, where $\delta(x_i, y_i) = 0$, if $x_i = y_i$, and 1, if $x_i \neq y_i$.

The Hamming distance is the number of dimensions in which v_1 and v_2 are different. Now, suppose two vectors v_1 and v_2 are identical. The hamming distance is equal to *zero*, which means that the probability of $h(v_1) = h(v_2)$ is high and is equal to the number of dimension agreements out of the total number of dimensions. In this case, *one*. Since vectors v_1 and v_2 disagree in $\mathcal{D}_{\mathcal{H}}(v_1, v_2)$ positions out of d positions, then they agree in $d - \mathcal{D}_{\mathcal{H}}(v_1, v_2)$ positions. Hence Prob $[h(v_1) = h(v_2)] = 1 - \mathcal{D}_{\mathcal{H}}(v_1, v_2)/d$.

Lemma 3.1. For any $r_1 < r_2$, $\mathcal{H}$ is a $(r_1, r_2, 1 - \frac{r_1}{d}, 1 - \frac{r_2}{d})$-sensitive family of hash functions.

Proof. Recall Definition 3.5. Let $p_1 = 1 - r_1/d$, and $p_2 = 1 - r_2/d$. A family $\mathcal{H}$ of hash functions $h : \mathcal{G} \rightarrow \mathcal{U}$ is called (r_1, r_2, p_1, p_2)-sensitive, if for every $v_i, v_j \in \mathcal{G}$ the following two conditions hold;

$$\begin{cases} \text{if } \mathcal{D}_{\mathcal{H}}(v_i, v_j) \leq r_1, \text{then Prob}_{h\in\mathcal{H}}\,[h(v_i) = h(v_j)] \geq p_1, \\ \text{if } \mathcal{D}_{\mathcal{H}}(v_i, v_j) \geq r_2, \text{then Prob}_{h\in\mathcal{H}}\,[h(v_i) = h(v_j)] \leq p_2. \end{cases}$$ □

We use this LSH to hash vectors with respect to the hamming distances among them, so that vectors near each other, at distance $\leq r_1$, are located in the same buckets with high probability of collision, while vectors far from each other (at distance $\geq r_2$) are likely to be in different buckets. We consider any pair that have the same hash value to be a candidate clone pair. Since we are comparing similarity rather than equality, we use various degrees of similarity thresholds to specify how similar two slicing vectors should be. We have thus encoded slices using numerical vectors and reduced the similarity problem to detecting similar vectors. We now explain how we hash vectors and cluster similar vectors.

3.4 Hashing and Clustering

Our modified LSH algorithm helps find near neighbors of a given query vector v efficiently. In our near neighbor search problem, we are given a set $\mathcal{G}$ of n slicing vectors, and the goal is to build a data structure that reports any vector within a given distance r to v. However, since some existing solutions to this problem suffer from the curse of dimensionality, researchers proposed approximation algorithms for the problem, such as the (c, r)-Approximate Near Neighbor algorithm (ANN) [14], in which a data structure may return any vector whose distance from the query vector is at most cr, for an approximation factor $c > 1$, provided that there exists a vector within distance r from the query vector v. More formally,

Definition 3.7 (**(c, r)-Approximate Near Neighbor).** Given a query vector v, a set of vectors $\mathcal{G}$ of size n, a distance r, and a factor $c > 1$, $\mathcal{U} = \{u \in \mathcal{G} \mid \mathcal{D}_{\mathcal{H}}(u, v) \leq cr\}$ is called an cr-ANN set of v, and any $u \in \mathcal{U}$ is a (c, r)-approximate near neighbor of v.

Algorithm 2: LSH Hashing and Detection

```
   Input: G: vectors, r : distance, p1 : probability
   Output: cr-ANN: near neighbor set
1  cr-ANN ⟵ ϕ
2  LSH(G, r, p1)
3  for each v ∈ G do
4      N ← queryLSH(v)
5      if N.size() > 1 then
6          cr-ANN ← cr-ANN ⋃ N
7      else
8          delete N
9  return cr-ANN
```

Definition 3.7 implies that we post-process $\mathcal{G}$ to create a data structure, cr-ANN, that contains clustered similar vectors. We show our hashing process and our algorithm for finding the cr-ANN set for each vector in Algorithm 2. The distance r is the largest distance allowed between a vector and its neighbors, that is, the threshold σ defined in Definition 3.4. We store all vectors into LSH hash tables (line 2). We feed r and p_1 (the minimal probability) to LSH, and then compute other parameters automatically and in optimal running time. We then use a vector v as a query point to get a cr-ANN set (line 4). Finally, we return the cr-ANN sets for all vectors to generate clone reports (line 9). If the cr-ANN set just contains the query vector v, which means there are no neighbors within distance r, then we delete it (line 8). These deleted sets represent newly added variables or methods in a cloned segment that do not have a match in the original code segment.

