The Journal of Systems & Software 184 (2022) 111115

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Clone detection through srcClone: A program slicing based approach ☆

Hakam W. Alomari *, Matthew Stephan

Department of Computer Science and Software Engineering, Miami University, Oxford, OH, USA

ARTICLE INFO

Article history:
Received 3 August 2020
Received in revised form 5 September 2021
Accepted 21 September 2021
Available online 14 October 2021

Keywords:
Code clone
Clone detection
Program slicing
Semantic clones

ABSTRACT

Software clone detection is an often used approach to understand and maintain software systems. One category of clones that is challenging to detect but very useful is semantic clones, which are similar in semantics but differ in syntax significantly. Semantic clone detectors have trouble scaling to larger systems and sometimes struggle with recall and precision. To address this, we developed a slice-based scalable approach that detects both syntactic and semantic code clones, srcClone. srcClone ascertains code segment similarity by assessing the similarity of their corresponding program slices. We employ a lightweight, publicly-available, scalable program slicer within our clone detection approach. Using dependency analysis to detect and assess cloned components, we discover insights about code components that can be affected by a clone pair or set. These elements are critical in impact analysis. It 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 first evaluate srcClone by comparing it to six state-of-the-art tools and two additional semantic clone detectors in performance, recall, and precision. We use the BigCloneBench real clones benchmark to facilitate comparison. We use an additional 16 established benchmark scenarios to perform a qualitative comparison between srcClone and 44 clone detection approaches in their capabilities to detect these scenarios. To further measure scalability, we evaluate srcClone on 191 versions of Linux kernel, containing approximately 87 MLOC. In our evaluations, we illustrate our approach is both relatively scalable and accurate. While its precision is slightly less than some other techniques, it makes up for it in higher recall including semantic clones unable to be found by any existing techniques. We contend our approach is an important advancement in software cloning that it is both demonstrably scalable and can detect more types of clones than existing work, thus providing developers greater information into their software.
© 2021 Elsevier Inc. All rights reserved.

1. Introduction

As software systems grow increasingly large, so too do the challenges involved in facilitating their comprehension and maintenance. Clone detection is one technique that can help address such challenges by finding repeated and similar software code within and across projects. A software clone is a software component, such as code or a model, that is either identical or similar to another component. Code clones, for example, can be classified by detectors as exact, renamed, near-miss, and semantic clones (Baker, 1995; Roy and Cordy, 2007). Semantic clones are syntactically dissimilar but implement the same functionality. In software, clones may exist due to a variety of reasons including, (1) copy–paste operations, (2) naive re-coding, or (3) design decisions about abstract data types that necessitate duplication (Baxter et al., 1998; Ducasse et al., 1999; Sajnani et al., 2016). While some presume software clones to be a negative indicator of software quality, that is not always the case (Kapser and Godfrey, 2008).

In large software systems, code clones' existence presents an economic burden to software maintenance and re-engineering tasks. A fault in a cloned unit of code can cause a ripple effect, consequently amplifying the debugging effort. Any change in one unit of code needs to be propagated to all other cloned units by programmers. This problem is representative of tasks that degrade maintainability of the system because of code clones. With the large amount of source code and omnipresence of modern software, large-scale clone detectors have become a necessity.

Many approaches for detecting clones have been proposed over the years. These approaches are present in surveys of software cloning literature (Roy and Cordy, 2007; Bellon et al., 2007; Rattan et al., 2013; Sheneamer and Kalita, 2016; Svajlenko and Roy, 2014). The clones detected by each of these approaches are limited fundamentally by each of the underlying definition of clone considered by the approaches, the similarity measures, and the analysis level applied to the source code by the clone detector. While there exist many approaches for clone detection,

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail addresses: alomarhw@miamioh.edu (H.W. Alomari), stephamd@miamioh.edu (M. Stephan).

https://doi.org/10.1016/j.jss.2021.111115
0164-1212/© 2021 Elsevier Inc. All rights reserved.

```
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

Fig. 1. Example motivation as proposed by Roy et al. (2009) and Roy and Cordy (2008c).

scalability and accuracy remain open research areas, particularly for detecting near-miss and semantic clones (Roy et al., 2009; Roy and Cordy, 2008c).

There are generally two main kinds of similarity measures of clones: *textual* and *functional* (Roy et al., 2009). Textual-based techniques result in clones that are Type-1, identical; Type-2, renamed; and Type-3, near-miss. Textual-based techniques often find clones within a program's contiguous and structured syntax. While there are workarounds to detect clones that are non-contiguous, reordered, and intertwined using textual-based techniques (Ragkhitwetsagul et al., 2018), generally, they are not designed nor well-suited to find semantic clones (Komondoor and Horwitz, 2001; Leitão, 2004; Gabel et al., 2008). Functional-based techniques use semantic analysis, resulting in Type-4 clones. This is very challenging as ascertaining components that have similar semantics can be imprecise and is generally undecidable (Jiang et al., 2007; Gabel et al., 2008). Some semantic clone detection techniques employ program semantic abstractions such as Control Flow Graphs (CFGs) (Allen, 1970) or Program Dependence Graphs (PDGs) (Ferrante et al., 1987). Clone detection in these techniques are instances of the NP-hard isomorphic sub-graph identification problem. Semantic clone detectors thus rely on solutions that are more approximative (Komondoor and Horwitz, 2001; Krinke, 2001; Gabel et al., 2008). However, these solutions still quite expensive.

Program slicing is one common and vetted method to help understand and assess software's semantics (Alomari et al., 2014b; Komondoor and Horwitz, 2001; Alomari and Stephan, 2018; Gabel et al., 2008; Xue et al., 2018b). While program slicing is generally demanding in both time and space, there are some specific program slicers that are lightweight, highly scalable, and publicly available (Alomari et al., 2014b).

As a motivating example, we present code considered by Roy et al. in their clone detection research (Roy et al., 2009; Roy and Cordy, 2008c). We chose this example intentionally as it provides an established baseline of comparison and contrast. The code in Fig. 1(a), is similar to that in Fig. 1(b); both have the same functionality, but the second figure has a different control statement. This scenario is just one example of hard-to-detect semantic clones that text-based and function-based techniques often struggle, as we showcase and discuss in this paper.

1.1. Contributions and paper organization

This paper presents our design process for, application of, and results in using our clone detection approach and corresponding tool, SRCCLONE, for large scale C/C++, Java, and C# software systems. SRCCLONE uses program slicing to find software clones. SRCCLONE obtains its slicing results via a scalable slicer, called SRCSLICE (Alomari et al., 2014b; Newman et al., 2016). We detect similar components using program slicing. Specifically, SRCCLONE calculates slice-based metrics using slicing information. These metrics help us compute certain slicing vectors to approximate the structural information inside slices, which we then use in a customized Locality Sensitive Hashing (LSH) algorithm (Datar et al., 2004) to efficiently hash and cluster similar vectors. SRCCLONE considers two code fragments semantically similar if their corresponding slices are also similar. These similar slices represent a candidate clone class.

This paper describes our extensions to our previous work on detecting source code clones using program slicing. Our extensions yield a much more complete and validated research product. We explicate that previous work and our new contributions herein. Our previous work in this direction includes,

1. Our initial position paper proposed the general idea and described our first foray of employing slicing to detect clones (Alomari and Stephan, 2018). This included using forward slices and an MD5 hashing algorithm (Rivest and Dusse, 1992) to detect code clones between different versions of software systems. Our proposal was to encode slices to strings and then supply those strings to an MD5 hash algorithm that produces a 128-bit hashed values.
2. We next researched, devised, and prototyped our current approach, SRCCLONE. We first introduced it in a conference paper (Alomari and Stephan, 2020), where we conducted a comparison study to two specific semantic clone detectors. We also performed recall and precision analysis using established benchmark scenarios.

This article makes the following novel contributions,

1. A redesigned algorithm that greatly enhances slicing vectors generation issues and LSH hashing. This includes new slice-based cognitive complexity metrics that use the slicing information. These metrics are used to generate the slicing vectors.
2. The additional novel capability of SRCCLONE to retrieve not only the cloned code but also the code that impacted by clones. Dependencies that exist within slices now help us to retrieve potentially impacted elements. Clones that can be found using our tool are all relevant and meaningful computations that involve a given variable(s).
3. We have completed a deeper and extended evaluation that includes additional experiments and results demonstrating the efficiency, accuracy, and scalability of SRCCLONE. Specifically, we in this article compare SRCCLONE with four non-slicing based and non-PDG based state-of-the-art clone detectors: DECKARD, NICAD, SOURCERERCC, and OREO, in addition to two code clone search engines: FACOY and SIAMESE. We evaluate the precision and recall of our approach against others using an established code clone benchmark with real-world projects, BigCloneBench (Svajlenko and Roy, 2015, 2016).
4. To further demonstrate the scalability, we have applied SRCCLONE to 10 years of versions of the Linux kernel. In this quantitative analysis, we have investigated the range of clone sizes in which most of the clones fall and try to answer the question of what the size is of a typical clone for

this system. We further investigated the linear correlation coefficient, and therefore the coefficient of determination, between various granularity levels of clones as a way to analyze the relation between them.

We organize the remainder of this article 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 contrasts related work to position the originality of our work and discusses the limitations of our study. Finally, Section 6 concludes the paper.

2. Background - program slicing and srcSlice

Given the context of this special-issue article, we assume readers have a fundamental understanding of code clone detection and we refer them to the many surveys describing the area for further reading and reinforcement (Roy and Cordy, 2007; Bellon et al., 2007; Rattan et al., 2013; Sheneamer and Kalita, 2016; Svajlenko and Roy, 2014; Ragkhitwetsagul et al., 2018). This section describes program slicing and srcSlice. We discuss only what srcSlice does and how we can use it to find clones. Complete details of the slicing algorithm and how srcSlice computing final slices are in past work by Alomari et al. (2014b, 2012).

Weiser was the originator of program slicing (Weiser, 1984). Their "slice" is executable statements that should preserve the behavior of the original software. This algorithm traces data and control dependencies for identifying the direct and indirect pertinent variables and statements. Existing slicing techniques differ broadly according to the type of slices they can compute. These techniques are covered in various surveys (Tip, 1994; Silva, 2012; Binkley and Harman, 2004). Generally, existing techniques do not scale very well to large-scale software systems since most of them use PDGs to compute slices, which are a costly process in the large context.

In this paper, we leverage srcSlice (Newman et al., 2016), which addresses scalability limitations by eliminating the effort and time needed to build entire PDGs. srcSlice integrates a textual-based approach with a lightweight static analysis infrastructure, srcML (Collard et al., 2011) (see srcML.org), which is an XML representation of source code. This format directly supports static analysis by providing direct access to abstract syntactic information within the source code. We use this syntactic information to identify program dependencies as needed (on-the-fly) when identifying the slice. srcML includes a toolkit that supports conversion between source code and its format. Several languages, including C, C++, C#, and Java are supported. This format has been used for lightweight code transformation (Collard et al., 2010), fact extraction (Collard et al., 2011), and pattern matching of complex code (Dragan et al., 2006).

srcSlice is a static forward decompositional and interprocedural slicing technique. srcSlice first converts a system into srcML format. srcSlice then gathers data about all files, functions, and variables in the system, and stores it all in a dictionary. This dictionary is three-tiered and includes three maps: files to functions, functions to variable names, and variable names to slice profiles. srcSlice does not rely fully on pre-computed data and control dependencies, as other slicing techniques. Instead, it computes selectively what it needs while it analyzes the code and keeps tracking just enough context to determine the dependencies. Therefore, srcSlice is very fast and efficient. On the Linux kernel version 4.06 with ≈13 MLOC, srcSlice computes around 2 million slices within 7 minutes (Newman et al., 2016). srcSlice has been used in a variety of contexts including maintenance effort estimation (Alomari et al., 2014a), extracting of cognitive complexity metrics (Alqadi, 2019; Alqadi and Maletic, 2020), def-use extraction between variable definitions and transform calls (Borrelli et al., 2020), and slicing visualization (Alomari et al., 2016).

For each variable in the system, srcSlice computes a slice profile, line by line, as it encounters identifiers. Each slice profile includes all the data gathered about an identifier during slicing. Because srcSlice uses a forward decompositional slicing definition, these data include all source code statements that are affected transitively by the value of the identifier along with data and control dependencies. After computing all slice profiles, a single pass through all the profiles is completed to take into account dependent variables, function calls, and direct pointer aliasing to generate the final slices. This pass is necessary because a variable's slice has impact beyond just its local scope. srcSlice uses the function calls and pointer aliases to calculate indirect and inter-procedural slicing information. For example, when a variable is used as an argument to a call, srcSlice computes how the value of the called variable is used within the call. Accordingly, srcSlice updates the slice profile with data about uses within the called function. Complete details of how the final slices are computed are given in srcSlice literature (Alomari et al., 2014b; Newman et al., 2016; Alomari et al., 2012).

Now, we present how we use srcML and srcSlice in our work to convert the source code into srcML format then slicing it using srcSlice tool. We used the code snippet in Fig. 1(a) as an example input to srcML, which we show in Listing 1. Its associated output can be seen in Listing 2, which we use as input to srcSlice. We show the system dictionary generated by srcSlice in Listing 3. As shown in Listing 2, the srcML format creates a fully queryable document containing the entire source code and preserves comments, white space, and prepossessing statements. The programming language and file name are also recorded, as we demonstrate in Listing 2. Every file within the system receives a similarly structured unit tag. The function tag surrounds all information for a particular function within the defined unit tag for a particular file. This information consists of the return type of the function, the function name, the parameter list, and all associated source code. Comments from the source code are also included. This information is formatted to follow XML syntax to allow for queries to be run against the document (Collard et al., 2013).

