Evaluating Context-Sensitive Slicing and Chopping

Jens Krinke
Universität Passau
Passau, Germany

Abstract

We present an empirical evaluation of three context-sensitive slicing algorithms and five context-sensitive chopping algorithms, and compare them to context-insensitive methods. Besides the algorithms by Reps et al. and Agrawal we investigate six new algorithms based on variations of k-limited call strings and approximative chopping based on summary information. It turns out that chopping based on summary information may have a prohibitive complexity, and that approximate algorithms are almost as precise and much faster.

1. Introduction

Slicing is an established technique for reverse engineering and other analyses like testing or debugging—it is available in research prototypes and even in commercial products. There are two main approaches to slicing: The original slicing technique from Weiser [19] is based on traditional data flow analysis; the other approach is based on Program Dependence Graphs [13, 6, 7]. Extensive evaluations of different slicing algorithms have not really been done yet—for control flow graph based algorithms some data reported by Atkinson and Griswold can be found in [4, 3, 2]. The only evaluation of program dependence based algorithms that the author is aware of has been conducted by Agrawal and Guo [1], who just compare two algorithms, where one has flaws (as shown in this paper). Slicing identifies statements in a program which may influence a given statement (the slicing criterion), but it cannot answer the question why a specific statement is part of a slice. A more focused approach can help: *Chopping* reveals the statements which are involved in a transitive dependence from one specific statement (the source criterion) to another specific statement (the target criterion). An evaluation of chopping algorithms has never been done, [14] reports only limited experience.

In the following we introduce different slicing (Section 2) and chopping (Section 4) algorithms, which are evaluated in Section 3 and 5. Conclusions are drawn in Section 6.

1.1. Program Dependence Graphs

The *Program Dependence Graph (PDG)* [13, 6, 7] is a directed graph whose vertices represent the statements and control predicates that occur in a program. It also contains a unique *entry* vertex. The edges represent the *dependences* between the components of the program: A *control dependence* edge from vertex v_1 to v_2 represents that the evaluation of the predicate that is represented by v_1 is controlling the execution of the component that is represented by v_2. A *data dependence* edge from vertex v_1 to v_2 represents that the component represented by v_1 assigns to a variable which may be used at the component represented by v_2.

The extension of the PDG for *interprocedural programs* introduces more vertices and edges: For every procedure a *procedure dependence graph* is constructed, which is basically a PDG with *formal-in* and *-out* vertices for every formal parameter of the procedure. A procedure call is represented by a *call* vertex and *actual-in* and *-out* vertices for each actual parameter. The call vertex is connected to the entry vertex by a *call* edge, the *actual-in* vertices are connected to their matching *formal-in* vertices via *parameter-in* edges and the *actual-out* vertices are connected to their matching *formal-out* vertices via *parameter-out* edges. Such a graph is called *Interprocedural Program Dependence Graph (IPDG)*. The *System Dependence Graph (SDG)* is an IPDG, where *summary edges* between actual-in and actual-out have been added representing transitive dependence due to calls [7].

1.2. Context-Insensitive Slicing

Slicing on (intraprocedural) PDGs is just a simple reachability problem: A (backward) slice to a vertex (the *slicing criterion*) is the set of vertices from which the criterion vertex is reachable. The algorithm is shown in Figure 1. We call it *context-insensitive slicing (CIS)*. Later on, we will also use forward slicing, which is the set of vertices reachable from the criterion vertex. If this algorithm is used on IPDGs or SDGs, the resulting slices are not accurate, as the *calling context* is not preserved: the algorithm may traverse

Let $G = (V,E)$ *be the given IPDG*
Let $s \in V$ *be the given slicing criterion*
$W = \{s\}$, *mark s as visited*
while W *is not empty* do
  remove one element w from W
  foreach $v \to w \in E$ do
    if *v is not yet marked* then
      $W = W \cup \{v\}$, *mark v as visited*
return *the set of all visited vertices*

Figure 1. Context-Insensitive Slicing

Let $G = (V,E)$ *be the given SDG*
Let $s \in V$ *be the given slicing criterion*
$W = \{s\}$, *mark s with* up
while W *is not empty* do
  remove one element w from W
  Let m_w *be the mark of w*
  foreach $v \to w \in E$ do
    Let m_v *be the mark of v*
    if $m_v \neq$ up then
      if $v \to w$ *is a parameter-in or call edge*
      and $m_w \neq$ down then
        $W = W \cup \{v\}$, *mark v as* up
      elsif $v \to w$ *is a parameter-out edge*
      and $m_v \neq$ down then
        $W = W \cup \{v\}$, *mark v as* down
      elsif $m_v \neq m_w$ then
        $W = W \cup \{v\}$, *mark v as* m_w
return *the set of all visited vertices*