3.5 Approach Properties

As per other near-miss clone detectors, our slice-based approach is not affected by the formatting and layout differences between code segments. The slice profile includes all directly or indirectly related statements only. Unlike most techniques, our approach allows different sizes of *CSs* to be compared at different granularity levels. A *CSs* to be compared in our approach may contain one *SP*, or several *SPs*. In our method, there is no token, string, tree, or graph to compare.

As we show in Table 3, the similarity of all slicing vectors at the variable level for both functions are identical. This means all vectors have the same hash values and represent a clone class. But can we consider both functions identical? If we compare vectors at the function level, both also have the same hash value with $\mathcal{D}_{\mathcal{H}} \approx 0.84$. If we consider a similarity threshold r to be equal 1 (both vectors differ in one position), then p_1 will equal 0.8, and both functions form a clone pair. We can alternatively say that both functions are

similar since all vectors at the variable levels are similar. However, this is not the case when we have new variables added to the cloned function. Therefore, we delete those variables using Algorithm 2. In fact, using the Type-4 definition of clones where only semantic similarity is considered, these functions are indeed clones and form a clone pair.

3.5.1 ***Normalization and Transformation***. In our slice-based method, there is no need for normalizing transformations as the case in many clone detection tools [33]. Although, almost all clone detection approaches remove and ignore whitespace and comments in the actual comparison, some approaches apply identifier normalization and/or reorganizing the source code to a standard form before the comparison process.

To better understand why we do not require normalization, let us consider the example of a function call, say `foo(sum, prod)` in Figure 1a. With typical text-based and line-based approaches, this simple call would either be normalized as `id()` or as `id(id, id)`. In the first normalization, all function calls in the code are treated as the same and thus a typical approach may generate false positives. In the second normalization, all function calls with two parameters will match, again generating many false positives and at the same time missing some potential matches, such as overloaded functions in C++ when only the number of parameters is changed, for example, `foo(sum)`. Nicad's group [31], discussed this example. They consider that at least the function name or the number of parameters should be the same to have a match with other function calls. As a result, it considers both function calls neither similar, by normalizing both to `id()`, nor dissimilar, by normalizing the first call to `id(id, id)` and the second call to `id(id)`.

In contrast, using slice-based detection, the first function call, `foo(sum, prod)`, generates two slice profiles, one for `sum` and one for `prod`. The *Cfuncs* field in the slice profile of `sum` contains `foo{1}`, and the slice profile of `prod` contains `foo{2}`. In the second function call, `foo(sum)`, we have just one slice profile for the `sum` variable and its *Cfuncs* field contains `foo{1}`. Therefore, both function calls are similar from the variable `sum` perspective.

3.5.2 ***Type-1 and Type-2 Detection***. To understand the effectiveness of our slice-based detection method, consider the following three simple code segments:

```
1    for (int i=0; i<10; i++)
2    for (int i=1; i<10; i++)
3    for (int j=2; j<20; j++)
```

For typical text-string-based techniques, such as Duploc [9], all the three segments are considered different. For text-line-based techniques, such as Nicad [31], just segments 1 and 2 are considered similar, since, the identifier name is changed in segment 3. Classical token-based techniques, such as CCFinder [18] and AST-based techniques, such as CloneDR [5], consider the three segments similar, since, the parse trees for these segments are identical and the code differs only in identifier names and literal values.

In our work, the three segments have the same slicing profiles and therefore same slicing vectors, $\langle 2, 1, 0, 0, 0\rangle$. The variable, either i or j, is defined twice inside the *for* loop, one for initialization and one for update, and is used once in the condition. There are no dependent variables, pointers, or called functions in the slicing profiles. Therefore, the three segments will be still similar and will be returned as clones.

Table 4: File-level Slice Profiles (*SPs*) for Figure 2a and Figure 2b, foo_t = foo_timed.

F	Def	Use	Dvars	Ptrs	Cfuncs
1	{1,5,10,11, 17}	{1,2,3,5,6,7,12,13,15, 16,17}	{abc}	{i, abc}	{fun{1}, foo{1, 2}}
2	{1,5,10,11, 12,13,17,21}	{1,2,3,5,6,7,14,15,18, 19,20,21}	{abc}	{i, abc}	{fun{1}, foo_t{1, 2}}

3.5.3 ***Type-3 and Type-4 Detection***. Consider the similar code snippets in Figure 2a and Figure 2b. Both perform the same overall computation, but Figure 2b contains extra statements (highlighted) and extra variables (*start* and *finish*) to time the loop. Both are implemented by different syntactic variants. Textual-based clone detection techniques are unable to detect these non-contiguous and interleaved clones [11].

Using our slice-based detection, Table 4 shows the slice profiles we generated for the code snippets in Figure 2a and Figure 2b at the file level using Eq. 3. We omit the slice profiles at both the variable and the function levels for brevity sake. The only difference is in the *Def* and *Use* fields. This is reasonable since the only change between the two codes is defining two variables, then using them.