```
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); }}
```

Listing 1: Example Input to srcML, Code Snippet in Fig. 1(a).

As shown in Listing 3, the names of the file are repeated for every variable and function they contain and the names of function will be repeated for every variable they contain. This is due to the non-nested nature of the output format. This is evident as *cordy-O.cpp* and *sumProd* are both repeated for every entry in the system dictionary. srcSlice typically tracks the impacted areas by a modification to a variable. For example, name, type, and value changes. Using srcSlice's dictionary, the user needs only to parse it into a specific data structure then check the *def*, *use*, *dvars*, *ptrs*, and *cfuncs* sets for each variable for which they are interested. This indicates exactly which statements in the system will potentially see changes in behavior due to a modification of the variable and which lines in system are similar to that variable.

The slice profiles for code snippets in Figs. 1(a) and 1(b) are shown in Table 1. These are the slice profiles for each variable.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="0.9.5" language="C++"
filename="Users/alomarhw/srcML_srcSlice/cordy/cordy-O.cpp"><function><type><name>void</name></type> <name>sumProd</name
><parameter_list>(<parameter><decl><type><name>int</name></type> <name>n</name></decl></parameter>)</parameter_list> <
block>{
<decl_stmt><decl><type><name>float</name></type> <name>sum</name><init>=<expr><literal type="number">0.0</literal></expr></
init></decl>;</decl_stmt> <comment type="line">//C1</comment>
<decl_stmt><decl><type><name>float</name></type> <name>prod</name> <init>=<expr><literal type="number">1.0</literal></expr><
/init></decl>;</decl_stmt>
<for>for <control>(<init><decl><type><name>int</name></type> <name>i</name><init>=<expr><literal type="number">1</literal></
expr></init></decl>;</init><condition><expr><name>i</name><operator>&lt;=</operator><name>n</name></expr>;</condition><
incr><expr><name>i</name> <operator>=</operator> <name>i</name><operator>+</operator><literal type="number">1</literal>
</expr></incr>)</control>
  <block>{<expr_stmt><expr><name>sum</name><operator>=</operator><name>sum</name> <operator>+</operator> <name>i</name></
  expr>;</expr_stmt>
  <expr_stmt><expr><name>prod</name> <operator>=</operator> <name>prod</name> <operator>*</operator> <name>i</name></expr>;
  </expr_stmt>
  <expr_stmt><expr><call><name>foo</name><argument_list>(<argument><expr><name>sum</name></expr></argument>, <argument><
  expr><name>prod</name></expr></argument>)</argument_list></call></expr>;</expr_stmt>}</block></for>}</block></function>
  </unit>
```

Listing 2: Example Output of srcML. This is the Input of srcSlice.

```
cordy-O.cpp,sumProd,i,def{4},use{4,5,6},dvars{prod,sum},ptrs{},cfuncs{}
cordy-O.cpp,sumProd,prod,def{3,6},use{7},dvars{},ptrs{},cfuncs{foo{2}}
cordy-O.cpp,sumProd,sum,def{2,5},use{7},dvars{},ptrs{},cfuncs{foo{1}}
cordy-O.cpp,sumProd,n,def{1},use{4},dvars{},ptrs{},cfuncs{}
```

Listing 3: Example Output of srcSlice's Slice Profiles.

Table 1
Variable-level slice profiles for code snippets in Fig. 1(a) and Fig. 1(b).

File	Function	Variable	Def	Use	Dvars	Ptrs	Cfuncs
Fig. 1(a)	sumProd	i	{4}	{4, 5, 6}	{prod, sum}	{ }	{ }
		prod	{3, 6}	{7}	{ }	{ }	{foo {2}}
		sum	{2, 5}	{7}	{ }	{ }	{foo {1}}
		n	{1}	{4}	{ }	{ }	{ }
Fig. 1(b)	sumProd_E	i	{4}	{5, 6, 7, 9}	{prod, sum}	{ }	{ }
		prod	{3, 7}	{8}	{ }	{ }	{foo {2}}
		sum	{2, 6}	{8}	{ }	{ }	{foo {1}}
		n	{1}	{5}	{ }	{ }	{ }

Table 2
Function-level slice profiles for code snippets in Fig. 1(a) and Fig. 1(b).

File	Function	Variable	Def	Use	Dvars	Ptrs	Cfuncs
Fig. 1(a)	sumProd	*all*	{1-6}	{4-7}	{prod, sum}	{ }	{foo {1, 2}}
Fig. 1(b)	sumProd_E	*all*	{1-4,6,7}	{5-9}	{prod, sum}	{ }	{foo {1, 2}}

Table 3
Final slices for code snippets in Fig. 1(a) and Fig. 1(b) based on the slice profiles as computed in Tables 1 and 2.

File	Function	Variable	Variable-level slice	Function-level slice
Fig. 1(a)	sumProd	i	{4, 5, 6, 7}	{1-7}
		prod	{3, 6, 7}	
		sum	{2, 5, 7}	
		n	{1, 4}	
Fig. 1(b)	sumProd_E	i	{4, 5, 6, 7, 8, 9}	{1-9}
		prod	{3, 7, 8}	
		sum	{2, 6, 8}	
		n	{1, 5}	

The complete slice is computed by srcSlice by taking the union of the slice profile of the slicing variable (that is, definition (*Def*) and use (*Use*) sets) with the slice profiles of the identifiers included in the dependent variables (*Dvars*), called functions (*Cfuncs*), and pointers (*Ptrs*) fields of the slicing variable, minus any lines that are before the initial definition of the slicing variable (that is, the set $\{1, \ldots, def(v) - 1\}$). Thus, in Fig. 1(a) since the *prod* and *sum* variables are data dependent on the *i* variable, the complete slice for $i = def(i) \cup use(i) \cup slice(prod) \cup slice(sum) - \{1, 2, 3\}$. This comes out to {4, 5, 6, 7}. These slices do not include the complete control-flow information as retrieved by srcSlice. Instead, we use limited-flow information that includes dependent variables, called functions, and aliases. The capability to generate control-flow paths is already present in srcSlice (Alomari et al., 2014a), however, storing this information incurs computational overhead. Not retrieving this information will affect some of the cognitive complexity slice-based metrics that are used to calculate the slicing vectors. Mainly, the *Slice Size* and the *Slice Coverage* metrics.

However, since our experiments depend on slices that computed at the function and file levels we found that not retrieving this information has a lower impact on the calculated slicing vectors. As the control flow information is not used and no control dependence computed, the slices computed and used by srcClone are similar to flow-insensitive data-only slices.

To cater to larger code fragments in the context of clone detection, we compute the slice profiles at the function level. Table 2 shows the slice profiles computed for the same snippets at the function level by taking the union of corresponding *Def*, *Use*, *Dvars*, *Ptrs*, and *Cfuncs* for all variables inside a given function. This could be repeated to compute the slice profiles at the file-level. Table 3 shows the final slices as computed by srcSlice at both the variable and function levels.

3. The srcClone approach: Clone detection using slicing

We describe herein our approach for detecting clones using program slicing. srcClone detects software clones through determining how similar the slicing information is for the components. Program slicing is a cognitive mechanism useful for locating code related to a specific feature/computation of interest. The slicing process removes unrelated parts of the program that have no impact on the semantics of interest to an analyst. In our work, each slice represents a segment of code that preserves dependencies that are relevant to a specific point of interest. Thus, we are able to identify system components with similar behaviors by comparing slices of two or more artifacts.

Our general process is as follows. Our first step is to translate the corresponding code into srcML, allowing us to employ srcSlice to retrieve data and information for each variable, function, and file in the system. srcClone then parses the corresponding output from srcSlice and computes slice-based metrics focused on complexity at various granularity levels. We follow this by turning slice profiles into slicing vectors through an analysis of the slice profiles' slices metrics. Using these slice vectors, we detect clones at both the file and function levels of granularity, and calculate them at their respective variable levels. srcClone then performs an efficient near-neighbor and hashing algorithm that clusters the vectors by considering their respective vector distances. By classifying vectors with hash values, we need only compare and contrast vectors sharing identical hash values. This allows us to optimize the amount of comparisons by a factor of the number of srcClone generated hash values.

3.1. Implementation and components

Our implementation of srcClone incorporates of a number of primary components, namely srcML, srcSlice, slice-based cognitive complexity metrics, slicing vectors, and LSH hashing and clustering. srcClone's pipeline is presented in Fig. 2. As we illustrate, srcClone generates the srcML over the source code in the form of git repository or archive to produce an XML representation of the code and abstract syntactic information from the AST. This information is crucial to srcSlice to recognize program dependencies as needed during the slicing process. We use the XML as input to srcSlice that generates the list of slicing profiles for each variable in the system. We parse these profiles to calculate the final slices and their corresponding slice-based complexity metrics and then 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. 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 versions of srcML and srcSlice support C, C++, Java, and C# languages so does srcClone. Prior to this work, it detected only clones on C and C++ languages.

3.2. Formal definitions

For us to detect clone fragments, we need allow modifications between these fragments to some extent. We must be aware that if the degree of modifications is too much, then we will lose some precision. As such, the similarity measures that we employ determine the clone types we detect. We also have to consider what srcClone should allow for the minimum code fragment size that is worthwhile. In different contexts and applications, minimal size is handled differently and impacts the identified clone quality notably. As some examples, SourcererCC (Sajnani et al., 2016) considers 6 LOC in their functional granularity as does Nicad (Roy and Cordy, 2008b) in their standard blocks. Deckard (Jiang et al., 2007) focuses on tokens of 50 or more when evaluating structured sub-trees. Other examples include those that consider begin-end blocks of specific minimum sizes or PDG subgraphs (Roy et al., 2009; Bellon et al., 2007; Rattan et al., 2013). High false positives can be the result of having too small or too large code fragments.

Identifying all classes of clones rather than sets of a pair of clones, as some tools do, is useful but presents a challenging and limiting problem. A naive approach for identifying all clone classes requires an exhaustive search of the source code, which is an intractable approach for larger bodies of code. This would necessitate comparing every possible fragment of code to every other similarly sized fragment of code in the same program. Even for finding clones within a single file, the time required is exponential on the number of comparable fragments in the file. With slicing vectors at different granularity levels, our approach can generate slicing vectors for code fragments of different size levels and provide a hierarchical internal semantic representation of the system that can be used to find a similar code of any size. Our focus and target is to be scalable and work with larger code bases, thus we focus on clone pairs only.

To explicate and help clarify, srcClone uses the next definitions of code fragments and clone types, which we defined in our previous work (Alomari and Stephan, 2020).

Definition 3.1 (*Code Fragment*)**.** Denoted by cf, is a set of code lines, not necessarily contiguous. It is the slice as computed at the variable-level, function-level, or file-level.

Definition 3.2 (*Clone Pair & Clone Class*)**.** We identify two code fragments cf_1 and cf_2 as a *clone pair* if they have similar corresponding slices s_1 and s_2. A group of similar code fragments form a *clone class*.

Definition 3.3 (*Clone Types*)**.** We follow the standard definitions (Roy et al., 2009; Bellon et al., 2007):

1. Type-1: Syntactically identical cfs, except for differences in white space, layout, and comments.
2. Type-2: Syntactically identical cfs, except for differences in identifier names, literal values, in addition to Type-1 clone differences.
3. Type-3: Syntactically similar cfs with further modifications, including added, modified and/or removed statements, in addition to Type-1 and Type-2 clone differences.
4. Type-4: Syntactically dissimilar cfs that perform the same overall computation but have different syntactic structures/implementations. They are also known as semantic clones.

Most of existing clone approaches detect clones up to Type-3. Type-4 clones requires essential analysis of the actual behavior and a deeper understanding of the intent of code fragment. In general, this is a hard problem, therefore, very few approaches

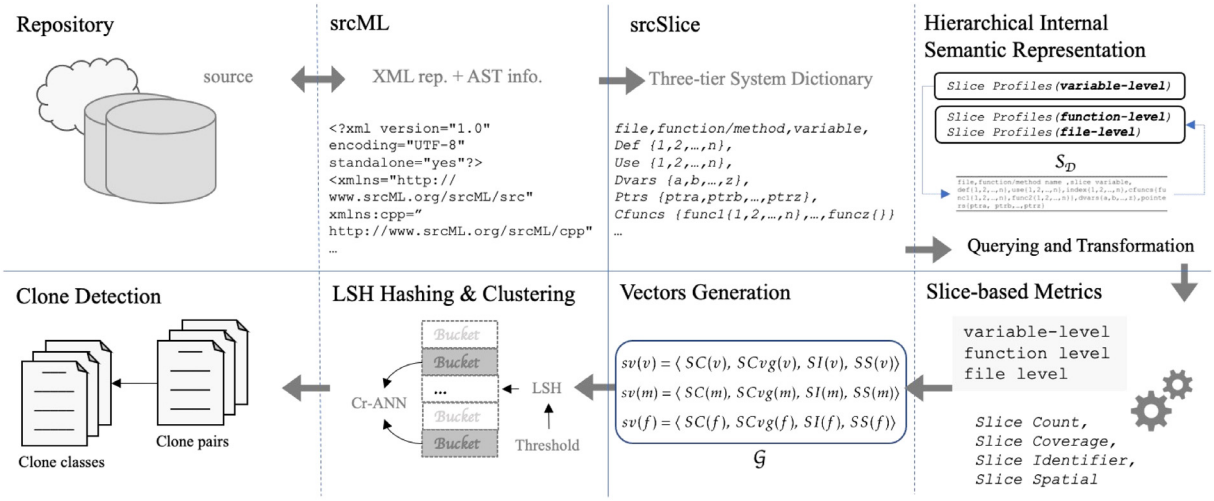

Fig. 2. The clone detection pipeline process using srcClone.

attempt to detect Type-4 clones (Saini et al., 2018). Researchers agree that in between Type-3 and Type-4 clones there is a spectrum of other clones that exhibit some syntactic-based similarities but are still very hard to detect (Saini et al., 2018; Sajnani et al., 2016; Svajlenko and Roy, 2015, 2016). For example, the state-of-the-art BigCloneBench (Svajlenko and Roy, 2016) reflected the vastness of this spectrum by including subcategories between Type-3 and Type-4, as follows,

1. Very Strongly Type-3 (denoted by *VST3*) with a syntactical similarity in the range [90%–100%),
2. Strongly Type-3 (denoted by *ST3*) with a syntactical similarity in the range [70%–90%),
3. Moderately Type-3 (denoted by *MT3*) with a syntactical similarity in the range [50%–70%),
4. Weakly Type-3 or Type-4 (denoted by *WT3/T4*) with a syntactical similarity in the range [0%–50%).

It is obvious in using these categories that srcClone needs some level of semantic awareness to detect WT3/T4 clones since their syntactical similarity is <50%. This type of *semantic* clone is generally out of the scope of much of the existing clone detection research, especially since the goal of clone detection is similarity detection, with existing approaches struggling to detect this type effectively (Sajnani et al., 2016; Komondoor and Horwitz, 2001; Krinke, 2001; Gabel et al., 2008). These types of clones are within the scope of our research and our resulting srcClone approach. Section 4.2 provides more details about BigCloneBench and its clone categories.

In our work, we view clones as semantically similar code fragments. We detect these semantic aspects using program slicing. Thus, it is important to define our notion of similar slices which we defined in our previous work (Alomari and Stephan, 2020).