Figure 2. Context-Sensitive Slicing in SDGs

a parameter-in edge coming from a call site into a procedure, may traverse some edges there and may traverse a parameter-out edge going to a different call site. The sequence of traversed edges (the path) is a so called *unrealizable path*. This means it is impossible for an execution that a called procedure does not return to its call site. We consider an interprocedural slice to be precise if all vertices included in the slice are reachable from the criterion by a *realizable* path.

1.3. Slicing with Summary Edges

Accurate slices can be calculated with a modified algorithm on SDGs. The benefit of SDGs is the presence of the *summary* edges that represent transitive dependence due to calls. The idea of [7, 15] is first to slice from the criterion only ascending into calling procedures, and then to slice from all visited vertices only descending into called procedures. The algorithm we use is shown in Figure 2, which we call *summary information slicing (SIS)*.

2. Context-Sensitive Slicing

There are situations where summary edges cannot be used: In presence of interference in explicit parallel programs, dependence is no longer transitive [9], a requirement for summary edges. As interference dependence crosses procedure boundaries, it cannot be summarized by summary edges. Under such circumstances the calling context has to be preserved explicitly during the traversal of the (threaded) IPDG.

A different approach for context-sensitive analysis is based on *call strings* [18] where the calling context is encoded explicitly during analysis in a sequence of call sites simulating the call stack of an abstract machine. In the rest of this section we will show how to use this approach for slicing (and later, chopping) in program dependence graphs. Let each call vertex and its actual-in and -out vertices in the IPDG G be given a unique index s_i. A sequence of call sites $c = s_{i_1} \ldots s_{i_n}$ is called a *call string*. During traversal of a parameter-out edge from a call site s going *down* into the called function, we generate a new, longer call string $c' = sc$. If we traverse a parameter-in or call edge back *up* to a call site s', this call site must *match* the current leading element of the call string ($c = s'c'$). Using call strings we can define a context-sensitive slicing method which is as precise as context-sensitive slicing based on summary edges—it is precise with respect to realizable paths [14].

2.1. Explicitly Context-Sensitive Slicing

In Figure 3 we show a general slicing algorithm which obeys the calling context via call strings. Variants of the algorithm come from the definition of down, up and match. The simplest definition is as follows:

$$\begin{aligned} \text{down}(c,s) &\to \text{cons}(s,c) \\ \text{up}(c) &\to \begin{cases} \text{cdr}(c) & | \; c \neq \varepsilon \\ \varepsilon & | \; c = \varepsilon \end{cases} \\ \text{match}(c_1,c_2) &\to c_1 = c_2 \end{aligned}$$

In presence of recursion this doesn't work, as neither the set of call strings nor the call strings themselves are finite. Agrawal and Guo presented an improved algorithm named ECS in [1], where the call strings are cycle free. They define down as follows (up and match are as above):

$$\text{down}(c,s) \to \begin{cases} \text{cons}(s,c) & | \; c = s_1 s_2 \ldots s_k \wedge \forall_{s_i} : s \neq s_i \\ s_i \ldots s_k & | \; c = s_1 s_2 \ldots s_k \wedge s = s_i \end{cases}$$

However, using this definition the resulting slices are not correct, as they might leave out vertices (statements) of the slice. The incorrectness is based on the following observation: As soon as a call string c would form $c =$

Let $G = (V,E)$ *be the given IPDG*
Let $s \in V$ *be the given slicing criterion*
$W = \{(s,\varepsilon)\}$
while W *is not empty* do
  remove one element (w,c) *from* W
  foreach $v \to w \in E$ do
    if v *has not been marked with a context* c'
      for which match(c,c') *holds* then
      if $v \to w$ *is a parameter-in or call edge* then
        Let s_v *be the call site of* v
        if $c = \varepsilon \vee \text{car}(c) = s_v$ then
          $c' = \text{up}(c)$
          $W = W \cup \{(v,c')\}$, *mark* v *with* c'
      elsif $v \to w$ *is a parameter-out edge* then
        Let s_w *be the call site of* w
        $c' = \text{down}(c, s_w)$
        $W = W \cup \{(v,c')\}$, *mark* v *with* c'
      else
        $W = W \cup \{(v,c)\}$, *mark* v *with* c
return *the set of all visited vertices*