These changes are more obvious to see in the generated slicing vectors. Table 5 shows the slicing vectors we generated for the variable, function, and file levels using Eqs. 4, 5, and 6, respectively. At the variable level, all vectors are identical except for the new variables, *start* and *finish*. There are no corresponding variables in the original file (Figure 2a). Since those two variables are newly added variables inside the function *main*, the vectors at the *main* function are the only affected vectors with this addition. Also the number of variables is impacted at the end of the function's vector, now having a value of 4 instead of 2. This change at the function level also impacts the vectors at the file level, and again the *Def* and *Use* fields in addition to the number of variables. However, the number of functions inside the file-level vectors remains the same. Therefore, we now comprehend functions *foo* and *foo_timed* are identical, as are the function *fun* in both versions. However, the similarity between the *main* function in both files is likely too different to be identified as a semantic clone using Eq. 5. Therefore, we are able to detect that those two files as similar by comparing the slicing vectors at the variable level. This is because the slicing vectors for *start* and *finish* are deleted early using Algorithm 2.

3.6 Implementation

Our implementation of srcClone consists of a number of primary components: srcML, srcSlice, slicing vectors, and LSH clustering. We present srcClone's pipeline in Figure 3. It begins with the source code in the form of git repository. Then, we run srcML to produce an XML representation of the source code and abstract syntactic information from the AST. We execute srcSlice using the XML as input and generate the list of slicing profiles for each variable in the system. We parse this data to generate the slicing vectors we need for the clone detection process. We then cluster all similar vectors using the LSH and its near neighbor algorithm.

```
1   int fun(int z){
2       z++;
3       return z;
4   }
5   void foo(int &x, int *y){
6       fun(x);
7       (*y)++;
8   }
9   int main(){
10      int abc = 0;
11      int i = 1;
12      while (i<=10){
13          foo(abc, &i);
14      }
15      std::cout<<"i:"<<i<<"abc:"<<abc;
16      std::cout<<fun(i);
17      abc = abc + i;
18      return 0; }
```

(a) Program A

```
1   int fun(int z){
2       z++;
3       return z;
4   }
5   void foo_timed(int &w, int *y){
6       fun(w);
7       (*y)++;
8   }
9   int main(){
10      int abc = 0;
11      int i = 1;
12      long start = get_time_millis();
13      long finish;
14      while (i<=10){
15          foo_timed(abc, &i);
16      }
17      finish = get_time_millis();
18      std::cout<<"loop took"<<finish-start;
19      std::cout<<"i:"<<i<<"abc:"<<abc;
20      std::cout<<fun(i);
21      abc = abc + i;
22      return 0; }
```

(b) Program B

Figure 2: Motivation Example as Proposed by Gabel et al. [11].

Table 5: Slicing Vectors (*svs*) for Figure 2a and Figure 2b, foo_t = foo_timed.

F	Func	Var	Slicing Vectors		
			Var-level	Func-level	File-level
1	foo	y	<1,1,0,1,0>	<1,5,0,2,1,2>	<**5,11**,1,2,3,**5**,3>
		x	<1,4,0,1,1>		
	fun	z	<1,2,0,0,0>	<1,2,0,0,0,1>	
	main	i	<1,10,1,0,2>	<**3,11**,1,0,3,**2**>	
		abc	<2,7,0,0,2>		
2	**foo_t**	y	<1,1,0,1,0>	<1,5,0,2,1,2>	<**8,12**,1,2,3,**7**,3>
		w	<1,4,0,1,1>		
	fun	z	<1,2,0,0,0>	<1,2,0,0,0,1>	
	main	i	<1,10,1,0,2>	<**6,12**,1,0,3,**4**>	
		abc	<2,7,0,0,2>		
		start	<**1,1,0,0,0**>		
		finish	<**2,1,0,0,0**>		

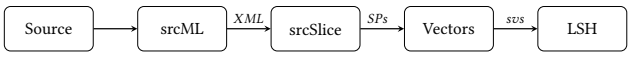

Figure 3: The srcClone Pipeline.

One of srcClone's strengths is that it can detect clones for non-compilable and incomplete source code. However, since srcClone uses srcML, if a repository contains files of an unsupported programming language, we ignore those files. Because the current version of srcSlice supports C/C++ only, so does srcClone.

4 EVALUATION

Evaluating clone detection tools is still problematic for researchers [6, 37]. We perform a preliminary evaluation of our approach and intend to conduct a more rigorous evaluation as future work. To evaluate and validate our approach and our srcClone tool initially we conducted a comparative study with two PDG-based approaches that use program slicing to detect clones as they are the most similar to our approach: (1) Komondoor and Horwitz [19], and (2) Gabel's [11]. Both tools used Grammatech's CodeSurfer [1] tool to build the PDGs and run slicing over C/C++ source code. There are some other approaches that are PDG-based, such as the Duplix [22] and GPLAG tools [25]. However, they are not using program slicing.