Definition 3.4 (*Similarity of Slices*)**.** s_1 and s_2, are similar slices if their corresponding slicing vectors, sv_1 and sv_2, are σ-similar, given a specified threshold σ.

Definition 3.5 (*Slicing Vectors Similarity*)**.** sv_1 and sv_2, are σ-similar slicing vectors for a threshold σ, if $\mathcal{S}\ (sv_1,\ sv_2) \leq \sigma$. $\mathcal{S}$ refers to a similarity function.

Since our approach is not a PDG- nor Tree-based as some others (Jiang et al., 2007; Gabel et al., 2008; Baxter et al., 1998), we use a distance metric on d-dimensional slicing vectors as a similarity function, as we will explain in Section 3.4.

3.3. Vectorizing the slices

We devised *slicing vectors* that encapsulate the slices' structural information. Vectorization is a crucial step within srcClone. To accomplish this we were first motivated by Deckard's *characteristic vectors* (Jiang et al., 2007). These are a numerical approximation of a particular sub-tree, which are calculated by approaches as the quantity of complete binary trees types needed to approximate a given tree.

Our work builds on characteristic vectors in multiple ways. Each slicing vector is made up of a fixed sized of the slice-based metric numbers. Additionally, no tree is used by srcClone in vector generation nor is there a need for value approximation of the dimensions of the vectors. In our case, slicing vectors are made up of cognitive complexity slice-based metrics that srcClone devises through the slice data within the respective slice. We use metrics that were introduced recently by Alqadi (2019), Alqadi and Maletic (2020) to identify the cognitive complexity of source-code modules, then accordingly identify those modules that are difficult to comprehend. The metrics include the following measures (Alqadi, 2019),

- *Slice Count*, *SC*, is the number of slices within a module. For a variable, this is equal to the number of slice profiles united to form the final slice. For a function, this is equal to the number of slices for all variables included in the function. The *SC* for a file is equal to the number of functions' slices.
- *Slice Size*, *SZ*, is the number of statements in a complete final slice.
- *Slice Coverage*, *SCvg*, is the slice size relative to the module size measured in LOC.
- *Slice Identifier*, *SI*, is the number of distinct identifiers within a slice. This is the number of variables and method invocations in *Dvars*, *Ptrs*, and *Cfuncs* fields in the slice. We consider *SI* as a clone impact set, denoted by $Clone_{Imp}$, for each slice. It plays a crucial role in impact analysis; in change impact analysis the number of change locations that need to be considered is related to the size of this set.
- *Slice Spatial*, *SS*, is the spatial distance in LOC between the first definition and the last use of the slicing variable divided by the module size, such as $slicedistance = (S_l - S_f)/w$, where S_f is the first statement in the slice, S_l is the last statement in the slice, and w is the module size measured in LOC.

These slice-based metrics can be computed by srcClone at different levels of granularity, such as file, function, and variable.

Generally speaking, high value of these metrics is in providing an indication of more logically complex code and behaviors that are difficult to comprehend. We use these slice-based metrics in our work since these metrics have been successfully employed and empirically evaluated in Alqadi (2019), Alqadi and Maletic (2020) to provide more insights into the program behavior than traditional code-based metrics.

For a system dictionary, srcClone parses it into a data structure, calculates the final slices, and then the slicing-based complexity metrics. We use these metrics to generate slicing vectors for the system's slices. The slicing vector, denoted by sv, for a given variable's slice, has the following dimensions (Alomari and Stephan, 2020),

$$sv(v) = \langle\ SC(v),\ SCvg(v),\ SI(v),\ SS(v)\rangle \tag{1}$$

The slice-based complexity metrics are represented by each dimension. In cases were two code components are clone pairs, they will have similar vectors. As expected, if one of the code fragments is modified in a minor way, the corresponding slicing vectors will have minor changes when it comes to their slices. At this point, it is no longer necessary to retain anymore information about the variable, including metadata and code structure. Despite this, we are able to still detect renamed, Type 2 clones, efficiently since two variables with identical slicing vectors yet different names will still be identified as clones by srcClone.

We illustrate this in Table 4, which is an updated version of the one from our past work (Alomari and Stephan, 2020) as it calculates the slicing vector using our updated algorithm. It illustrates the slicing vectors resulting from the slices we computed earlier for variables and functions in Table 3, along with the $Clone_{Imp}$ set and the Hamming distance $\mathcal{D}_{\mathcal{H}}$ both at the function level. In the first variable-level instance, we see that the dimensions of the vectors are different in *sumProd_E* from those in function *sumProd*, which we bold. There is a minor change in $SCvg$ is noticed in *sumProd_E* resulting from the difference in function size.

At this point, we have reduced the slices to a set of vectors at the variable level of granularity. We now need to compare n slicing vectors corresponding to the n variables. When comparing methods, we generate a unique method slice profile for every method and creates slice profiles in the same manner as variables. In this Equation, the sv for m have dimensions that are analogous to what we did for variables:

$$sv(m) = \langle\ SC(m),\ SCvg(m),\ SI(m),\ SS(m)\rangle \tag{2}$$

srcClone repeats the same process for each file, where the sv for f is,

$$sv(f) = \langle\ SC(f),\ SCvg(f),\ SI(f),\ SS(f)\rangle \tag{3}$$

Using our updated algorithm from that in our past work, we see in Table 4 the slicing vector for both functions is identical and is equal to $\langle 4, 1, 3, 0.9\rangle$. The number of variables is the same for every function, 4. The $SCvg$ is the same despite having different function sizes. This stems from the size of the union between all slices for all variables in each function and its relation to its corresponding size. For example, the $SCvg$ for *sumProd* is equal to $\frac{SZ}{function\ size}$ and this equal to $\frac{s(i)\cup s(prod)\cup s(sum)\cup s(n)}{function\ size} = \frac{\{4-7\}\cup\{3,6,7\}\cup\{2,5,7\}\cup\{1,4\}}{7\ lOC} = \frac{7}{7} = 1$. And for *sumProd_E* is equal to 1. The *SI* in both functions are also similar since the union between *Dvars*, *Ptrs*, and *Cfuncs* sets for both functions are identical and equal to 3. Finally, the *SS* for both functions are similar since the slice distance for each function is relatively the same when comparing it to its corresponding function size.

Algorithm 1, shows how we generate slicing vectors for the three granularity levels of slices, denoted by v-level, m-level, and f-level. Given a system dictionary $\mathcal{S}_{\mathcal{D}}$, we create slicing vectors for the respective slices in a single pass. The primary steps are iterating over each slice profile (*denoted by sp*) (line 4), calculating its complete slice (line 5), calculating the slice-based cognitive complexity metrics (lines 6–10), and encoding the slicing vector (line 11). This algorithm returns the $\mathcal{G}$ set with all granularity levels of slicing vectors in the system. We use the same algorithm to calculate slicing vectors at the function and file levels. This algorithm is extended from our earlier version of the algorithm (Alomari and Stephan, 2020) by using the complete final slices instead of just the slice profiles and using slice-based complexity metrics to build the slicing vectors instead of using just the size of the slice profile's fields, e.g., $|Def|$.

Algorithm 1: v- , m- , or f-level Slicing Vectors Generation

```
1  function SVG (S_D: system dictionary): vectors set G
       /* S_D = {sp_1, sp_2, ..., sp_n}                                   */
2      begin
3          G ⟵ ϕ
4          for each sp ∈ S_D do
5              CS ← ComputeCompleteSlice(sp)   // More details are given in past work Alomari et al. (2014b), Newman et al. (2016)
6              SC ← # of united sps used to compute CS
7              SZ ← |{CS}|  // All statements of all united sps
8              SCvg ← SZ/w   // w is the module size in LOC
9              SI ← |{Dvars} ∪ {ptrs} ∪ {Cfuncs}|
10             SS ← (S_l − S_f)/w   // S_f & S_l are the 1st and last statements in the slice
11             sv ⟵ ⟨ SC, SCvg, SI, SS ⟩
12         G ← G ⋃ {sv}
13     return G
```

3.4. Similarity and matching

To detect clones and their pairs, we analyze each slicing vector by looking for similarity with all other vectors. In doing so, we face several challenges. These include the similarity of dimensions, near-miss clone detection, and scalability concerns. To address clones of Type 3, near-miss, srcClone compares dimensions of similarity rather then trying to find only exact equality matches. For large software systems, pair-wise similarity analysis is infeasible from a computational perspective. We tackle this by considering only pairs that are more likely to be similar. To do so, we employ Locality Sensitive Hashing (LSH) (Indyk and Motwani, 1998) in our approach and srcClone implementation. LSH allows for ascertaining what specific items in a set are similar to one another. LSH hashes items into "buckets", which contain items that are likely to be similar to one another. This reduces the number of comparisons significantly as demonstrated in AST-based clone detection approaches, among others (Gabel et al., 2008; Jiang et al., 2007).

In our adaptation of LSH, we use it to hash slicing vectors, determine neighbor sets that are close to each other, and provide clone reports. The buckets have a high chance of containing similar slicing vectors. Vectors that are not similar to one another will be in different buckets in most cases. This results in clone detection is better suited to find clones with different degrees of similarity. Using this adapted algorithm, we find srcClone is able to discover clones in a few minutes from millions of vectors. As we present in our evaluation later in this paper, we do so with a very low false positive rate. We present our implementation of the LSH algorithm, which we established in previous work (Alomari and Stephan, 2020),

Table 4
Slicing vectors for variable and function levels for the slices computed in Table 3.

File	Function	Variable	Slicing vectors (*svs*)		*Clone*$_{Imp}$	$\mathcal{D}_{\mathcal{H}}$
			Variable-level	Function-level		
Fig. 1(a)	sumProd	i	⟨3, 0.6, 2, 0.4⟩	⟨4, 1, 3, 0.9⟩	{prod, sum, foo()}	0.0
		prod	⟨2, 0.4, 1, 0.6⟩			
		sum	⟨2, 0.4, 1, 0.7⟩			
		n	⟨1, 0.3, 0, 0.4⟩			
Fig. 1(b)	sumProd_E	i	⟨3, **0.7**, 2, **0.6**⟩	⟨4, 1, 3, 0.9⟩	{prod, sum, foo()}	
		prod	⟨2, **0.3**, 1, 0.6⟩			
		sum	⟨2, **0.3**, 1, 0.7⟩			
		n	⟨1, **0.2**, 0, 0.4⟩			

Definition 3.6 ((r_1, r_2, p_1, p_2)-*Locality Sensitive Hashing*)**.** $\mathcal{D}$ is a distance measure. $r_1 < r_2$ are two distances in this measure. Consider a set of slicing vectors $\mathcal{G}$, and a collision probability values $p_1 > p_2$. Thus, 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;

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

A higher probability of similarity is indicated by our algorithm when there is a smaller distance between vectors. Distances ideally will address three requirements: symmetry, triangle inequality, and non-negativity. Such as Hamming or Euclidean distances. Next, we define the two distance measures for numerical slicing vectors that we use in our implementation,

Definition 3.7 (*Distance Measures on Slicing Vectors*)**.** Given a set of slicing vectors $\mathcal{G}$, let $v_1 = \langle x_1, \ldots, x_d\rangle$ and $v_2 = \langle y_1, \ldots, y_d\rangle$ be two d-dimensional vectors $\in \mathcal{G}$. The *Hamming distance* of v_1 and v_2, $\mathcal{D}_{\mathcal{H}}(v_1, v_2) = \sum_{i=1}^{d} |x_i - y_i|$. The *Euclidean distance* of v_1 and v_2, $\mathcal{D}_{\mathcal{E}}(v_1, v_2) = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$.

These distance measures are easy to calculate and there exists efficient algorithms for near-neighbor queries for numerical vectors. In Euclidean distance, there exist a locality-sensitive family of hash functions for any $r_1 \le r_2$, which is applicable for any number of dimensions. However, it is not known what exactly the probability, p_1, is that two vectors at distance r_1 hash to the same bucket. However, it is certain that it is greater than p_2, which is the probability that two vectors at distance r_2 hash to the same bucket. This is because this probability increases as the distance shrink. Thus, even if p_1 and p_2 cannot be calculated easily by an algorithm, there is a (r_1, r_2, p_1, p_2)-sensitive family of hash functions for any $r_1 \le r_2$ and any given number of dimensions (Rajaraman and Ullman, 2011). In contrast, there is one known value on lower bounds for LSH under Hamming distance (Motwani et al., 2006). Without loss of generality, if we assume that vectors are Boolean with dimensions of either 0 or 1, then $\mathcal{D}_{\mathcal{H}}(v_1, v_2) = \delta(x_i, y_i)$, where $\delta(x_i, y_i) = 1$, if $x_i \ne y_i$, and 0, if $x_i = y_i$. Thus, the Hamming distance is the number of dimensions in which v_1 and v_2 are different. When v_1 and v_2 are identical then the hamming distance is equal to *zero*. This 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 the d positions, thus they agree in $d - \mathcal{D}_{\mathcal{H}}(v_1, v_2)$ positions. Hence Prob $[h(v_1) = h(v_2)] = 1 - \mathcal{D}_{\mathcal{H}}(v_1, v_2)/d$. Therefore, we can deduce the following lemma as we did in our past work (Alomari and Stephan, 2020),

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. We build off of Definition 3.6. Let $p_1 = 1 - r_1/d$, and $p_2 = 1 - r_2/d$. A family $\mathcal{H}$ of hash functions $h : \mathcal{G} \to \mathcal{U}$ 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;

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

By applying the above requirements for a distance measure, we see that Hamming distances are non-negative, symmetric, and the distance between two analogous vectors is 0. Also, they are transitive, as the distance between any three vectors satisfy the triangle inequality; $\mathcal{D}_{\mathcal{H}}(v_1, v_2) + \mathcal{D}_{\mathcal{H}}(v_2, v_3) \ge \mathcal{D}_{\mathcal{H}}(v_1, v_3)$. Thus, we able to use Hamming distance as a metric on the d-dimensional vectors. To ensure vectors near each one another, we employ LSH to hash the vectors using the relevant Hamming distances between them. As we are looking for similarity rather than direct equality, we consider multiple degrees of thresholds of similarity in order to be specific about the similarity between two slicing vectors. This has us represent slices through numerical values and causes a re-framing of the similarity problem to a similar vector detection problem. We now elaborate on how we hash the slicing vectors and cluster those that are similar.

