srcSlice: A Tool for Efficient Static Forward Slicing

Christian D. Newman[1], Tessandra Sage[1], Michael L. Collard[2], Hakam W. Alomari[3], Jonathan I. Maletic[1]

[1]Department of Computer Science
Kent State University
Kent, Ohio USA
{cnewman, jmaletic}@kent.edu

[2]Department of Computer Science
The University of Akron
Akron, Ohio USA
collard@uakron.edu

[3]Department of Computer Science and Software Engineering
Miami University
Oxford, Ohio, USA
alomarhw@miamioh.edu

***Abstract*— An efficient lightweight forward static slicing tool is presented. The tool is implemented on top of srcML, an XML representation of source code. The approach does not compute the full program dependence graph but instead dependency information is computed as needed while computing the slice on a variable. The result is a list of line numbers, dependent variables, aliases, and function calls that are part of the slice for a given variable. The tool produces the slice for all variables in a system. The approach is highly scalable and can generate the slices for all variables of the Linux kernel in less than 15 minutes. A demonstration video is at: https://youtu.be/McvFUVSGg-g**

***Index Terms*—static forward program slicing, srcML**

I. Introduction

Program slicing is a widely used method for understanding and detecting the impact of changes to software. The idea is fairly simple; given a variable and the location of that variable in a program, tell me what other parts of the program are affected by this variable. Unfortunately, there are not any open-source slicing tools available that are scalable to large systems. While some free versions of commercial tools are available for academic use they are limited on the size of code base that can be analyzed. For example, the free version of CodeSurfer will not slice programs over 200KLOC.

The concept of program slicing was originally identified by Weiser [16, 17] as a debugging aid. He defined the slice as an executable program that preserved the behavior of the original program. Weiser's algorithm traces the data and control dependencies by solving data-flow equations for determining the direct and indirect relevant variables and statements. Since that time, a large number of different slicing techniques have been proposed and implemented. These techniques are broadly distinguished according to the type of slices; static versus dynamic [14, 18], closure versus executable [18], inter-procedural versus intra-procedural [6, 8], and forward versus backward [9, 18].

Program slicing is typically based on the notion of a Program Dependence Graph (PDG) [12] or one of its variants, e.g., a System Dependence Graph (SDG) [10]. Unfortunately, building the PDG/SDG is quite costly in terms of computational time and space. As such, slicing approaches generally do not scale well and while there are some (costly) workarounds, generating slices for a very large system can often take days of computing time. Additionally, many tools are strictly limited to an upper bound on the size of the program they can slice.

The tool presented here, srcSlice, addresses this limitation by eliminating the time and effort needed to build the entire PDG. In short, it combines a text-based approach, similar to Cordy [5], with a lightweight static analysis infrastructure that only computes dependence information as needed (aka on-the-fly) while computing the slice for each variable in the program. The slicing process is performed using the srcML [3, 4] format for source code. The srcML format provides direct access to abstract syntactic information to support static analysis. While this lightweight approach will generally never match the completeness of generating a PDG/SDG and performing deep pointer analysis, it does provide an accurate approximation of a program slice in an extremely short time. We found up to four orders of magnitude increase in speed for large systems compared to a full PDG approach [1, 2].

A very fast and scalable, yet slightly less accurate, slicing tool is useful for a number of reasons. Developers can estimate the impact of a change to a large system within minutes versus hours/days. This is very important for planning the implementation of new features and understanding how a change is related to other parts of the system. Additionally, we feel a fast slicing approach could open up new avenues of research in metrics and mining of histories based on slicing. That is, slicing can now be conducted on very large systems and on entire version histories in very practical time frames. This opens the door to a number of experiments and empirical investigations previously too costly to undertake.

II. The srcSlice Tool

The srcSlice tool implements a forward, static slicing technique. Complete details of the slicing algorithm used is given in [1, 2]. The tool is enabled by the srcML [3, 4] infrastructure. srcML augments source code with abstract syntactic information. This syntactic information is used to identify program dependencies as needed when computing the slice.

Given a system (in the srcML format), srcSlice gathers data about every file, function, and variable throughout the system; storing it all in a three-tier dictionary. Unlike other slicing techniques, it does not rely fully on pre-computed data and control dependencies. Instead, as the code is analyzed, srcSlice selectively computes what is required while keeping track of just enough context to determine the dependencies. Because of this, srcSlice is very memory efficient, with its dominating memory footprint coming from storing data in the dictionary.