Additionally, we use established benchmark scenarios to perform qualitative analysis proposed by Roy et al. [33]. In their paper, they compared and evaluated almost all well-known existing clone detection techniques that include classical and state-of-the-art techniques. They designed 16 different hypothetical editing scenarios that represent the typical changes to copy/pasted code. These scenarios are categorized under the 4 major types of clones, as shown in Section 4.3.

The objectives of our evaluation is twofold, we want to determine if the clones produced from srcClone are comparable to those produced by others in terms of the correctness and the size of the clones. The second objective is to demonstrate that our approach is highly scalable and efficient. These objectives lead to the following two primary research questions:

- RQ1: Does srcClone identify accurate clones?
- RQ2: Is srcClone scalable and efficient?

We performed our evaluations on a macOS with 4 GHz Intel Core i7 and 8 GB DDR3. We used srcML v 0.9.5 and srcSlice Beta-1.0. In our work, the number of dimensions for $SP(v)$ is fixed and is equal to *five*. Therefore, $p_1 = 1 - \frac{r_1}{5}$. We set srcClone's threshold σ and r value to 1. This will give a p_1 values equal to 0.80, 0.83, and 0.86 at the variable, function, and file levels, respectively. To compare with Gabel's and Komondoor's we compare the *cr*-ANN set of srcClone with the subgraphs sets they returned as candidates clones.

4.1 Comparative Results - Komondoor's

We start comparing srcClone with Komondoor's using the largest system they used to run their experiments, the GNU *bison* Unix utility system. While they do not state exactly what version they used, based on the date of their publication we decided to use version 1.29. Table 6 lists the approximate system sizes measured by LOC, number of files, PDGs for Komondoor's, and *SPs* for srcClone.

[1] http://www.grammatech.com

Table 6: System Sizes. Whitespace and Comments are Counted. LOC = Lines of Code, PDG = Program Dependence Graph, and SP = Slice Profile.

System	LOC	Files	PDGs	SPs
bison 1.29	24,8 K	89	28,548	1939
Linux 2.6.16	6,6 M	15,762	–	656,081

Table 7: CG: Clone Groups. For Komondoor and Gabel this is the Number of PDG Nodes. For srcClone this is the Number of Slicing Vectors (*svs*).

	Komondoor		**srcClone**	
System	**Exec. T**	**CG**	**Exec. T**	**CG**
bison 1.29	1h 34m 5s	800	1m 41s	1125
	Gabel		**srcClone**	
Linux 2.6.16	9h 48m 3s	255,108	3m 8s	316,221

Table 7 shows the results of both tools. In 1 hour and 34 minutes, Komondoor's tool found 800 clone groups out of 28.5K PDGs. These groups range in size from 5 to 227 PDGs. In less than two minutes, srcClone found 1125 clone groups at the three granularity levels.

We manually verify precision of the returned clones by our tool and compared it to those returned by Komondoor's. We show some examples of interesting clones that are non-contiguous, reordered, or intertwined in Figure 4. This code snippet is cloned five times inside the *bison* system. Komondoor was able to detect only two copies of it. Specifically, the highlighted code in Figure 4a and Figure 4c. Our tool was able to detect all the five copies. We show the slicing vectors of the functions that include these snippets in Figure 4f. The *svs* at the variable level are identical for common variables between the code snippets. For example, the *svs* for variable *fp3* as shown in the last column all are similar. Komondoor was not able to detect the line number 1 in Figures 4a and 4c. This is due to the heuristic they used during the slicing process. When slicing backward from the nodes that are inside the loops (*while* and *for*), they add to the slice the nodes that are outside the loops only if the loop predicates match. Since both predicates do not match, the initial assignments to pointer $fp3$ are not included in the clones.

4.2 Comparative Results - Gabel's

We choose the largest system they used in their evaluation, which is the Linux kernel. They did not mention the version number they used, however, since they compared their work with Deckard's, and Deckard used Linux kernel version 2.6.16, so do we. We show the system sizes in Table 6. Table 7 illustrates a summary of our results. In 9 hours and 48 minutes, Gabel's tool found 255,108 clone groups. These groups range in size from 4 to 32 PDGs. The total number of detected clone lines ranges from 30,367 to 940,497. srcClone, within 3 minutes, was able to find 316,221 similar slicing vectors. Specifically, 254,103 on the variable level, 37,117 at the function level, and 25,001 at the file level. Its total number of detected clone lines ranges from 275,576 to 2,794,133.

4.3 Qualitative Analysis - Roy's Scenarios

For this component of our evaluation we employ the scenarios from Roy et al. [33]. These present an established predictive benchmark for the recall and precision of clone detection techniques. Some scenarios are difficult and represent obstacles for the clone detection techniques. We evaluated srcClone against all four proposed scenarios, which includes 16 sub-scenarios. We present our slicing vectors for the original code segment in Figure 5 and the results for Scenarios 1, 2, 3, and 4 as shown in Figures 6, 7, 8, and 9, respectively.