3.5. Hashing and clustering

The aforementioned LSH algorithm helps SRCCLONE to efficiently find near neighbors of a given slicing vector v. Given a set $\mathcal{G}$ of n slicing vectors, the goal of SRCCLONE is to build a data structure that reports any vector within a given distance r to v. However, since this problem suffer from the curse of dimensionality, researchers have been proposed several approximations, such as (c, r)-Approximate Near Neighbor algorithm (ANN) (Indyk and Motwani, 1998). This algorithm returns a data structure that reports any vector with distance at most cr from the query vector v, with an approximation factor $c > 1$. This provided that there exists a vector within distance r from the query vector v. More formally,

Definition 3.8 ((c, r)-*Approximate Near Neighbor (Alomari and Stephan, 2020)*)**.** Given a query vector v, a set of vectors $\mathcal{G}$ of size n, a distance r, and a factor $c > 1$, $\mathcal{U} = \{u \in \mathcal{G} \mid \mathcal{D}_{\mathcal{H}}(u, v) \le cr\}$ is called an cr-ANN set of v, and any $u \in \mathcal{U}$ is a (c, r)-approximate near neighbor of v.

This definition implies that we can post-process the set $\mathcal{G}$ to create a data structure, called cr-ANN, that contains clustered closer/similar vectors. Algorithm 2 shows our hashing process and our algorithm for finding the cr-ANN set for all the vectors in $\mathcal{G}$. The largest distance that is allowed between a query vector and its neighbors is r. This is the threshold σ defined in Definition 3.5. As shown in the algorithm, all vectors are stored into LSH hash tables (line 2). We feeding 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 set 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 code fragments that do not have a match in the original code. For example, newly added variables or methods. In our work, the number of dimensions for any slicing vector is fixed and is equal to *four*. Therefore, $p_1 = 1 - \frac{r_1}{4}$. We set SRCCLONE's threshold σ and r value to 1, which means one position or dimension is allowed to be different between two vectors. This will give a p_1 value equal to 0.75 or 75% minimum similarity.

Algorithm 2: LSH Hashing and Clustering

```
Input: G: contains n slicing vectors, r : distance, p1 : minimal
       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
```

3.6. Properties and limitations

We have discussed the detailed properties of our approach in our previous work (Alomari and Stephan, 2020). Specifically, we showed how our slice-based approach is not affected by layout differences and formatting between code fragments. However, layout in this current work can now affect the computation of the *Slice Spatial* metric and thus the slicing vector. The decompositional slicing definition used by SRCCLONE returns all relevant computations involving a slicing variable(s). Our approach, unlike most approaches, compare different sizes of code fragments at different granularity levels. A code fragment may contains one slice profile or several. There is no string, tree, token, or graph to compare.

As we show in Table 4, if we compare the slicing vectors at the variable level, we can see that the similarity for all vectors is ≥50%, including variable *i*. However, at the function level both vectors are exactly identical. If the similarity threshold r is equal to 1, both vectors differ in one position. Thus, p_1 will equal 0.75, and both functions form a clone pair with a similarity value equal to 100%. Alternatively, we can say that both functions are similar since the slicing vectors at the variable levels are similar with a similarity values ≥50%. However, the similarity in our approach is affected when new variables are added to the cloned function. Therefore, we delete variables that have no corresponding variables in the original code using Algorithm 2.

In our previous work (Alomari and Stephan, 2020), we also discussed how our slice-based method needs no normalizing transformations as the case in many clone detection techniques (Roy et al., 2009). Also, we have discussed the effectiveness of our approach in detecting all the types of clones in detail and the intuitions behind the approach should work properly. Please refer to our previous work for more details and a better understanding of all the aforementioned properties. Next, we adapted one example from our previous work and explain it herein.

Consider the code snippets in Figs. 3(a) and 3(b). They are similar, both perform the same overall computation, however extra statements (highlighted) and variables (*start* and *finish*) are included in Fig. 3(b) to time the *while* loop. Existing textual-based clone detectors are not able to detect these non-contiguous and interleaved clones (Gabel et al., 2008). Using SRCCLONE, Table 5 shows the slice profiles we generated for the code snippets in Figs. 3(a) and 3(b) at the file level. As shown, there is just a difference in the *Def* and *Use* fields. This is expected since the only difference between the two code snippets is defining two variables, then using them. The change in the literal names/values does not affect our results.

These differences are more noticeable in the generated slicing vectors as we show in Table 6. These vectors are for the variable, function, and file levels using Eqs. (1), (2), and (3) respectively. As we can see, all vectors at the variable level are identical except for the variables inside the *main* function since it contains the new added variables, *start* and *finish*. This is reasonable since the vectors at the *main* function are the only affected vectors with this addition. This difference at the function level also impacts the slicing vectors generated at the file level, and again the *Def* and *Use* fields in addition to the number of variables. However, the number of functions inside both files remains the same. Therefore, we now understand functions *foo* and *foo_timed* are identical, as are the function *fun* in both versions. The similarity between *main* function in both files is likely week to be identified as a semantic clone using Eq. (2). Therefore, we are able to increase the similarity to detect those two files as similar by comparing the slicing vectors either at the variable level since the slicing vectors for *start* and *finish* are deleted early using Algorithm 2. Alternatively, we can consider the file level in which the slicing vectors are similar with 100% similarity.

4. Evaluation and discussion

Assessing and analyzing clone detectors is still an open challenge (Bellon et al., 2007; Svajlenko and Roy, 2014). They have been evaluated by researchers often using the information retrieval measures *precision*, how many of identified clones are actual clones, and *recall*, how many of the actual clones are identified. Other evaluation measures may include the (1) levels or types of clones, for example, textual or structural, (2) feasibility with respect to the realistic state of large software systems such as efficiency, scalability, and robustness, (3) the gap between the input and the working artifacts, such as source code vs the PDGs, and (4) the portability of the tool to different languages and platforms. We believe these are all important factors in the evaluation of clone detection techniques and attempt to cover them with our work. We evaluate the accuracy (precision and recall), performance (execution time), and scalability of SRCCLONE, and compare it to other leading and relevant clone detectors. Specifically, we conducted the following experiments to evaluate and validate our approach and our SRCCLONE tool,

1. In our previous work (Alomari and Stephan, 2020), 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 (2001), and (2) Gabel's (Gabel et al., 2008). 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 Graph-based, such as the DUPLIX (Krinke, 2001) and GPLAG tools (Liu et al., 2006). However, they are not using program slicing.

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

```
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;
19 }
```

(a) Example code snippet - Listing (1)

```
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;
23 }
```

(b) Semantically similar code - Listing (2)

Fig. 3. Motivation example as proposed by Gabel et al. (2008).

Table 5
File-level slice profiles for the code snippets in Figs. 3(a) and 3(b).

File	Function	Variable	Def	Use	Dvars	Ptrs	Cfuncs
Fig. 3(a)	all	all	{1, 5, 10, 11, 17}	{1-3, 5-7, 12, 13, 15-17}	{abc}	{i, abc}	{fun {1}, foo {1, 2}}
Fig. 3(b)	all	all	{1, 5, 10-13, 17, 21}	{1-3, 5-7, 14, 15, 18-21}	{abc}	{i, abc}	{fun {1}, **foo_timed** {1, 2}}

Table 6
Slicing vectors (*svs*) for Fig. 3(a) and Fig. 3(b) at the three granularity levels.

File	Function	Variable	Slicing vectors		
			Variable-level	Function-level	File-level
Fig. 3(a)	foo	y x	⟨2, 0.5, 1, 0.5⟩ ⟨3, 1.3, 2, 1.3⟩	⟨2, 1.5, 3, 1.5⟩	⟨3, 0.7, 5, 0.8⟩
	fun	z	⟨1, 0.8, 0, 0.5⟩	⟨1, 0.8, 0, 0.5⟩	
	main	i abc	⟨4, 1, 3, 1.5⟩ ⟨3, 0.8, 2, 1.5⟩	⟨2, 1.1, 4, 1.5⟩	
Fig. 3(b)	**foo_timed**	y **w**	⟨2, 0.5, 1, 0.5⟩ ⟨3, 1.3, 2, 1.3⟩	⟨2, 1.5, 3, 1.5⟩	⟨3, 0.7, 5, 0.8⟩
	fun	z	⟨1, 0.8, 0, 0.5⟩	⟨1, 0.8, 0, 0.5⟩	
	main	i abc **start** **finish**	⟨4, **0.7**, 3, **1.3**⟩ ⟨3, **0.6**, 2, **1.2**⟩ ⟨**1, 0.1, 0, 0.4**⟩ ⟨**1, 0.2, 0, 0.3**⟩	⟨**4**, 1.1, 4, **1.3**⟩	

2. To achieve evaluation rigor in the clones detection community, we completed an additional comparative study with some recent and well-known state-of-the-art clone detectors, even though they are not using program slicing. These tools are DECKARD (Jiang et al., 2007), NICAD (Roy and Cordy, 2008b), SOURCERERCC (Sajnani et al., 2016), OREO (Saini et al., 2018), FACOY (Kim et al., 2018), and SIAMESE (Ragkhitwetsagul and Krinke, 2019). These tools are described generally in the literature as the most popular and successful state-of-the-art tools and are publicly available (Svajlenko and Roy, 2014, 2015; Sajnani, 2016; Ragkhitwetsagul et al., 2018). In our comparison, we use an established state-of-the-art clone detection benchmark, called BigCloneBench (Svajlenko and Roy, 2015; Svajlenko et al., 2014).
3. We use established benchmark scenarios proposed by Roy et al. (2009) to perform qualitative analysis of SRCCLONE. In their paper, they compared and evaluated many well-known clone detection techniques that include classical and state-of-the-art techniques. They tailored 16 different hypothetical editing scenarios to represent typical changes to copy/pasted code. These scenarios are categorized under the four major types of clones as shown in Figs. 6(a) to 6(p).
4. Finally, we performed an extensive quantitative experiments on 191 releases of the Linux kernel that span ten years of development and contains around 87 MLOC. We measure the three clone granularity levels between releases and use the measures as data points for classification. Clones between releases are demonstrated as pairs of cloned slices.

Table 7
System sizes. White space and comments are counted. PDG = Program Dependence Graph.

System	LOC	Files	PDGs	Slice profiles (*sps*)
bison 1.29	24,8 K	89	28,548	1,939
Linux 2.6.16	6,6 M	15,762	–	656,081

At an abstract level, we have two primary research questions on which we focus:

- **RQ1:** Does srcClone detect accurate clones?
- **RQ2:** How efficient (time) and scalable (size) is srcClone?

For reference and reproducibility, we outline the specifications of the systems we used in our evaluation. For hardware, we used a 4 GHz Intel i7 computer with 8 GB DDR3, running macOS. From a software perspective, we employed the publicly available srcML version 0.9.5 and srcSlice Beta-1.0 version.[2] We set srcClone's threshold σ and r value to 1. This will give a p_1 values equal to 0.75.

4.1. Comparative results - Komondoor and Gabel

In this comparative study we used the largest systems the authors used to run their experiments, which are (1) the GNU *bison* Unix utility system version 1.29[3] and (2) the *Linux* kernel version 2.6.16.[4] 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. We also compared the execution times of the tools. The execution time for Komondoor comprises the three main steps of their algorithm: (1) time to find clone pairs, (2) time to eliminate subsumed clone pairs, and (3) time to combine pairs into groups. The Gabel's execution time includes the time needed to build the PDG, dumping the PDG information, and the execution times for both the AST-based clone detection algorithm and the PDG-based clone detection algorithm. The PDG and the AST times implicitly include the vector generations and the clustering processes times. Finally, the execution time for our srcClone tool includes (1) the slicing time (time to compute slice profiles) (2) time to compute complete slices, and generate the slicing vectors, (3) time to compute slice-based metrics, and (3) the LSH hashing and clustering.

Table 7 lists the approximate system sizes measured by LOC, number of files, PDGs, and slice profiles for srcClone. Table 8 shows the results of both tools in compare to srcClone. Komondoor's tool spent 1 h and 34 min to find 800 clone groups out of 28,5 K PDGs. The size of these groups ranges from 5 to 227 PDGs. In compare, srcClone found around 1316 clone groups within 2 min. Gabel's tool took 9 h and 35 min to find 255,108 clone groups. These groups range in size from 4 to 32 PDGs. In comparison, srcClone was able to find 345,103 similar slicing vectors within ≈3 min.

Our results indicate that srcClone finds more clone groups and is more efficient than the other tools we considered. We discuss the qualitative aspects of our comparison. Gabel et al. observe that semantic clones often contained within them syntactic clones, together with declaration statements and initializations. We present an example of some of these in Fig. 4. To verify our precision, we contrast the clones from Komondoor to those from our approach. As we demonstrate in the figure, this code clone exists in five instances within the *bison* system. Komondoor found two of those instances. Specifically, lines 5 to 8 in Fig. 4(a) and lines 6 to 9 in Fig. 4(c). srcClone found all five copies, as we illustrate in Figs. 4(a) to 4(e). We present slicing vectors for the functions that contain the snippets in Fig. 4(f). We see that at the variable level, *svs*, are similar for variables that are shared among the code. For example, the *svs* for variable *fp3* are 75% similar. In Komondoor's work, they were not able to identify line number 1 in both Fig. 4(a) and Fig. 4(c). This is attributed to the slicing process heuristic they employ. In their case, they slice backward from the nodes within the loops (*while* and *for*). They further include in their slice the nodes outside of the loops if the loop predicates correspond/match to one another. As a result of this design decision, their assignments to pointer *fp3* are not considered during clone detection.

4.2. Evaluation using BigCloneBench — State-of-the-art Tools

For these state-of-the-art comparison experiments, we consider all clones inside BigCloneBench that are six lines of code or 50 tokens in length or greater. This is the typical minimum clone size usually used for measuring recall as reported in various studies (Bellon et al., 2007; Svajlenko and Roy, 2015; Saini et al., 2018; Sajnani et al., 2016). We compare srcClone's performance against multiple clone detectors: Deckard (Jiang et al., 2007), NiCad (Roy and Cordy, 2008b), SourcererCC (Sajnani et al., 2016), and Oreo (Saini et al., 2018). We additionally consider two code clone search engines: FaCoY (Kim et al., 2018) and Siamese (Ragkhitwetsagul and Krinke, 2019). These tools, in addition to our experiments with Komondoor's and Gabel's, cover a good range of clone detection approach categories. Komondoor's and Gabel's are graph-based, Deckard is tree-based, NiCad is text-based, SourcererCC is a token-based, Oreo is a metrics-based, and FaCoY and Siamese both are code clone search engines that are token-based. We initially considered including CCFinderX (Kamiya et al., 2002) tool, however we decided not to because it is a token-based approach and it is already compared to the SourcererCC in other research (Sajnani et al., 2016) that found SourcererCC performed better than CCFinderX.