A slice profile for all variables is computed line by line as variables are encountered. After all slice profiles are computed, a single pass through all the profiles is done to take into account dependent variables, function calls, control-flow edges, and direct pointer aliasing to generate the final slices.

a.	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<<std::endl;
	16.	std::cout<<fun(i);
	17.	abc=abc+i;
	18.	}
b.	*Slice Profile(abc): def={10,17}, use={1,2,5,6,13,15} slines={1,2,5,7,10,13,15,17}, dvars={abc}, pointers={}, cfuncs={fun{1},foo{1}}*	
	Slice Profile(i): def={11}, use={1,2,5,7,12,13,15,16,17}, slines={1,2,5,7,11,12,13,15,16,17}, dvars={abc}, pointers={}, cfuncs={fun{1}, foo{2}}	
	Slice Profile(z): def={1}, use={2}, slines={1,2}, dvars={}, pointers={}, cfuncs={}	
	Slice Profile(y): def={5}, use={7}, slines={5,7}, dvars={}, pointers={i}, cfuncs={}	
	Slice Profile(x): def={5}, use={1,2,6}, slines={1,2,5,6}, dvars={}, pointers={abc}, cfuncs={fun{1}}	

Figure 1 (a) Sample source code, (b) System dictionary with all slice profiles for the source code in (a).

A. Slice Profile

The slice profile for an identifier contains all data gathered about that identifier during the slicing process. The following is a list of that information:

- *File*, *function*, and *type*, *variable* names – what file/function the variable is in and what type.
- *Def* – what line a variable was defined or redefined on (this can be declaration, definition, or assignment. Def keeps track of when a variable's value changes). Def is used to differentiate between variables with the same name but in differing scopes.
- *Use* – what line a variable is used on. This refers to a variable's value being used in some computation with no modification to the variable's value. The combination of def and use can be used to construct def-use chains.
- *Slines* – all lines that a variable was defined or used on. This is the union between def and use.
- *Cfunctions* - a list of functions called using the slicing variable.
- *Dvariables* - a list of variables that are data dependent on the slice variable.
- *Pointers* - a list of aliases of the slicing variable. The elements of this list are variables to which the slicing variable is a pointer. We are unable to do full pointer resolution, however, quite a bit can be statically calculated.

An example of a slice computed by srcSlice on a small program is given in Figure 1. The first portion of the figure (a) presents a small program constructed to show how srcSlice computes the profile. The second part of the figure (b) is the slice profile for the program in (a).

We see that the variable i is defined at line 11 and has uses in the 'main', 'fun' and 'foo' functions. That is, an interprocedural slice is computed. The only variable dependent on i is the variable abc at line 17. Simple pointer analysis is also done as can be seen in the profiles for variables y and x.

B. Output

srcSlice produces a system dictionary of all the slice profiles of all variables. It is three-tiered and consists of three maps. On the first level is a map from files to functions, on the second level is a map from functions to variable names, and on the third level is a map from variable names to slice profiles.

Figure 2 is an example of srcSlice's output. In the future, we plan to support other formats and allow for the user to query the internal dictionary constructed by srcSlice directly. For now, srcSlice provides an easy-to-parse format.

```
file,fcn/mthd name,slice variable,def{1,2,…,n},
use{1,2,…,n},dvars{a,b,…,z},pointers{ptra,ptrb,…,ptr
z},cfuncs{func1{1,2,…,n},func2{1,2,…,n}}
```

Figure 2 The output format of srcSlice.

If we insert the slice profiles from the code in Figure 1 into the format described above, we get the following:

```
srcslicetest.cpp,main,i,def{11},use{1,2,5,
7,12,13,15,16,17},dvars{abc},pointers{},cf
uncs{fun{1},foo{2}}
srcslicetest.cpp,main,abc,def{10,17},use{1
,2,5,6,13,15},dvars{},pointers{},cfuncs{fu
n{1},foo{1}}
srcslicetest.cpp,fun,z,def{1},use{2},dvars
{},pointers{},cfuncs{}
srcslicetest.cpp,foo,y,def{5},use{7},dvars
{},pointers{i},cfuncs{}
srcslicetest.cpp,foo,x,def{5},use{1,2,6},d
vars{},pointers{abc},cfuncs{fun{1}}
```

Notice that, due to the non-nested nature of the output format, file names will be repeated for every function and variable they contain and function names will be repeated for every variable they contain; this is seen in the fact that srcslicetest.cpp is repeated for every entry. A typical use case of srcSlice is to see what areas a modification to a variable may impact. For example, a name change, type change, value change, etc. Using the output from srcSlice, the user needs only parse it into a data structure and then check the def/use sets for each that variable they are interested. This informs them of exactly what lines in the system will potentially see some change in behavior due to a modification of the variable.