We compared srcClone with all clone detection techniques used by Roy et al. In total, we compared 42 techniques based on their ability and feasibility of detecting the four proposed scenarios. We show an abridged version of our results in Table 8 comparing our approach to the top coverage approaches in each category of clone detector in addition to the two we compared to earlier. Scenario 4 is the most difficult to detect, with most of the techniques failing. For our approach, we were successful at detecting, and all 16 (sub)scenarios using the function granularity. As shown in Table 8, the coverage column represents the recall and the pies are representing the precision or false positive for each technique.

5 RELATED WORK

Many clone detection techniques exist in the literature. These techniques can be categorized into text-based [16, 17, 27], token-based [4, 18, 24], tree-based [5, 21], metrics-based [20, 26, 34], and graph-based [11, 19, 22, 25]. We talk about the most related works herein.

PDG-based clone detection algorithms detect clones as isomorphic sub-graphs in a given PDG. Since this problem is NP hard, related algorithms use approximative PDG solutions and are quite expensive. To find isomorphic sub-graphs, some approaches used backward program slicing [40]. However, using a backward slicing results in a tradeoff between *precision* (false positive) and *recall* (undiscovered clones). There are similar sub-graphs that cannot be detected by only using backward slicing [11, 19]. Komondoor and Horwitz [19] use backward program slicing to find isomorphic PDG's sub-graphs. They find clones by starting with two matching nodes in the PDG and slicing backwards from those nodes at the same time, then comparing the resulted sub-graphs. Hamid and Zaytsev [13] conducted a replication of this work to find refactorable semantic clones based on PDG and backward program slicing. Similarly, Gabel et al. [11] uses forward program slicing [7] to find semantic PDG sub-graphs, maps these sub-graphs to related structured syntax trees, and finds clones using the Deckard clone detection approach [15]. They used intra-procedural forward slicing to increase the recall by finding different flows of data throughout a given function.

In our past work, we envisioned employing slicing to detect clones but in a different way [3]. We proposed using forward slices and an MD5 hashing algorithm to detect code clones between different versions of software systems. Our proposal was to encode slices to strings and then feed those strings to an MD5 hash algorithm that produces a 128-bit hash value. The work in this paper builds on that idea, however, it is different as we now use embedded slicing information and the slicing fields to generate slicing vectors that represent each slice uniquely.

Deckard implements a tree similarity approach that uses the idea of *characteristic vectors*. Each vector is a numerical approximation of a particular subtree. In contrast, we introduce *slicing vectors*

```
1 fp3 = lookaheadset + tokensetsize;
2
3   for (i=lookaheads[state]; i < k; i++)
4     {
5       fp1 = LA + i * tokensetsize;
6       fp2 = lookaheadset;
7       while (fp2 < fp3)
8     *fp2++ |= *fp1++;
9     }
```

(a) conflicts.c/count_sr_conflicts

```
1   fp3 = rowp + tokensetsize;
2   for (sp=lookback[i]; sp; sp = sp->next)
3 {
4   fp1 = rowp;
5   fp2 = F + tokensetsize * sp->value;
6   while (fp1 < fp3)
7     *fp1++ |= *fp2++;
8 }
9   rowp = fp3;
```

(b) lalr.c/compute_lookaheads

```
1 fp3 = base + tokensetsize;
2 ...
3 if (rp) {
4   while ((j = *rp++) >= 0) {
5   ...
6   fp1 = base;
7   fp2 = F + j * tokensetsize;
8   while (fp1 < fp3)
9     *fp1++ |= *fp2++; }}
```

(c) lalr.c/traverse

```
1 ...
2     fp2 = fp1;
3     fp3 = lookaheadset;
4
5     while (fp3 < fp4)
6 *fp3++ |= *fp2++;
```

(d) conflicts.c/set_conflicts

```
1     ...
2     fp2 = shiftset;
3     fp3 = lookaheadset;
4
5     while (fp3 < fp4)
6       *fp2++ |= *fp3++;
```

(e) conflicts.c/print_reductions

Func	$sv(m)$	*LoC*	$sv(fp3)$
(a) count_sr_conflicts	⟨21, 27, 1, 4, 1, 8⟩	67	⟨2, 2, 0, 1, 0⟩
(b) compute_lookaheads	⟨18, 16, 1, 4, 1, 8⟩	40	⟨2, 2, 0, 1, 0⟩
(c) traverse	⟨18, 21, 1, 4, 1, 8⟩	54	⟨2, 2, 0, 1, 0⟩
(d) set_conflicts	⟨21, 23, 1, 2, 2, 9⟩	87	⟨2, 2, 0, 1, 0⟩
(e) print_reductions	⟨63, 75, 8, 4, 4, 20⟩	227	⟨2, 2, 0, 1, 0⟩

(f) Slicing Vectors for 5 Functions and 1 Common Variable

Figure 4: GNU `bison-1.29/src/` Unix Utility System.

Table 8: Abridged Table for Clone Detection Scenario-Based Evaluation [33] Appended with srcClone.