We use the BigCloneEval framework (Svajlenko and Roy, 2016) to measure the recall. This framework is built and used for performing recall evaluation experiments of clone detection tools using the BigCloneBench clone detection benchmark (Svajlenko and Roy, 2015; Svajlenko et al., 2014). BigCloneBench is a real-world benchmark of known "real" clones in the IJaDataset source repository,[5] a large inter-project repository consisting of three million source files (350 MLOC) from 25,000 open-source projects that are mined from SourceForge and Google Code. The current version of BigCloneBench includes over eight million validated clone pairs that spanning the four primary clone types. We recommend reading the BigCloneBench literature for more background on its workings (Svajlenko and Roy, 2015; Svajlenko et al., 2014).

BigCloneEval automates evaluation experiments and produces a thorough recall evaluation report that summarizes a tool's recall performance for BigCloneBench. Since the BigCloneBench and IJaDataset are very large, BigCloneEval contains only the source files needed to perform the recall measurement. Unfortunately, BigCloneEval does not measure the precision of clone detection. Therefore, we measured the precision of our tool by manually validating a chosen sample of its results. This is a common practice of clone detectors since there is no standard benchmark, tool, and methodology for measuring precision (Sajnani et al., 2016; Svajlenko and Roy, 2016; Ragkhitwetsagul and Krinke, 2019). We use the following author-recommended configurations for the tools we evaluated,

[2] Both can be acquired from www.srcml.org along with instructions and documentation.

[3] https://www.gnu.org/software/bison/.

[4] https://mirrors.edge.kernel.org/pub/linux/kernel/v2.6/.

[5] Available at https://jeffsvajlenko.weebly.com/bigcloneeval.html.

Table 8
CG, for Komondoor and Gabel this is the number of PDG nodes. For srcClone is the number of similar slicing vectors.

System	Execution time	Clone groups (CG)	Execution time	Clone groups (CG)
bison 1.29	**Komondoor**		**srcClone**	
	1 h 34 m 5 s	800	2 m 16 s	1316
Linux 2.6.16	**Gabel**		**srcClone**	
	9 h 48 m 3 s	255,108	3 m 21 s	345,103

```
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

Function	*sv (m)*	*Lines of Code*	*sv (fp3)*
(A) conflicts.c/count_sr_conflicts	⟨12, 0.7, 10, 0.9⟩	67	⟨2, 0.1, 1, **0.6**⟩
(B) lalr.c/compute_lookaheads	⟨9, 0.9, 8, 0.9⟩	40	⟨2, 0.1, 1, **0.5**⟩
(C) lalr.c/traverse	⟨9, 0.7, 8, 0.9⟩	54	⟨2, 0.1, 1, 0.8⟩
(D) conflicts.c/set_conflicts	⟨12, 0.5, 10, 0.9⟩	87	⟨2, 0.1, 1, 0.8⟩
(E) conflicts.c/print_reductions	⟨22, 0.6, 17, 0.9⟩	227	⟨2, 0.1, 1, 0.8⟩

(f) Slicing Vectors for Five Functions and One Common Variable, *fp3*

Fig. 4. GNU `bison-1.29/src/` unix utility system (Alomari and Stephan, 2020).

- DECKARD: Minimum tokens = 50, Token stride = 2, and Similarity threshold = 85.
- NICAD: Minimum lines = 6, Blind identifier normalization = True, Literal abstraction = True, and Similarity threshold = 70.
- SOURCERERCC: Minimum lines = 6, Function granularity, and Similarity threshold = 70.
- OREO: Minimum tokens = 15, Threshold for input partition = 60%, and Action filter threshold = 55.
- SRCCLONE: Similarity threshold = 75% or *r*-value = 1.

4.2.1. Recall results

Table 9 summarizes our recall results per clone type for SRCCLONE and all the other approaches we are considering. The numbers in parenthesis shown next to each category titles represent the number of clones in BigCloneBench. The (%) title show the recall percentage and (#) title show the number of clones detected for each clone category. The bold typeface represents the best recall numbers. Some of the recall numbers shown for other approaches are provided by validation work from research on SourceererCC's (Sajnani et al., 2016), Oreo's (Saini et al., 2018), FaCoY's (Kim et al., 2018), and Siamese's (Ragkhitwetsagul and Krinke, 2019) where their authors evaluated these tools using BigCloneBench.

As we illustrate in Table 9, SRCCLONE performs better than other tools mainly on *WT3/T4* with a recall score of 78%. For *VST3*, OREO performs the best on this category; 100%. However, the *VST3* recall by SRCCLONE is still relatively strong, with a percentage equal to 99%. For *ST3* and *MT3*, SIAMESE obtained the highest recall of 99% and 88%, respectively. Again, the *ST3* and *MT3* recall scores by SRCCLONE are still relatively strong, with a percentage equal to 95% and 77%, respectively. It is important to be aware that SRCCLONE is using a 75% similarity threshold, which we derived through tuning. We believe lowering it we would allow these clones to be detected, however, that would decrease the precision. SRCCLONE detects 1 to 2 orders of magnitude more clones than others on the harder-to-detect categories, such as *WT3/T4*. This is expected as our tool is designed to detect these harder-to-detect clones between Type-3 and/or Type-4. Compared to the other tools, where the highest recall is 17% by SIAMESE, our 78% in *WT3/T4* category is a great improvement. Overall, SRCCLONE has the best recall results and SIAMESE is the second.

Table 9
Recall and precision results on BigCloneBench. Values in bold highlight the largest value in the column. S = sample.

Tool	Recall results												Precision results
	T1 %	(35,802) #	T2 %	(4,577) #	VST3 %	(4,156) #	ST3 %	(15,031) #	MT3 %	(80,023) #	WT3/T4 %	(7,804,868) #	s = 400
Code clone detectors													
srcClone (Alomari and Stephan, 2020)	**100**	35,801	**100**	4,571	99	4,115	95	14,280	77	61,622	**78**	6,087,790	90.4%
Deckard (Jiang et al., 2007; Saini et al., 2018; Sajnani et al., 2016)	60	21,481	58	2,655	62	2,577	31	4,660	12	9,603	1	78,048	35%–93%
NiCad (Roy and Cordy, 2008b; Saini et al., 2018; Sajnani et al., 2016)	**100**	35,769	99	4,541	98	4,091	93	13,910	0.8	671	0	12	89%–**99**%
SourcererCC (Sajnani et al., 2016; Saini et al., 2018)	**100**	35,797	97	4,462	93	3,871	60	9,099	5	4,187	0	2,005	91%–98%
Oreo (Saini et al., 2018)	**100**	35,798	99	4,547	**100**	4,139	89	13,391	30	23,834	0.7	57,273	89.5%
Code clone search engines													
FaCoY (Kim et al., 2018)	65	23,271	90	4,119	67	2,784	69	10,371	41	32,809	10	780,486	92.8%[a]
Siamese (Ragkhitwetsagul and Krinke, 2019)	99	35,444	99	4,531	99	4,114	**99**	14,880	**88**	70,420	17	1,326,827	94.8%[b]

[a]FaCoY: Focus on manually analyzing sampled false positives, among 28 cases, for 26 cases, FaCoY points to the correct files but wrong locations than actual clones, and it completely fails in only 2 cases.
[b]Siamese: Mean Reciprocal Rank (MRR) - on average across queries, the reciprocal rank of the relevant document (clone) to each query in the search result.

Table 10
Slicing vectors for the 16 sub-scenarios mentioned in Fig. 6, S1(a)–S4(d).

Function	*Slicing* vector	*Rate* (precision)	*Coverage* (recall)
Scenario-1(a)	⟨4, 1, 3, 0.9⟩	●	100%
Scenario-1(b)	⟨4, 1, 3, 0.9⟩	●	
Scenario-1(c)	⟨4, 1, 3, 0.9⟩	●	
Scenario-2(a)	⟨4, 1, 3, 0.9⟩	●	100%
Scenario-2(b)	⟨4, 1, 3, 0.9⟩	●	
Scenario-2(c)	⟨4, 1, 3, 0.9⟩	●	
Scenario-2(d)	⟨4, 1, 3, 0.9⟩	●	
Scenario-3(a)	⟨4, 1, 3, 0.9⟩	●	100%
Scenario-3(b)	⟨4, 1, 3, 0.9⟩	●	
Scenario-3(c)	⟨4, 1, 3, 0.9⟩	●	
Scenario-3(d)	⟨4, **0.9**, **2**, 0.9⟩	◒	
Scenario-3(e)	⟨4, 1, 3, 0.9⟩	●	
Scenario-4(a)	⟨4, 1, 3, 0.9⟩	●	100%
Scenario-4(b)	⟨4, 1, 3, 0.9⟩	●	
Scenario-4(c)	⟨4, 1, 3, 0.9⟩	●	
Scenario-4(d)	⟨4, 1, 3, 0.9⟩	●	

We believe that our approach could assist in the automatic repair of faulty code as *WT3/T4* reflect different clones deviated from the original faulty code occurring in the system. Tools that support program repair tasks would reduce the time and effort gaps between finding the defect and fixing it by automatically repair program bugs. The srcClone approach could be used to find variants of buggy code (that is, clones) that both retain required functionality (that is, semantic) and also repairs the defect.

4.2.2. Precision results

We manually calculated and inspected the precision results of all clone detectors. For each tool in question, we selected randomly 400 clone pairs from the returned clones. This sample is statistically significant with a 95% confidence level and 5% of confidence interval. In this we follow the precedent and example set by others (Sajnani et al., 2016; Saini et al., 2018; Ragkhitwetsagul and Krinke, 2019). The first author took the role of a human investigator (judge) and validated the returned clones. The judge was kept blind from the source of each clone pair, thus did not know which tool has produced the result. In this task, we followed very strict definitions of what constitutes different categories of clones in order to minimize different opinions of the investigator and minimize investigator bias. To give maximum credit to the other tools, in addition to our precision results, we take the precision values of these tools (if it greater than our results) that is made publicly available either by the authors of the tools or other comparison experiments.

Table 9 illustrates the precision results for all the tools. srcClone's precision is 90.4%. All other tools, except Deckard and Oreo, performed better than srcClone. The precision of Deckard found to be 35% with 85% similarity threshold, and in other studies found to be 93% (using a lower similarity threshold) (Saini et al., 2018; Jiang et al., 2007). For NiCad we found precision values from 89 to 99% (with 70% threshold), depending on the configurations (Svajlenko et al., 2013). SourcererCC has a precision of 91 to 98%, with 70% similarity threshold (Sajnani et al., 2016; Saini et al., 2018). For Oreo, we found the same precision of 89.5% as they measured in their experiments using 55% similarity threshold. Finally, for FaCoY and Siamese, a precision of 92.8% and 94.8% are what they recorded based on information retrieval error measures such as MRR (mean reciprocal rank). For more details, see their work (Kim et al., 2018; Ragkhitwetsagul and Krinke, 2019).

In general, lower precision is a result of Type-2 normalizations (Bellon et al., 2007; Göde and Koschke, 2013), which causes false positives to have similar normalized sequences, as we discussed in our previous work (Alomari and Stephan, 2020). We know that some of our cognitive complexity slice-based metrics that we embedded in the slicing vectors are using LOC, such as $SCvg$ and SS. Therefore, this can affect the layout of srcClone. Also, We believe that our relatively low precision is a function

```
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.

Variable	*Slicing Vector*
i	⟨3, 0.6, 2, 0.4⟩
prod	⟨2, 0.4, 1, 0.6⟩
sum	⟨2, 0.4, 1, 0.7⟩
n	⟨1, 0.3, 0, 0.4⟩
sumProd	**⟨4, 1, 3, 0.9⟩**

(b) Slicing Vectors.

Fig. 5. Original copy by Roy et al. (2009) and its slicing vectors.

of using the srcSlice tool. As srcSlice performs lightweight data flow analysis, srcClone precision results are sub-optimal. However, this is the leading approach to slice such large-scale source code. We contend that lightweight analysis is acceptable for slice-based clone detection, by providing valuable candidate clones to maintainers including categories of clones seldom or never reached before by other tools, such as *WT3/T4*.

4.3. Qualitative analysis using Roy's scenario-based evaluation

To achieve qualitative analysis, we employ the research by Roy et al. (2009). Their work provides a validated benchmark that is predictive for both precision and recall. These scenarios are tailored to be challenging and represent clone detector obstacles. We use all 16 sub-scenarios to evaluate srcClone. We present the original code fragment, alongside our slicing vectors in Fig. 5. We present an overview of the scenarios in Fig. 6. We illustrate the corresponding slice vectors for the scenarios in Table 10. Following the same convention as Roy et al. (2009), we use the following rankings, as follows: (●) represents *very well*, (◕) for *well*, (◓) *medium*, (◔) *low*, (◔) *probably can*, (◯) *probably cannot*, and (○) *cannot*. For example, *well* means that the technique can detect the clones of the scenario but may return a few false positives, may miss some clones, and does not meet one or more of the *very well* rating criteria. For complete details of the rating criteria readers can read the work by Roy et al. (2009).

As we demonstrate in Table 10, we were successful detecting all scenarios' clones with a recall (coverage) equal to 100%. The precision is also *very well* for all scenarios, except for sub-scenario s3(d) shown in Fig. 6(k). This is because deleting line number 6 affects the slice vector for variables *i* and *prod* inside the *main* function. For example, the $sv(i)$ in the original copy is equal to ⟨3, 0.6, 2, 0.4⟩ as shown in Fig. 5(b), however, the $sv(i)$ in s3(d) scenario is equal to ⟨2, 0.4, 1, 0.4⟩. And for *prod* variable it is ⟨2, 0.3, 1, 0.6⟩ instead of ⟨2, 0.4, 1, 0.6⟩. This is the only change, all the other variables still having the same slicing vectors.

We evaluate and compare our work with all those approaches considered by Roy et al. In doing so, we contrast 42 clone detection approaches in their capabilities to detect the 16 sub scenarios. We show our results in Table 11 comparing our approach to other clone detectors in addition to the two new clone approaches released after Roy's study which are Oreo in 2018 and SourcererCC in 2016. In the table, recall is illustrated in the coverage column, while the pies are represent the precision, or false positive frequency. Scenario 4 is the most difficult to detect, with most of the techniques failing. Using the function granularity of srcClone, we were able to detect all 16 sub scenarios.