TABLE I. RESULTS OF SRCSLICE APPLIED TO MULTIPLE SYSTEMS WITH THE SIZE OF THE SYSTEM, THE NUMBER OF VARIABLES FOUND, AND THE EXECUTION TIME FOR THE SRCSLICE TOOL.

System	Size (LOC)	Variables	Execution Time
Linux Kernel-4.06	~13,000,000	~1,918,000	7 min
Blender-2.68	~1,300,000	~265,000	70 sec
Inkscape-0-.91	~410,000	~74,000	18 sec

C. Computing the slice

Given the information computed in the slice profile, we now explain how the final slice is calculated. The slice of a variable is more than just the impact it has on the scope to which it is local. For this reason, srcSlice uses data contained in Cfunctions and Pointers to calculate indirect and inter-procedural slice information. To do this, srcSlice records whether a given variable is an alias. When this variable is used, srcSlice determines what it most recently recorded as aliasing (the most recent variable the pointer is pointed to) and then updates the slice profile for the variable being pointed at (so use, def, Cfunctions, etc., are all updated to reflect usages through the alias). In this way, it unions together slice profiles for alias variables and the variables the alias is pointing. Our approach is conservative about handling aliases and only unions when absolutely certain.

In much the same way, inter-procedural slice information is gathered by using the list of functions called using the slicing variable (Cfunctions). When a variable is used as an argument to a call, srcSlice records the name of function called and the position of the variable in the argument list. We use data about the functions gathered by srcSlice to then compute how the value of the called variable is used within the call. The slice profile is updated with data about uses within the called function. Once all of this is complete, a more detailed view of where each variable is defined/modified and used is obtained.

It is worth noting that the def and use containers allow srcSlice to compute which definition of a variable corresponds to which uses of a variable through merely comparing line numbers. This means that def-use chains can easily be computed by merely comparing which lines in the use-vector fall between lines in the def-vector. While srcSlice does not explicitly compute these on its own, computing them given the information srcSlice provides is quite straightforward.

III. IMPLEMENTATION & PERFORMANCE

srcSlice uses the srcML format and is implemented as a SAX parser. The particular implementation of SAX parser is a C++ wrapper around libxml2's SAX interface called srcSAX; it was made specifically to support building tools that use srcML. Since SAX parsers store no data about previously seen tags, they are very memory and operation efficient. To take full advantage of this feature of SAX, we have worked to store as little data as possible and minimize repetition. As a result, srcSlice is very fast. On the largest system we present here (see Table I), the Linux Kernel, srcSlice clocks in at about 274K identifiers per minute. On Blender, srcSlice ran at 240K identifiers per minute. However, the Linux Kernel had about 7 times the number of identifiers as Blender, so there is not quite a 1:1 ratio between the number of variables and the size increase of code. Despite this, the ratio between the speed of execution for srcSlice on Linux and Blender is negligible. Even accounting for a higher number of identifiers per line on Blender, srcSlice scales extremely well and we continue to look for ways to increase srcSlice's speed.

IV. COMPARISON TO CODESURFER

In a previous study [1, 2] srcSlice's accuracy is compared with CodeSurfer, a heavyweight slicing tool from GrammaTech (www.grammatech.com). The results of this paper show that srcSlice has reasonable accuracy given its speed and relatively lightweight approach to slicing. We will summarize the results in this paper and encourage interested readers to examine the full paper for more information.