● *very well* ◐ *well* ◓ *medium* ◔ *low* ◔ *probably can* ○ *probably cannot* ○ *cannot*

	Citation	Scenario 1			Scenario 2				Scenario 3					Scenario 4				Coverage
		a	b	c	a	b	c	d	a	b	c	d	e	a	b	c	d	%
Text-Based	Nasehi [27]	◓	◓	◓	◓	◓	◓	◓	◓	◓	◔	◔	◔	○	○	○	◓	81
Token-Based	CP-Miner [24]	◐	◐	◐	◐	◐	◐	◔	◐	◐	◓	◓	◓	◔	◔	◔	○	75
Tree-Based	CloneDr [5]	●	●	●	●	●	●	◔	◓	◓	◔	◓	◓	◔	○	○	○	75
Metrics-Based	Kontogiannis [20]	◐	◐	◐	◐	◐	◐	◔	◔	◔	◔	◔	◔	◓	◓	◓	◔	100
Graph-Based	GPLAG [25] *	◐	◐	◐	◐	◐	◐	◔	◔	◔	◓	◓	◓	◐	◐	◔	◓	100
	Komondoor [19]	◐	◐	◐	◐	◐	◐	◔	◔	◔	◔	◔	◔	◐	◓	◔	○	75
	Gabel [11]	◐	◐	◐	◐	◐	◐	◐	◐	◐	◓	◓	◓	◓	◓	◓	○	94
Slice-Based	srcClone	●	●	●	●	●	●	●	●	●	◐	◐	●	●	●	●	◐	100

* A technique/tool with special limitations or other main purpose than clone detection, such as whole file comparison, visualization only, plagiarism detection, IDE support or other special issues.

to capture structural information of slices. Each vector has a fixed size that is based on the number of slicing fields we calculate using srcClone. It is fully acceptable to use more or fewer fields here as long as the generated vector is unique and detects all the structural information of a given slice.

Gallagher and Lucas [12] tried to use a backward decompositional slicing using PDGs to answer the question *"Are decomposition slices clones?"*. However, they did not reach any conclusions.

Krinke [22] developed another PDG-based clone detection approach that does not suffer from the tradeoff between recall and precision. They used fine-grained PDGs to detect code fragments as a clone candidates. As it is a PDG approach, it is quite expensive. For example, The *bison* system from our evaluation takes Krinke approximately 1 hour to find similar PDGs. We chose not to compare against Krinke in this evaluation because they do not use program slicing.

Xue et al. [41] presents Clone-Slicer to detect domain-specific clones on binaries rather than the source code level. They used forward slicing to remove pointer-irrelevant instructions. This differs from our work in that it uses binary executables instead of the source code, and they find pointer-related clones only.

The GPLAG [25] tool is a graph-based, however, it is designed for a different purpose than clone detection and has some corresponding limitations [33].

6 FUTURE WORK AND CONCLUSIONS

We have presented our approach and initial results for detecting code clones using program slices. We believe this is a new perspective on semantic clone detection, allowing for large system cloning analysis and comprehension using semantic clones. We employ a scalable lightweight slicing tool to compute the necessary slicing data. While this tool is inter-procedural and highly scalable, its tradeoff is that it may not match the accuracy of generating a

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod =1.0;
4 for (int i=1;i<=n;i++)
5     {sum=sum + i;
6     prod = prod * i;
7     foo(sum, prod); }}
```

(a) Original Copy

Var	*sv*
i	⟨2, 3, 2, 0, 0⟩
prod	⟨2, 1, 0, 0, 1⟩
sum	⟨2, 1, 0, 0, 1⟩
n	⟨1, 1, 0, 0, 0⟩
sumProd	**⟨6, 4, 2, 0, 1, 4⟩**

(b) Slicing Vectors

Figure 5: Original Copy by Roy et al. [33] and its Vectors (*sv*).

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod =1.0;
4  for (int i=1;i<=n;i++)
5       {sum=sum + i;
6        prod = prod * i;
7        foo(sum, prod); }}
```

(a) S1, whitespace changes

```
1 void sumProd(int n) {
2 float sum=0.0; //C1'
3 float prod =1.0; //C
4 for (int i=1;i<=n;i++)
5     {sum=sum + i;
6     prod = prod * i;
7     foo(sum, prod); }}
```

(b) S1, commenting changes

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod =1.0;
4 for (int i=1;i<=n;i++){
5     sum=sum + i;
6     prod = prod * i;
7     foo(sum, prod); }}
```

(c) S1, formatting changes

Func	*Sv*	*R*
S1(a)	⟨6, 4, 2, 0, 1, 4⟩	●
S1(b)	⟨6, 4, 2, 0, 1, 4⟩	●
S1(c)	⟨6, 4, 2, 0, 1, 4⟩	●

(d) Slicing Vectors, S1(a)-S1(c)

Figure 6: Scenario-1, Type-1 Clone.