4.4. Quantitative analysis using linux kernel

This component of our evaluation aims to identify which granularity level of slicing vectors a good reflection of the extent of our discovered clones, and if it a fair representation of the size of the clones. We use in our evaluation 191 versions of the Linux kernel, version 1.1.0, released in 1994, to version 2.0.40, released in 2004, with a total number of 87 MLOC across all versions.

To categorize clones, looking at a single clone in isolation, is of minor use in the absence of predefined classification criteria. First, we take a comprehensive view of the subset of the history of all the clones and then we use some statistic methods to classify them. For each consecutive pair of versions, we consider the three granularity levels. The values of these levels are used as data points for classification. For categorization, we employ five quartiles, from Q0 to Q4. We use the Inter-Quartile Range (IQR = Q3 - Q1) as our measure of spread. This is helpful to identify outliers. We use the 1.5 *x* IQR rule: Anything $<$ Q1 - (1.5 *x* IQR) or $>$ Q3 + (1.5 *x* IQR) are outliers. We provide the data and quartiles for the Linux kernel in Table 12. Based on this classification, we decide to distribute the data into 5 regions that representing classes of clone size for each granularity. These regions are: extra-small, small, medium, large, and extra-large. For instance, at the variable level, the percentage of clones that has a size of $\leq$ 31 LOC is 25.1%. Our first observation that most of the clones are either small or extra small. Particularly, for all the 3 granularities, ≈75% of the clones are either small or extra-small. However, a larger clones also do exist. We manually inspected some selected versions and we found that most usually occurring clones were relevant to initialization of some new hardware drivers. The addition of new driver was frequently correlated with the introduction of new source clones. One possible explanation could be that developers try to use current tested and working driver's code instead of create a new one from scratch.

Given this pattern that most of the clones being quite small across all three granularity levels of clones, these results suggest that we should investigate if any of these clone levels are correlated statistically to one another. That is, if a clone has a modest number of variables, does that also an indication that a modest number of functions were cloned?. To answer this, we investigated the linear correlation coefficient and the coefficient of determination (Klett, 1972) to check if there is a correlation between any two clone levels for a singular size category.

The coefficient of determination, denoted by r^2, is a measure that allows us to decide our certainty in making prediction from a specific model/graph. The values for r^2 is such that $0 < r^2 < 1$, and represents the ferocity of the linear association between y and x. For example, if $r^2 = 0.75$, which means that 75% of the total variation in y could be explained by this linear relationship between x and y. The other 25% of the total variation in y remains unexplained. Before examining the values of r^2, let us present the percentages of commonality between the 3 granularity levels over the 5 categories. Fig. 7 shows the cross intersection distributions over the 3 levels of granularity. There are 3 groups of columns. The first one is the cross intersection of *variables* × *functions*, next is *variables* × *files*, and last is *functions* × *files*. Across the size categories, we can see that the cross intersection between each measure is similar. That is, when a clone has a small number of functions, it also has a small number of variables with

(a) S1(a), Whitespace changes

(b) S1(b), Commenting changes

(c) S1(c), Formatting changes

(d) S2(a), Systematic renaming of identifiers and literals

(e) S2(b), Renaming the identifiers, not necessarily systematically

(f) S2(c), Renaming data types and literal values, not necessarily consistent

(g) S2(d), Using expressions

(h) S3(a), Small insertions some lines

(i) S3(b), Small deletions some lines

(j) S3(c), Insert some new lines

(k) S3(d), Deletes some lines, line 6

(l) S3(e), Changes to whole line 5

(m) s4(a), Reorders data independent decls

(n) S4(b), Reorders data independent stmts

(o) S4(c), Reorders data dependent stmts

(p) S4(d), Replaces a control statement

Fig. 6. Taxonomy of Editing Scenarios for Different Clone Types. *Sx* = Scenario 1 or Type-1, Scenario 2 or Type-2, Scenario 3 or Type-3, or Scenario 4 or Type-4. S1 has 3 sub-scenarios, S2 has 4 sub-scenarios, S3 has 5 sub-scenarios, and S4 has 4 sub-scenarios.

the highest frequency that is compared to the total. To compute this distribution, we count the clones that have the same size category for both levels and divide that by the total number of Linux's clones. There is ≈50% commonality between the clones that are small, whereas all other clone sizes have commonality with <25%.

In order to examine the correlations between levels. First, we want to address the question, "if there are a large number of variables cloned, was there also a large number of functions cloned or a large number of files cloned?". For this, we need to calculate the correlation over categories. We provide the values of r^2 for Linux kernel in Table 13. There is a strong correlation between number of *variables* × *functions*, with only 70% of clones able to

Table 11
Scenario-based evaluation of the surveyed clone detection techniques and tools (Roy et al., 2009) 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	Johnson (Johnson, 1994b,?)	●	●	◕	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	19
	Duploc (Ducasse et al., 1999)	●	●	∘	∘	∘	∘	∘	◕	◕	∘	∘	◓	∘	∘	∘	∘	31
	sif (Manber et al., 1994)[a]	◓	◓	◓	∘	∘	◔	◔	◔	◔	∘	∘	∘	◔	◔	◔	∘	19
	DuDe (Wettel and Marinescu, 2005)	●	●	◔	∘	∘	∘	∘	◕	◕	◓	◓	◓	∘	∘	∘	∘	44
	SDD (Lee and Jeong, 2005)	◕	◕	◓	∘	∘	∘	∘	◕	◕	◔	◔	◔	∘	∘	∘	∘	50
	Marcus (Marcus and Maletic, 2001)[a]	◓	∘	◓	∘	∘	◓	◔	◓	◓	∘	∘	∘	◓	◓	◓	∘	56
	Basic NICAD (Roy and Cordy, 2008a)	●	●	●	◔	◔	◔	◔	◕	◕	◕	◕	◕	◔	◔	◔	∘	50
	Full NICAD (Roy and Cordy, 2008b)	●	●	●	●	◕	◕	◕	◕	◕	◕	◕	◕	◔	◔	◔	∘	75
	Nasehi (Nasehi et al., 2007)	◓	◓	◓	◓	◓	◓	◓	◓	◓	◔	◔	◔	○	○	○	◓	81
	Simian (Harris, 2020)	●	●	○	◓	◓	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	31
Token-based	Dup (Baker, 1995)	●	●	∘	●	∘	◕	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	25
	CCFinder(X) (Kamiya et al., 2002)	◕	◕	◕	◕	◕	◕	∘	◔	◔	∘	∘	∘	∘	∘	∘	∘	38
	Gemini (Ueda et al., 2002)[a]	◕	◕	◕	◕	◓	◕	∘	◓	◓	◓	◓	◓	◔	◔	◔	∘	69
	RTF (Basit and Jarzabek, 2007)	●	●	●	◕	◕	◕	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	38
	CP-Miner (Li et al., 2006)	◕	◕	◕	◕	◕	◕	◔	◕	◕	◓	◓	◓	◔	◔	◔	∘	75
	SHINOBI (Yamashina et al., 2008)[a]	◕	◕	◕	◕	◕	◕	∘	◔	◔	∘	∘	∘	∘	∘	∘	∘	38
	CPD (Project, 2020)	●	○	●	◓	◓	◓	○	∘	∘	∘	∘	∘	∘	∘	∘	∘	31
	Clone Detective (Landwerth, 2020)	◕	◕	◕	◕	◕	◕	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	38
	clones/iClones (Koschke et al., 2006)	●	●	◕	●	◕	◕	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	38
	SourcererCC (Sajnani et al., 2016)	●	●	●	●	●	◕	◕	◕	◕	◕	◓	◓	∘	∘	∘	∘	75
Tree-based	CloneDr (Baxter et al., 1998)	●	●	●	●	●	●	◔	◓	◓	◔	◓	◓	◔	∘	∘	∘	75
	Asta (Evans et al., 2009)	●	●	●	◓	◓	◓	◓	◓	◓	∘	∘	∘	∘	∘	∘	∘	56
	cpdetector/clast (Koschke et al., 2006)	●	●	●	◕	◕	◕	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	38
	Deckard (Jiang et al., 2007)	◕	◕	◕	◕	◕	◕	∘	◓	◓	◓	◓	◓	∘	∘	∘	∘	69
	Tairas (Tairas and Gray, 2006)	◕	◕	◕	○	○	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	25
	CloneDetection (Wahler et al., 2004)	◕	◕	◕	◕	◕	◓	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	38
	CloneDigger (Bulychev and Minea, 2008)	●	●	●	◕	◕	◓	◓	◓	◓	∘	∘	○	∘	∘	∘	∘	56
	C2D2 (Kraft et al., 2008)	◓	◓	◓	◓	◓	◓	◓	◓	◓	○	○	○	∘	∘	∘	∘	56
	Juillerat (Juillerat and Hirsbrunner, 2006)	●	●	●	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	∘	19
	SimScan (SimScan, 2020)	●	●	●	◕	◕	◕	◔	◔	◔	◔	◔	◔	∘	∘	∘	∘	38
	ccdiml (Bellon, 2002)	●	●	●	●	●	●	∘	◓	◓	◔	●	◓	∘	∘	∘	∘	69

(*continued on next page*)

be explained in a linear relationship, while 30% cannot. Between number of *functions* × *files*, only 60% of clones can be explained in a linear relationship, while 40% cannot. The remainder show no strong correlation and have a low r^2 value.

The two granularity levels of clones that demonstrate the strongest correlation are variables and functions. This is logical since functions are a group of slice profiles of variables. Thus, as the number of cloned variables increases, so does the number of cloned functions. The low correlation between variables and files suggest that clones can have the same number of variables but very different number of files. For example, a clone can contain one file with ten variables cloned or a clone can be ten files with

Table 11 (*continued*).

	● *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	%
Metrics-based	Kontogiannis (Kontogiannis et al., 1996)	◐	◐	◐	◐	◐	◐	◔	◔	◔	◔	◔	◔	◓	◓	◓	◔	100
	Mayrand (Mayrand et al., 1996)	◐	∘	◐	◐	◐	◐	◓	◓	◓	◔	◔	◔	◓	◓	◔	∘	88
	Dagenais (Dagenais et al., 1998)[a]	◓	◓	◓	◓	◓	◓	◓	◓	◓	◔	◔	◔	◓	◓	◔	○	69
	Merlo (Merlo et al., 2002, 2004)	◐	◐	◐	◐	◐	◐	◔	◓	◓	◔	◔	◔	◓	◓	◔	∘	94
	Davey (Davey et al., 1995)	◐	◐	◐	◐	◐	◐	◔	◓	◓	◔	◔	◔	◓	◔	◔	○	75
	Patenaude (Patenaude et al., 1999)	◐	◐	◐	◐	◐	◐	◔	◔	◔	◔	◔	◔	◔	◔	◔	∘	75
	Kontogiannis (Kontogiannis, 1997)	◐	◐	◐	◐	◐	◐	◔	◔	◔	◔	◔	◔	◔	◔	◔	◔	100
	Antoniol (Antoniol et al., 2001, 2002)	◐	◐	◐	◐	◐	◐	◓	◓	◓	◔	◔	◔	◓	◓	◔	∘	94
	Oreo (Saini et al., 2018)	●	●	●	●	●	●	●	◐	◐	◐	◐	◐	◓	◓	◔	◔	100
Graph-based	Duplix (Krinke, 2001)	◐	◐	◐	◐	◐	◐	◓	◓	◓	◓	◓	◓	◐	◓	◔	∘	94
	Komondoor (Komondoor and Horwitz, 2001)	◐	◐	◐	◐	◐	◐	◔	◔	◔	◔	◔	◔	◐	◓	◔	∘	75
	GPLAG (Liu et al., 2006)[a]	◐	◐	◐	◐	◐	◐	◔	◔	◔	◓	◓	◓	◐	◐	◔	◓	100
	Gabel (Gabel et al., 2008)	◐	◐	◐	◐	◐	◐	◐	◐	◐	◓	◓	◓	◓	◓	◓	∘	94
	SRCCLONE (ALOMARI AND STEPHAN, 2020)	●	●	●	●	●	●	●	●	●	●	◓	●	●	●	●	●	100

[a]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.

Table 12
Linux kernel clones categorized by size for each of the three levels over 191 versions spanning 10 years.

Quarterlies	Variable-level			Function-level			File-level		
Q0(Min)	0			3			1		
Q1	32			54			12		
Q2 Median	94			73			15		
Q3	297			90			28		
Q4(Max)	5419			1903			598		
IQR	265			36			17		
Boxplot	**Range**	**Freq.**	**Ratio**	**Range**	**Freq.**	**Ratio**	**Range**	**Freq.**	**Ratio**
x-Small	0–31	47	25.1%	3–53	45	24.1%	1–11	40	21.4%
Small	32–297	93	49.7%	54–90	96	51.3%	12–28	100	53.5%
Medium	298–695	30	16.0%	91–144	23	12.3%	29–54	24	12.8%
Large	696–1092	7	3.7%	145–198	7	3.7%	55–79	10	5.3%
x-Large	1093–5419	10	5.3%	199–1903	16	8.6%	80–598	13	7.0%

one variable cloned for each file. Thus, the correlation is little between those two levels.

Table 13
Linux kernel r^2 values for all granularity levels combinations.

Linux kernel	*Variables* × *functions*	*Variables* × *files*	*Functions* × *files*
r^2	0.7	0.1	0.6

5. Related work and threats to validity

5.1. Related work

Many clone detection approaches exist in software engineering literature addressing the identification of both exact and near-miss clones. These approaches can be categorized as text-based (Baker, 1995; Ducasse et al., 1999; Roy and Cordy, 2008b; Johnson, 1994; Nasehi et al., 2007; Johnson, 1993), token-based (Baker, 1995; Sajnani et al., 2016; Kim et al., 2018; Ragkhitwetsagul and Krinke, 2019; Kamiya et al., 2002; Li et al., 2006), tree-based (Baxter et al., 1998; Jiang et al., 2007; Koschke et al., 2006; Yang, 1991), metrics-based (Saini et al., 2018; Kontogiannis et al., 1996; Mayrand et al., 1996; Davey et al., 1995; Casazza et al., 2001), and graph-based (Komondoor and Horwitz, 2001; Gabel et al., 2008; Krinke, 2001; Liu et al., 2006; Chen et al., 2014). More details about these techniques can be found in related surveys (Rattan et al., 2013; Roy et al., 2009; Ragkhitwetsagul et al., 2018; Saini et al., 2018).