TABLE II. INTERSECTED SLICE OVER 13 FILES FROM ENSCRIPT-1.6.5, WHERE (%) IN THE CODESURFER (CS) AND SRCSLICE (SS) COLUMNS IS THE SLICE SIZE RELATIVE TO LOC, (%) IN THE INTERSECTION COLUMN IS THE INTERSECTED SLICE RELATIVE TO BOTH TOOLS SLICE SIZE, (SM) IS THE RELATIVE SAFETY MARGIN FOR A SLICE.

enscript-1.6.5	Size		Slice Size						Intersection		
	LOC	SLOC	CodeSurfer (CS)			srcSlice (sS)			Lines	CS	sS
File Name			Lines	%	SM	Lines	%	SM		%	%
src/psgen.c	2860	1993	1351	67.8	1.75	863	43.3	1.12	771	57.1	89.3
src/util.c	2156	1623	1227	75.6	1.48	853	52.6	1.03	827	67.4	97.0
src/main.c	2660	1406	1178	83.8	1.59	768	54.6	1.04	739	62.7	96.2
src/mkafmmap.c	250	153	92	60.1	2.04	45	29.4	1.00	45	48.9	100.0
afmlib/strhash.c	386	268	145	54.1	1.36	145	54.1	1.36	107	73.8	73.8
afmlib/afmparse.c	1017	759	636	83.8	2.05	313	41.2	1.01	310	48.7	99.0
states/ex.c	2378	1536	813	52.9	3.35	279	18.2	1.15	243	29.9	87.1
states/gram.c	2408	1607	433	26.9	2.41	301	18.7	1.67	180	41.6	59.8
afmlib/afm.c	824	590	468	79.3	1.31	357	60.5	1.00	357	76.3	100.0
afmlib/afmtest.c	184	113	67	59.3	1.60	42	37.2	1.00	42	62.7	100.0
afmlib/deffont.c	379	323	311	96.3	8.18	38	11.8	1.00	38	12.2	100.0
afmlib/e_88594.c	284	261	190	72.8		0	0.0		0	0.0	0.0
afmlib/e_mac.c	284	261	219	83.9		0	0.0		0	0.0	0.0
Total	**16070**	**10893**	**7130**			**4004**			**3659**		
Average	**1236**	**838**	**548**	**69.0**	**2.47**	**308**	**32.4**	**1.13**	**281**	**52.8**	**91.1**
Min	**184**	**113**	**67**	**26.9**	**1.31**	**0**	**11.8**	**1**	**0**	**12.2**	**59.8**
Max	**2860**	**1993**	**1351**	**96.3**	**8.18**	**863**	**60.5**	**1.67**	**827**	**76.3**	**100**

Table II presents the results of running CodeSurfer and srcSlice. It presents the size of all systems from the perspective of both LOC and SLOC in the size column, then in the Slice Size column it presents the number of lines in each slice, the line size relative to LOC (%), and the safety margin (SM), which is the size of the resultant slice divided by the size of the intersected slice. CodeSurfer produced a maximum SM of 8.18% and a minimum of 1.31%. srcSlice produced a maximum of 1.67% and a minimum of 1%. The slice size produced by srcSlice is consistently closer to the intersected

slice. Thus srcSlice found the smallest slice that both slicers agreed on as being a proper slice of the given systems.

V. Using srcSlice

The srcSlice tool is a command-line tool that is run on the srcML format of a source code file(s). There is a bit of extra data that srcSlice requires from srcML, however. By default, srcML does not include the position of names in its output. In order for srcSlice to work, we need line positions. To turn line positions on in srcML, simply use the position option:

```
srcml file.cpp --position
```

To run srcSlice on the Linux kernel, the command is:

```
srcslice linux-4.0.6.cpp.xml > slices.dict
```

where linux-4.0.6.cpp.xml is the srcML format of the Linux Kernel and slices.dict is the desired output file. By default, srcSlice outputs to stdout. You can download srcML and srcSlice from www.srcML.org. Instructions for installing along with documentation are also available at this site.

VI. Related Work

We focus on slicing approaches directly related to srcSlice. We refer readers interested in PDG-based slicing approaches a number of surveys on [13, 14, 18]. Gallagher et al [7] proposed the definition of decomposition slicing as a maintenance aid in order to capture all the computation related to a given slicing variable. The decomposition-slicing definition is used by Tonella [15] to construct a concept lattice of decomposition slices. A lightweight-slicing approach for object-oriented programs using dynamic and static analysis, called dependence-cache slicing, is proposed in [11]. In the context of maintaining large-scale systems, another lightweight maintenance tool, called *TuringTool*, was proposed by Cordy et al [5] and was designed to support several maintenance.

srcSlice is distinguished from this work in multiple ways. The method used is not PDG based and there is no graph to traverse or data-flow equations to be solved. Only on-the-fly information is retrieved as needed. Unlike most of the others we compute the slice for every variable in the program including local and global. As new variables are encountered they are added to the slice profile.