```
1 void sumProd(int n){
2 float s=0.0; //C1
3 float p =1.0;
4 for (int j=1;j<=n;j++)
5     {s=s + j;
6     p = p * j;
7     foo(s, p); }}
```

(a) S2, systematic renaming

```
1 void sumProd(int n){
2 float s=0.0; //C1
3 float p = 1.0;
4 for (int j=1;j<=n;j++)
5     {s=s + j;
6     p = p * j;
7     foo(p, s); }}
```

(b) S2,unsystematic renaming

```
1 void sumProd(int n) {
2 int sum=0; //C1
3 int prod =1;
4 for (int i=1;i<=n;i++)
5     {sum=sum + i;
6     prod = prod * i;
7     foo(sum, prod); }}
```

(c) S2, changing types and literals, not consistent

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod =1.0;
4 for (int i=1;i<=n;i++)
5     {sum=sum + (i*i);
6     prod = prod*(i*i);
7     foo(sum, prod); }}
```

(d) S2, replacing parameters with expressions

Func	*Sv*	*R*
S2(a)	⟨6, 4, 2, 0, 1, 4⟩	●
S2(b)	⟨6, 4, 2, 0, 1, 4⟩	●
S2(c)	⟨6, 4, 2, 0, 1, 4⟩	●
S2(d)	⟨6, 4, 2, 0, 1, 4⟩	●

(e) Slicing Vectors, S2(a)-S2(d)

Figure 7: Scenario-2, Type-2 Clone.

complete PDG/SDG. However, srcSlice produces reliable accuracy given its speed and lightweight approach.

We encode slicing profiles as vectors and hash them efficiently using LSH at three levels of granularity. We have implemented our algorithm in the tool srcClone. We compared srcClone with two PDG-based tools that use program slicing. Our evaluation demonstrates that it is practical and scales to millions of lines of code. This tool is very competitive with other detection tools, and is capable of producing clones that are related semantically for a given variable, function, or file.

In the future, we plan to compare srcClone with non-slicing based and non-PDG based state of the art tools such as: SourcererCC [35], Oreo [34], and NiCad [31] on standard benchmarks, such as BigCloneBench [38]. We plan to continue this line of research and provide slice-based clone information that will help maintainers understand clones across versions. Also, our future

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod = 1.0;
4 for (int i=1;i<=n;i++)
5     {sum=sum + i;
6     prod = prod * i;
7     foo(sum,prod,n); }}
```

(a) S3, small insertion line 7

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod = 1.0;
4 for (int i=1;i<=n;i++)
5     {sum=sum + i;
6     prod = prod * i;
7     foo(prod); }}
```

(b) S3, small deletion line 7

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod = 1.0;
4 for (int i=1;i<=n;i++)
5     {sum=sum + i;
6     prod = prod * i;
7     if (n % 2)==0 {
8     foo(sum, prod);} }}
```

(c) S3, insert new line 7

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod =1.0;
4 for (int i=1;i<=n;i++)
5     {sum=sum + i;
6     //line deleted
7     foo(sum, prod); }}
```

(d) S3, delete line 6

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod =1.0;
4 for (int i=1;i<=n;i++)
5  { if(i%2) sum=sum+i;
6     prod = prod *i;
7     foo(sum, prod); }}
```

(e) S3, changes to whole line 5

Func	*Sv*	*R*
S3(a)	⟨6, 4, 2, 0, 1, 4⟩	●
S3(b)	⟨6, 4, 2, 0, 1, 4⟩	●
S3(c)	⟨6, **5**, 2, 0, 1, 4⟩	◕
S3(d)	⟨**5**, **3**, **1**, 0, 1, 4⟩	◕
S3(e)	⟨6, 4, 2, 0, 1, 4⟩	●

(f) Slicing Vectors, S3(a)-S3(e)

Figure 8: Scenario-3, Type-3 Clone.

```
1 void sumProd(int n) {
2 float prod =1.0;
3 float sum=0.0; //C1
4 for (int i=1;i<=n;i++)
5     {sum=sum + i;
6     prod = prod * i;
7     foo(sum, prod); }}
```

(a) S4, reorders data independent declarations

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod =1.0;
4 for (int i=1;i<=n;i++)
5     {prod = prod * i;
6     sum=sum + i;
7     foo(sum, prod); }}
```

(b) S4, reorders data independent statements

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod = 1.0;
4 for (int i=1;i<=n;i++)
5     {sum=sum + i;
6     foo(sum, prod);
7     prod=prod * i; }}
```

(c) S4, reorders data dependent statements