Token-based and Text-based approaches have been applied in the literature successfully. Other approaches can operate on different source code abstractions, such as abstract syntax trees (AST) and PDG. In AST-based approaches, similar sub-trees are reported as structural clones. PDG-based approaches identify behavioral clones that exhibit similar data and control flow. Metric-based approaches detect clones at different levels of similarity

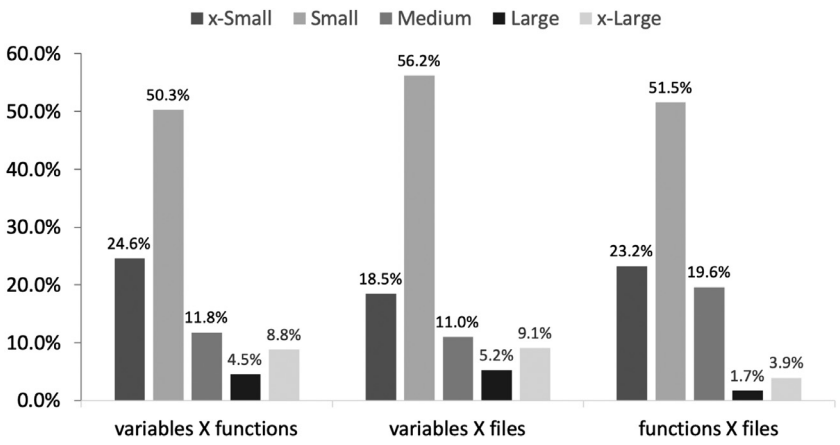

Fig. 7. Histogram the commonality between the granularity levels of clones. This gives the percentage of clones that are common between measure for each size category.

(that is, textual, layout, expression and control flow similarity) or combinations thereof. In work by Marcus and Maletic (2001), they use an information retrieval approach, latent semantic indexing, to detect source code clones that exhibit higher level domain concepts, features or functionality.

When working at the level of source code, software engineers are restricted to text-based and token-based techniques. These approaches are simple and efficient but identify only limited levels of clones. Alternatively, working at a higher level of abstraction such as the AST or PDG seems to be an attractive option for detecting structural and behavioral clones. However, moving to higher levels abstraction requires more complete parsing of the input source code than the more traditional lexical techniques. The capability or robustness of clone detectors may be compromised and thus their performance in dealing with the realistic nature of large software systems, including non-compliable code, code fragments, and others, will be significantly reduced. Moreover, higher abstraction levels may lose valuable information, making it difficult to map results back to the original source code. For example, ASTs can discard line number information and thus cannot be translated back to the programmer's view of source code.

Krinke (2001) created a PDG-based clone detection technique that does not experience any trade-off between recall and precision. They used fine-grained PDGs to detect code fragments as a clone candidates. Since it is a PDG-based approach, it is quite expensive. For example, the *bison* system that we included in our evaluation takes Krinke's technique around one hour to find similar PDGs. We decided not to contrast our work to Krinke in our evaluation because they do not use program slicing.

Xue et al. (2018b) introduce CLONE-SLICER to detect domain-specific clones on binaries instead of at the source code level. They use forward slicing to remove instructions that are pointer-irrelevant. This differs from our work in that they consider binary executables instead of the source code, and they find clones that are pointer-related only. They evaluated their approach using some real-world applications and compare it to their previous clone detector tool CLONE-HUNTER (Xue et al., 2018).

The GPLAG (Liu et al., 2006) tool is graph-based. However, it is designed for a different objective than clone detection and has some corresponding limitations. For more details about these limitations please refer to (Roy et al., 2009).

In graph-based clone detection approaches the detection process of clones is the problem of finding isomorphic sub-graphs in a given PDG. Since this problem is known to be NP hard, related algorithms use approximative solutions that are PDG-based, thus are quite expensive. To find isomorphic sub-graphs, some algorithms used backward program slicing (Weiser, 1984). However, this results in a tradeoff between precision and recall. There are similar (not identical) sub-graphs that cannot be discovered by using only backward slicing (Komondoor and Horwitz, 2001; Gabel et al., 2008). Komondoor and Horwitz (2001) 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 simultaneously, then comparing the resulted sub-graphs. Hamid et al. (2014) conducted a replication of this work to find refactorable semantic clones based on PDG and backward program slicing. Similarly, Gabel et al. (2008) uses forward program slicing (Bergeretti and Carré, 1985) to find semantic PDG sub-graphs, maps these sub-graphs to related structured syntax trees, and finds clones using the DECKARD clone detection approach (Jiang et al., 2007). They used intra-procedural forward slicing to increase the recall by finding different flows of data throughout a given function.

In a previous work by us on clone detection, we proposed a new way of employing slicing to detect code clones (Alomari and Stephan, 2018). We devised the concept of using forward slicing and an MD5 hashing algorithm to detect clones between several versions of software systems. The proposal was to represent slices as strings and provide them as input to an MD5 hash algorithm that produces a 128-bit hash value. Our work here is motivated by that work, but in contrast exploits the embedded slicing information and associated slicing fields to calculate behavioral related slice-based metrics that then use these metrics to generate slicing vectors that represent each slice uniquely.

The idea of the *characteristic vectors* is implemented by DECKARD approach to detect clones as a tree similarity problem. Each vector represents a numerical approximation of a particular subtree. In contrast, we introduce *slicing vectors* to capture the structural information of slices. Each vector in our work has a fixed size that is based on the number of slicing metrics we calculate using SRCCLONE.

Gallagher and Layman (2003) were the first trying to use a backward decompositional slicing that is PDG-based to answer the question *"Are decomposition slices clones?"*. However, they did not reach any conclusions. Instead they presented pros and cons for slice-clones as software clones. The most interested pros and cons they found were (1) slice-clones arise due to failure to identify or use abstract data types, (2) similar slices are not on the same variables, (3) similar sized analysis of slices leads to a straightforward detection of clones, and (4) slice-clones were happened by accident, not programmer's intent. In contrast, our approach uses forward decompositional slicing and it is not PDG-based. This is an important difference as slices identified using

our approach are different than those found by Gallagher and are based on totally different slice definition, and thus the pros and cons are not directly applicable. In the future, we plan to investigate these pros and cons explicitly and more in-depth for our slice-clones.

NiCad (Roy and Cordy, 2008b) and SourcererCC (Sajnani et al., 2016) are often considered to be current state-of-the-art tools in detecting clones up to Type-3 clones. NiCad uses a text-based approach that involving syntactic pretty-printing with a flexible code normalization and filtering. SourcererCC uses hybrid of token- and index-based techniques and targets the first three types of clones. Like NiCad, SourcererCC uses filtering heuristics that are based on token ordering to reduce the size of the index and the number of comparisons between code segments and tokens. SourcererCC's scalability is demonstrated with IJaDataset and BigCloneBench clone benchmark. In comparison to CCFinderX (Kamiya et al., 2002), SourcererCC has better results in detecting Type-3 clones over 100MLOC. However, similar to NiCad, SourcererCC had poor recall for the Moderately Type-3 clones (MT3) and Weakly Type-3 or Type-4 clones (WT3/T4) that have syntactical similarity in [0%–70%). NiCad has a poor execution time for large inputs and has scalability issues at the 100 MLOC input size.

A new metrics-based clone detector, called Oreo (Saini et al., 2018) detects the Type-1 to Type-3 clones and is also capable to detect MT3 and WT3/T4 harder-to-detect clones. This approach combines machine learning, information retrieval, and software metrics. They use clone pairs produced by SourcererCC to train their metrics similarity model. Therefore, Oreo is affected by the detection capability of SourcererCC and inherits some corresponding limitations. For example, using function granularity as a code blocks for clones. However, it still detects 1 to 2 orders of magnitude more harder-to-detect clone pairs than SourcererCC and NiCad. The authors of Oreo expressed their interest in exploring the impact of training their model using a different clone detector and at a finer granularities than functions, which they believe will enhance the detection quality. This is what we implemented in our slice-based detector, as it is able to find clones at three levels of granularities, from variable to file.

In the context of searching for clones in online sources, special types of code clone search techniques are needed. These techniques accepts a code fragment as a query then perform a code-to-code search in large-scale code bases. For example, FaCoY (Kim et al., 2018) is a token-based and semantic-based technique that employs the notion of *query alternation* in the process of detecting clones in software. FaCoY fully relies on source code (static) and implements a *query alternation strategy*. It uses other tokens that may be related to the functional behavior of the input code instead of directly matching code query tokens with the source code in the search space. While FaCoY is a relatively scalable clone detection and clone search technique, finding Type-3 clones in a scalable fashion is still an open challenge. A related scalable, and incremental code clone search technique, is Siamese (Ragkhitwetsagul and Krinke, 2019). This technique incorporates multiple source code representations, query reduction, and a customized ranking function to improve the performance of the clone search process. Similar to srcClone, Siamese is resilient to incomplete or uncompilable code fragments. Siamese and FaCoY are not clone detector tools as much as they are clone search tools. Thus, these tools do not report a set of clones that can be used to measure recall and precision (Ragkhitwetsagul and Krinke, 2019). However, these tools can be evaluated for recall by issuing multiple queries and evaluated the returned ranked results. For example, they have chosen a specific number of methods that can be used as the queries. Those selected methods do not include all the methods included in BigCloneBench and the authors claimed that this does not affect the clone recall.

5.2. Threats to validity

Our measurements of srcClone's precision could be affected by two factors: (1) the personal subjectivity of the judge, human bias, and (2) the underlying lightweight slicer we used to extract the slicing information. The first factor can be reduced by involving more judges and increasing the size of the randomly selected sample to investigate. For the second factor, because of our lightweight approach, there are some limitations to our use of srcSlice. Specifically, srcSlice is not fully supporting the control flow and is flow insensitive. For example, a variable is not considered to be used when in the same statement it is also defined. While our approach addresses the simpler control flow dependencies and def/use situations, we do not address the more complex cases as they are beyond the scope of srcSlice tool and its (and our) goal of efficiency versus accuracy. We manually checked a random sample of the generated slices and we found the slices are akin to what is carried by a more heavyweight slicers. We do not foresee srcSlice as a lightweight slicer being a replacement for heavyweight slicers, rather, we use our approach to judge if it is prudent to use slicing in clone detection problem. The difference in run time allows decisions to be made quickly. We feel we open the door for researchers to leverage program slicing to address the cloning problem especially when we believe srcSlice produces reliable accuracy given its speed. Also, we are not aware of any publicly available slicer that can scale to MLOC like the BigCloneBench data set. For competing tools, we measured the precision and compared it to srcClone. Also, we have used some of their published precision values in cases where we doubted the measured results.

This study is the same as other clone detection studies, in that it may be affected by the tools' configurations. However, for each competing tool we tried to use the "optimal" configurations as suggested by the authors. The BigCloneEval framework provided a great help and support to accomplish this.

6. Conclusion

We have described our approach and initial results for detecting source code clones employing program slicing, srcClone. We contend this is a new perspective on semantic clone detection that enables large-scale cloning analysis using slice-based clones. We leverage our scalable and lightweight slicing tool, srcSlice, to compute the required slicing information. While srcClone may not be as accurate as approaches that generate complete SDGs or PDGs, it is very scalable and allows for inter-procedural analysis. It affords reliable accuracy, especially in consideration of its strength of speed and being a lightweight approach.

We contrast srcClone with two PDG-based approaches that use program slicing. We additionally evaluate srcClone alongside six state-of-the-art tools using BigCloneBench. Our evaluation demonstrates that srcClone is practical and scales to millions of lines of code. Our approach performs on par or better than other detection tools, and is capable of producing clones that are related semantically for a given variable, function, or file. This includes clones that are difficult, if not impossible, for leading tools to detect.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Allen, F.E., 1970. Control flow analysis. ACM Sigplan Notices 5 (7), 1–19.

Alomari, H.W., Collard, M.L., Maletic, J.I., 2012. A very efficient and scalable forward static slicing approach. In: 2012 19th Working Conference on Reverse Engineering. IEEE, Kingston, ON, Canada, pp. 425–434.

Alomari, H.W., Collard, M.L., Maletic, J.I., 2014a. A slice-based estimation approach for maintenance effort. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 81–90.

Alomari, H.W., Collard, M.L., Maletic, J.I., Alhindawi, N., Meqdadi, O., 2014b. SrcSlice: Very efficient and scalable forward static slicing. J. Software: Evolut. Proc 26 (11), 931–961.

Alomari, H.W., Jennings, R.A., De Souza, P.V., Stephan, M., Gannod, G.C., 2016. VizSlice: Visualizing large scale software slices. In: 2016 IEEE Working Conference on Software Visualization (VISSOFT). IEEE, pp. 101–105.

Alomari, H.W., Stephan, M., 2018. Towards slice-based semantic clone detection. In: 2018 IEEE 12th International Workshop on Software Clones (IWSC). IEEE, Campobasso, Italy, pp. 58–59.

Alomari, H.W., Stephan, M., 2020. Srcclone: detecting code clones via decompositional slicing. In: Proceedings of the 28th International Conference on Program Comprehension. IEEE/ACM, pp. 274–284.

Alqadi, B., 2019. The relationship between cognitive complexity and the probability of defects. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 600–604.

Alqadi, B.S., Maletic, J.I., 2020. Slice-based cognitive complexity metrics for defect prediction. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 411–422.

Antoniol, G., Casazza, G., Di Penta, M., Merlo, E., 2001. Modeling clones evolution through time series. In: Proceedings IEEE International Conference on Software Maintenance. ICSM 2001. IEEE, Florence, Italy, pp. 273–280.

Antoniol, G., Villano, U., Merlo, E., Di Penta, M., 2002. Analyzing cloning evolution in the Linux kernel. Inf. Softw. Technol. 44 (13), 755–765.

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

Basit, H.A., Jarzabek, S., 2007. Efficient token based clone detection with flexible tokenization. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM, Dubrovnik, Croatia, pp. 513–516.

Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L., 1998. Clone detection using abstract syntax trees. In: Proceedings. International Conference on Software Maintenance. IEEE, Bethesda, MD, USA, pp. 368–377.

Bellon, S., 2002. Vergleich Von Techniken Zur Erkennung Duplizierten Quellcodes. (Master's Thesis), Institut Fur Softwaretechnologie, Universitat Stuttgart, Stuttgart, Germany.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E., 2007. Comparison and evaluation of clone detection tools. IEEE Trans. Softw. Eng. 33 (9), 577–591.