VII. Conclusions and Future Work

There are several features of srcSlice that the current release does not contain. We intend to slowly roll these extra features out as they are tested. We would like the ability to directly query srcSlice's map instead of having to parse the output. We also want to add the ability for users to create their own output formats to be more interoperable with other tools. In terms of what data srcSlice is able to obtain, we intend to add the ability to record how objects' member functions and variables are being used on a per-object basis. We also are working to improve accuracy of pointer aliasing. While pointer-analysis is difficult, much information can be gleaned statically. Currently, srcSlice is built for C/C++. The srcML format, however, additionally supports Java, and C# and we intend to have srcSlice support those languages in the future.

References

[1] Alomari, H. W., Collard, M. L., and Maletic, J. I., " A Very Efficient and Scalable Forward Static Slicing Approach", IEEE International Working Conference on Reverse Engineering (WCRE'12), October 15-18, 2012, pp. 425-434.

[2] Alomari, H. W., Collard, M. L., Maletic, J. I., Alhindawi, N., and Meqdadi, O., "srcSlice: Very Efficient and Scalable Forward Static Slicing", *Journal of Software: Evolution and Process*, vol. 26, no. 11, Nov. 2014, pp. 931-961.

[3] Collard, M. L., Decker, M., and Maletic, J. I., "Lightweight Transformation and Fact Extraction with the srcML Toolkit", IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11), Sept 25-26 2011, pp. 10 pages.

[4] Collard, M. L., Maletic, J. I., and Robinson, B. P., "A Lightweight Transformational Approach to Support Large Scale Adaptive Changes", in International Conference on Software Maintenance (ICSM '10), 2010, pp. 1-10.

[5] Cordy, J. R., Eliot, N. L., and Robertson, M. G., "TuringTool: A User Interface to Aid in the Software Maintenance Task", *IEEE TSE*, vol. 16, no. 3, 1990, pp. 294-301.

[6] Gallagher, K. and Binkley, D. W., "Program Slicing", in Frontiers of Software Maintenance (FoSM '08), 2008, pp. 58-67.

[7] Gallagher, K. B. and Lyle, J. R., "Using Program Slicing in Software Maintenance", *IEEE TSE*, vol. 17, 1991, pp. 751-761.

[8] Horwitz, S., Reps, T., and Binkley, D., "Interprocedural Slicing using Dependence Graphs", *ACM SIGPLAN Notices*, vol. 23, no. 7, 1988, pp. 35-46.

[9] Kumar, S. and Horwitz, S., "Better Slicing of Programs with Jumps and Switches", International Conference on Fundamental Approaches to Software Engineering (FASE '02), 2002, pp. 96-112

[10] Liang, D. and Harrold, M. J., "Slicing Objects Using System Dependence Graphs", International Conference on Software Maintenance (ICSM '98), 1998, pp. 358-367.

[11] Ohata, F., Hirose, K., Fujii, M., and Inoue, K., "A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information", Asia-Pacific on Software Engineering Conference (APSEC '01), 2001, pp. 273-283.

[12] Ottenstein, K. J. and Ottenstein, L. M., "The Program Dependence Graph in a Software Development Environment", *ACM SIGSOFT Software Eng Notes*, vol. 9, no. 3, 1984, pp. 177-184.

[13] Silva, J., "A Vocabulary of Program Slicing-based Techniques", *ACM Comput. Surv.*, vol. 44, no. 3, 2012, pp. 1-41.

[14] Tip, F., "A Survey of Program Slicing Techniques", *Journal of Programming Language*, vol. 3, 1995, pp. 121-189.

[15] Tonella, P., "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis", *IEEE TSE*, vol. 29, no. 6, 2003, pp. 495-509.

[16] Weiser, M., *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. PhD Thesis*, U of Michigan, Ph.D. Dissertation, 1979.

[17] Weiser, M., "Program Slicing", International Conference on Software Engineering (ICSE '81), 1981, pp. 439-449.

[18] Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L., "A Brief Survey of Program Slicing", *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, 2005, pp. 1-36.