```
1 void sumProd(int n) {
2 float sum=0.0; //C1
3 float prod = 1.0;
4 int i=0;
5  while (i<=n)
6   { sum=sum + i;
7     prod = prod * i;
8     foo(sum, prod);
9     i++; }}
```

(d) S4, replaces a control statement

Func	*sv(m)*	*Rate*
S4(a)	⟨6, 4, 2, 0, 1, 4⟩	●
S4(b)	⟨6, 4, 2, 0, 1, 4⟩	●
S4(c)	⟨6, 4, 2, 0, 1, 4⟩	●
S4(d)	⟨**7**, 4, 2, 0, 1, 4⟩	◕

(e) Slicing Vectors, S4(a)-S4(d)

Figure 9: Scenario-4, Type-4 Clone.

effort will be devoted to replicate our analyses on closed source systems, which might exhibit different results. Finally, an interesting area of investigation we will pursue is the influence of each field in the slicing vector. We believe srcClone is an important step in improving program comprehension and we plan on conducting further experiments with developers and refinements in the future.

REFERENCES

[1] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.

[2] Hakam W Alomari, Michael L Collard, Jonathan I Maletic, Nouh Alhindawi, and Omar Meqdadi. 2014. srcSlice: very efficient and scalable forward static slicing. *Journal of Software: Evolution and Process* 26, 11 (2014), 931–961.

[3] Hakam W Alomari and Matthew Stephan. 2018. Towards slice-based semantic clone detection. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. IEEE, Campobasso, Italy, 58–59.

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

[5] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance*. IEEE, Bethesda, MD, USA, 368–377.

[6] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering* 33, 9 (2007), 577–591.

[7] Jean-Francois Bergeretti and Bernard A Carré. 1985. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 1 (1985), 37–61.

[8] Michael L Collard, Michael J Decker, and Jonathan I Maletic. 2011. Lightweight transformation and fact extraction with the srcML toolkit. In *2011 IEEE 11th international working conference on source code analysis and manipulation*. IEEE, Williamsburg, VI, USA, 173–184.

[9] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *International Conference on Software Maintenance*. IEEE, Oxford, England, UK, 109–118.

[10] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[11] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*. ACM, Leipzig, Germany, 321–330.

[12] Keith Gallagher and Lucas Layman. 2003. Are decomposition slices clones?. In *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE, Portland, OR, USA, 251–256.

[13] Ammar Hamid, Vadim Zaytsev, et al. 2014. Detecting Refactorable Clones by Slicing Program Dependence Graphs.. In *SATToSE*. SATToSE 2014, Italy, 37–48.

[14] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, Texas, Dallas, USA, 604–613.

[15] 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, USA, 96–105.

[16] J Howard Johnson. 1993. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*. IBM Press, 171–183.

[17] J Howard Johnson. 1994. Substring Matching for Clone Detection and Change Tracking.. In *ICSM*, Vol. 94. Victoria, BC, Canada, 120–126.

[18] 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.

[19] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *International static analysis symposium*. Springer, Berlin, Heidelberg, 40–56.

[20] Kostas A Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. 1996. Pattern matching for clone and concept detection. *Automated Software Engineering* 3, 1-2 (1996), 77–108.

[21] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*. IEEE, Benevento, Italy, 253–262.

[22] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*. IEEE, Stuttgart, Germany, 301–309.

[23] António Menezes Leitão. 2004. Detection of redundant code using R 2 D 2. *software quality journal* 12, 4 (2004), 361–382.

[24] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering* 32, 3 (2006), 176–192.

[25] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, PA, Philadelphia, USA, 872–881.

[26] Jean Mayrand, Claude Leblanc, and Ettore Merlo. 1996. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics.. In *icsm*, Vol. 96. IEEE, Monterey, CA, USA, 244.

[27] Seyyed Mehdi Nasehi, Gholam Reza Sotudeh, and Maziar Gomrokchi. 2007. Source code enhancement using reduction of duplicated code. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*. ACTA Press, USA, 192–197.

[28] Christian D Newman, Tessandra Sage, Michael L Collard, Hakam W Alomari, and Jonathan I Maletic. 2016. srcSlice: a tool for efficient static forward slicing. In *International Conference on Software Engineering Companion*. IEEE, Austin, TX, USA, 621–624.

[29] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[30] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[31] Chanchal K Roy and James R Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th iEEE international conference on program comprehension*. IEEE, Amsterdam, Netherlands, 172–181.

[32] Chanchal K Roy and James R Cordy. 2008. Scenario-based comparison of clone detection techniques. In *2008 16th IEEE International Conference on Program Comprehension*. IEEE, Amsterdam, Netherlands, 153–162.

[33] 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.

[34] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 2018. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Lake Buena Vista, FL, USA, 354–365.

[35] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, Austin, TX, USA, 1157–1168.

[36] Josep Silva. 2012. A vocabulary of program slicing-based techniques. *ACM computing surveys (CSUR)* 44, 3 (2012), 12.

[37] Jeffrey Svajlenko and Chanchal K Roy. 2014. Evaluating modern clone detection tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, Victoria, BC, Canada, 321–330.

[38] Jeffrey Svajlenko and Chanchal K Roy. 2015. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, IEEE, Bremen, Germany, 131–140.

[39] Frank Tip. 1994. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam.

[40] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[41] Hongfa Xue, Guru Venkataramani, and Tian Lan. 2018. Clone-slicer: Detecting domain specific binary code clones through program slicing. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*. ACM, Toronto, Canada, 27–33.