Bergeretti, J.-F., Carré, B.A., 1985. Information-flow and data-flow analysis of while-programs. ACM Trans. Progr. Lang. Syst (TOPLAS) 7 (1), 37–61.

Binkley, D.W., Harman, M., 2004. A survey of empirical results on program slicing. Adv Comput 62 (105178), 105–178.

Borrelli, A., Nardone, V., Di Lucca, G.A., Canfora, G., Di Penta, M., 2020. Detecting video game-specific bad smells in unity projects. In: 2020 Mining Software Repositories, October 5–6, 2020, Seoul, Republic of Korea (MSR), pp. 11.

Bulychev, P., Minea, M., 2008. Duplicate code detection using anti-unification. In: Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering, p. 4.

Casazza, G., Antoniol, G., Villano, U., Merlo, E., Di Penta, M., 2001. Identifying clones in the linux kernel. In: Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation. IEEE, pp. 90–97.

Chen, K., Liu, P., Zhang, Y., 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In: Proceedings of the 36th International Conference on Software Engineering, pp. 175–186.

Collard, M.L., Decker, M.J., Maletic, J.I., 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, pp. 173–184.

Collard, M.L., Decker, M.J., Maletic, J.I., 2013. SrcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In: 2013 IEEE International Conference on Software Maintenance. IEEE, pp. 516–519.

Collard, M.L., Maletic, J.I., Robinson, B.P., 2010. A lightweight transformational approach to support large scale adaptive changes. In: 2010 IEEE International Conference on Software Maintenance. IEEE, pp. 1–10.

Dagenais, M., Merlo, E., Laguë, B., Proulx, D., 1998. Clones occurrence in large object oriented software packages. In: Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research. IBM Press, Canada, p. 10.

Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S., 2004. Locality-sensitive hashing scheme based on P-stable distributions. In: Proceedings of the Twentieth Annual Symposium on Computational Geometry. ACM, pp. 253–262.

Davey, N., Barson, P., Field, S., Frank, R., Tansley, D., 1995. The development of a software clone detector. Int. J. Appl. Softw. Tech.

Dragan, N., Collard, M.L., Maletic, J.I., 2006. Reverse engineering method stereotypes. In: 2006 22nd IEEE International Conference on Software Maintenance. IEEE, pp. 24–34.

Ducasse, S., Rieger, M., Demeyer, S., 1999. A language independent approach for detecting duplicated code. In: International Conference on Software Maintenance. IEEE, Oxford, England, UK, pp. 109–118.

Evans, W.S., Fraser, C.W., Ma, F., 2009. Clone detection via structural abstraction. Softw. Qual. J. 17 (4), 309–330.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Progr. Languag. Syst (TOPLAS) 9 (3), 319–349.

Gabel, M., Jiang, L., Su, Z., 2008. Scalable detection of semantic clones. In: Proceedings of the 30th International Conference on Software Engineering. ACM, Leipzig, Germany, pp. 321–330.

Gallagher, K., Layman, L., 2003. Are decomposition slices clones? In: 11th IEEE International Workshop on Program Comprehension, 2003.. IEEE, Portland, OR, USA, pp. 251–256.

Göde, N., Koschke, R., 2013. Studying clone evolution using incremental clone detection. J. Softw: Evolu. Proc 25 (2), 165–192.

Hamid, A., Zaytsev, V., et al., 2014. Detecting refactorable clones by slicing program dependence graphs. In: SATToSE. SATToSE 2014, Italy, pp. 37–48.

Harris, S., Simian - Similarity Analyser, URL https://www.harukizaemon.com/simian/. (Accessed January 14, 2020).

Indyk, P., Motwani, R., 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, pp. 604–613.

Jiang, L., Misherghi, G., Su, Z., Glondu, S., 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, pp. 96–105.

Johnson, J.H., 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, pp. 171–183.

Johnson, J.H., 1994. Substring matching for clone detection and change tracking. In: ICSM, vol. 94, BC, Canada, pp. 120–126.

Johnson, J.H., 1994b. Visualizing textual redundancy in legacy source. In: Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research. IBM Press, p. 32.

Juillerat, N., Hirsbrunner, B., 2006. An algorithm for detecting and removing clones in java code. In: Proceedings of the 3rd Workshop on Software Evolution Through Transformations: Embracing the Change, SeTra, Vol. 2006, pp. 63–74.

Kamiya, T., Kusumoto, S., Inoue, K., 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28 (7), 654–670.

Kapser, C.J., Godfrey, M.W., 2008. "Cloning considered harmful" considered harmful: Patterns of cloning in software. Empir. Softw. Eng. 13 (6), 645.

Kim, K., Kim, D., Bissyandé, T.F., Choi, E., Li, L., Klein, J., Traon, Y.L., 2018. FaCoY: A code-to-code search engine. In: Proceedings of the 40th International Conference on Software Engineering, pp. 946–957.

Klett, J., 1972. Applied Multivariate Analysis. McGraw-Hill, New York.

Komondoor, R., Horwitz, S., 2001. Using slicing to identify duplication in source code. In: International Static Analysis Symposium. Springer, Berlin, Heidelberg, pp. 40–56.

Kontogiannis, K., 1997. Evaluation experiments on the detection of programming patterns using software metrics. In: Proceedings of the Fourth Working Conference on Reverse Engineering. IEEE, Amsterdam, Netherlands, pp. 44–54.

Kontogiannis, K.A., DeMori, R., Merlo, E., Galler, M., Bernstein, M., 1996. Pattern matching for clone and concept detection. Autom. Softw. Eng 3 (1–2), 77–108.

Koschke, R., Falke, R., Frenzel, P., 2006. Clone detection using abstract syntax suffix trees. In: 2006 13th Working Conference on Reverse Engineering. IEEE, Benevento, Italy, pp. 253–262.

Kraft, N.A., Bonds, B.W., Smith, R.K., 2008. Cross-language clone detection. In: SEKE, 54–59.

Krinke, J., 2001. Identifying similar code with program dependence graphs. In: Proceedings Eighth Working Conference on Reverse Engineering. IEEE, Stuttgart, Germany, pp. 301–309.

Landwerth, I., Clone Detective for Visual Studio, URL https://marketplace.visualstudio.com/items?itemName=ImmoLandwerthMSFT.CloneDetectiveforVisualStudio. (Accessed January 14, 2020).

Lee, S., Jeong, I., 2005. SDD: High performance code clone detection system for large scale source code. In: Companion To the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM, CA, San Diego, USA, pp. 140–141.

Leitão, A.M., 2004. Detection of redundant code using R2D2. Softw. Qual. J. 12 (4), 361–382.

Li, Z., Lu, S., Myagmar, S., Zhou, Y., 2006. CP-MIner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. Softw. Eng. 32 (3), 176–192.
Liu, C., Chen, C., Han, J., Yu, P.S., 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, pp. 872–881.
Manber, U., et al., 1994. Finding similar files in a large file system. In: Usenix Winter. 94, Usenix Winter, pp. 1–10.
Marcus, A., Maletic, J.I., 2001. Identification of high-level concept clones in source code. In: Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001). IEEE, San Diego, CA, USA, pp. 107–114.
Mayrand, J., Leblanc, C., Merlo, E., 1996. Experiment on the automatic detection of function clones in a software system using metrics. In: Icsm. 96, IEEE, Monterey, CA, USA, p. 244.
Merlo, E., Antoniol, G., Di Penta, M., Rollo, V.F., 2004. Linear complexity object-oriented similarity for clone detection and software evolution analyses. In: 20th IEEE International Conference on Software Maintenance, 2004 Proceedings. IEEE, Chicago, IL, USA, pp. 412–416.
Merlo, E., Dagenais, M., Bachand, P., Sormani, J., Gradara, S., Antoniol, G., 2002. Investigating large software system evolution: the Linux kernel. In: Proceedings 26th Annual International Computer Software and Applications. IEEE, Oxford, UK, pp. 421–426.
Motwani, R., Naor, A., Panigrahi, R., 2006. Lower bounds on locality sensitive hashing. In: Proceedings of the Twenty-Second Annual Symposium on Computational Geometry, pp. 154–157.
Nasehi, S.M., Sotudeh, G.R., Gomrokchi, M., 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, pp. 192–197.
Newman, C.D., Sage, T., Collard, M.L., Alomari, H.W., Maletic, J.I., 2016. SrcSlice: A tool for efficient static forward slicing. In: International Conference on Software Engineering Companion. IEEE, Austin, TX, USA, pp. 621–624.
Patenaude, J.-F., Merlo, E., Dagenais, M., Laguë, B., 1999. Extending software quality assessment techniques to java systems. In: Proceedings Seventh International Workshop on Program Comprehension. IEEE, Pittsburgh, PA, USA, pp. 49–56.
Project, P.O.S., PMD Source Code Analyzer Project, URL https://pmd.github.io/latest/pmd_userdocs_cpd.html. (Accessed January 14, 2020).
Ragkhitwetsagul, C., Krinke, J., 2019. Siamese: Scalable and incremental code clone search via multiple code representations. Empir. Softw. Eng. 24 (4), 2236–2284.
Ragkhitwetsagul, C., Krinke, J., Clark, D., 2018. A comparison of code similarity analysers. Empir. Softw. Eng. 23 (4), 2464–2519.
Rajaraman, A., Ullman, J.D., 2011. Mining of Massive Datasets. Cambridge University Press.
Rattan, D., Bhatia, R., Singh, M., 2013. Software clone detection: A systematic review. Inf. Softw. Technol. 55 (7), 1165–1199.
Rivest, R., Dusse, S., 1992. The MD5 Message-digest Algorithm. MIT Laboratory for Computer Science.
Roy, C.K., Cordy, J.R., 2007. A survey on software clone detection research. Queen's School of Comput. TR 541 (115), 64–68.
Roy, C.K., Cordy, J.R., 2008a. An empirical study of function clones in open source software. In: 2008 15th Working Conference on Reverse Engineering. IEEE, Antwerp, Belgium, pp. 81–90.
Roy, C.K., Cordy, J.R., 2008b. 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, Netherlands, pp. 172–181.
Roy, C.K., Cordy, J.R., 2008c. Scenario-based comparison of clone detection techniques. In: 2008 16th IEEE International Conference on Program Comprehension. IEEE, Amsterdam, Netherlands, pp. 153–162.
Roy, C.K., Cordy, J.R., Koschke, R., 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Progr 74 (7), 470–495.
Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P., Lopes, C.V., 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, pp. 354–365.
Sajnani, H., 2016. Large-scale code clone detection (Ph.D. thesis). UC Irvine.
Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., Lopes, C.V., 2016. Sourcerercc: Scaling code clone detection to big-code. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, Austin, TX, USA, pp. 1157–1168.
Sheneamer, A., Kalita, J., 2016. A survey of software clone detection techniques. Int. J. Comput. Appl. 137 (10), 1–21.
Silva, J., 2012. A vocabulary of program slicing-based techniques. ACM Comput. Surv. 44 (3), 12.
SimScan, SimScan Duplicate Code Detection Tool, URL http://cc.embarcadero.com/Item/19813. (Accessed January 14, 2020).
Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K., Mia, M.M., 2014. Towards a big data curated benchmark of inter-project code clones. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, Victoria, BC, Canada, pp. 476–480.
Svajlenko, J., Roy, C.K., 2014. Evaluating modern clone detection tools. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, Victoria, BC, Canada, pp. 321–330.
Svajlenko, J., Roy, C.K., 2015. Evaluating clone detection tools with BigCloneBench. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, Bremen, Germany, pp. 131–140.
Svajlenko, J., Roy, C.K., 2016. BigCloneEval: A clone detection tool evaluation framework with BigCloneBench. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 596–600.
Svajlenko, J., Roy, C.K., Cordy, J.R., 2013. A mutation analysis based benchmarking framework for clone detectors. In: 2013 7th International Workshop on Software Clones (IWSC). IEEE, pp. 8–9.
Tairas, R., Gray, J., 2006. Phoenix-based clone detection using suffix trees. In: Proceedings of the 44th Annual Southeast Regional Conference. ACM, Florida, Melbourne, pp. 679–684.
Tip, F., 1994. a Survey of Program Slicing Techniques. Centrum voor Wiskunde en Informatica Amsterdam.
Ueda, Y., Kamiya, T., Kusumoto, S., Inoue, K., 2002. On detection of gapped code clones using gap locations. In: Ninth Asia-Pacific Software Engineering Conference. IEEE, pp. 327–336.
Wahler, V., Seipel, D., Wolff, J., Fischer, G., 2004. Clone detection in source code by frequent itemset techniques. In: Source Code Analysis and Manipulation, Fourth IEEE International Workshop on. IEEE, Chicago, IL, USA, pp. 128–135.
Weiser, M., 1984. Program slicing. IEEE Trans. Softw. Eng. (4), 352–357.
Wettel, R., Marinescu, R., 2005. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In: Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05). IEEE, Timisoara, Romania, pp. 8–pp.
Xue, H., Venkataramani, G., Lan, T., 2018. Clone-hunter: Accelerated bound checks elimination via binary code clone detection. In: International Workshop on Machine Learning and Programming Languages, pp. 11–19.
Xue, H., Venkataramani, G., Lan, T., 2018b. 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, Canada, pp. 27–33.
Yamashina, T., Uwano, H., Fushida, K., Kamei, Y., Nagura, M., Kawaguchi, S., Iida, H., 2008. SHINOBI: A real-time code clone detection tool for software maintenance. Nara Inst. Sci. Tech 26.
Yang, W., 1991. Identifying syntactic differences between two programs. Softw. - Pract. Exp. 21 (7), 739–755.

Hakam Alomari is an Assistant Professor in the Department of Computer Science and Software Engineering at Miami University in Ohio, USA. He received his Ph.D. in Computer Science from Kent State University in Ohio, USA in 2012. His main focus is lightweight software analysis for software engineering usages. The objective is to develop new analysis methods that are highly scalable for application on very large software systems. His current research interests include program slicing and analysis, clone detection, software visualization, and software testing.

Matthew Stephan is an Assistant Professor in the Department of Computer Science and Software Engineering at Miami University in Ohio, USA. He received his Ph.D. in Computer Science from Queen's University in Kingston, Ontario, Canada after earning his Master's and Bachelor's from the University of Waterloo, also in Ontario, Canada. He has over 30 publications on a variety of Software Engineering topics including software clones, model-driven engineering, and domain specific languages. He has been funded by the National Science Foundation (USA) and The Natural Sciences and Engineering Research Council of Canada.