Figure 3. Explicitly Context-Sensitive Slicing

$s_x s_{y_1} .. s_{y_n} s_x s_{z_1} ... s_{z_m}$, it is replaced by $c' = s_x s_{z_1} ... s_{z_m}$ to remove the cycle. Now, the algorithm fails to propagate the effects with call string c' (which includes effects with call string c) back to the call site s_{y_1}, as $c'' = s_{y_1} .. s_{y_n} s_x s_{z_1} ... s_{z_m}$ is not generated by up(c'). A counter example is given in Figure 4, where two procedures are shown as an IPDG. Procedure 1 contains its entry vertex `1` and a call site *A*, composed from the call vertex `A1`, two actual-in vertices `A2` and `A3`, and an actual-out vertex `A4`. The second procedure contains its entry node `2`, two formal-in vertices `3` and `4`, a formal-out vertex `6`, a vertex `if` for an if-statement, and a call site *B*, composed from the call vertex `B1`, two actual-in vertices `B2` and `B3`, and an actual-out vertex `B4`. The vertices inside the procedures are connected by control and data dependence edges. Also, procedure 1 calls procedure 2 at call site *A*, and procedure 2 calls itself at call site *B*. The actual-in/actual-out vertices are properly connected to their formal-in/formal-out vertices by parameter-in/parameter-out edges; the call edges are not shown. Let us do a backward slice from `A4`: The initial worklist is $\{(\texttt{A4},\varepsilon)\}$. `A4` is reachable from `A1` and `6`, and the worklist will contain $\{(\texttt{A1},\varepsilon),(\texttt{6},A)\}$. Next, `A1` is visited, which is reachable from `1`, and this leaves the worklist as $\{(\texttt{6},A)\}$. Vertex `6` is reached (transitively) from `2`, `3`, `B1`, `B4`, `if`, which are all marked with the call string *A*. `B4` is reached from `6` by recursion. At this point, the worklist is $\{(\texttt{3},A),(\texttt{6},BA)\}$ and vertex `6` has two marks: *A* and *BA*. Vertex `3` is reachable from `A2`, where the call string matches (at this point, vertex `3` is only marked with *A*). It is also reachable from `B2`, but the call string doesn't match. From vertex `6` the call string *BA* is now propagated to `2`, `3`,

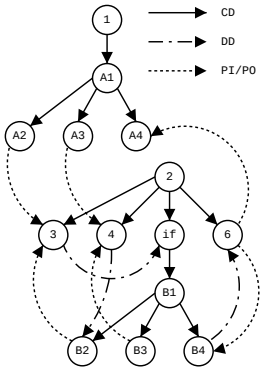

Figure 4. Counter example for ECS

`B1`, `B4`, and `if`. Because vertex `3` is now marked with *BA*, we propagate *A* to `B2`, which is visited the first time. `B2` is reachable from vertex `4` and transitively from `A3`. At this point, only `B3` has not been visited yet, the vertices `1` and `A1`–`A4` are marked with ε; `2`, `3`, `if`, `6`, `B1`, `B4` are marked with *A* and *BA*, and `4` and `B2` are only marked with *A*. Now, the worklist only contains $\{(\texttt{B4},BA)\}$. Due to the recursive call, a new call string *BBA* would be generated and propagated to vertex `6`. However, the cycle removement in *down* folds *BBA* into *BA*, which has been used previously at vertex `6`. The ECS algorithm now terminates (the worklist is empty) but the generated slice doesn't contain vertex `B3`, which is wrong: a propagation of *BBA* would visit `B3`.

2.2. Limited Context Slicing

One popular approach to circumvent the infinity problems is the length limitation of the call strings (*call string suffix approximation* in [18]). For a given *k*, the call string is not allowed to be longer than *k* elements. If the call string is already *k* elements long before concatenation, the oldest element is removed:

$$\text{down}(c,s) \to \begin{cases} \text{cons}(s,c) & \mid c = s_1 s_2 \ldots s_l \wedge l < k \\ s s_1 \ldots s_{k-1} & \mid l = k \end{cases}$$

We can use the general algorithm of Figure 3 directly with these modifications. We call this variant *k-limited context slicing (kLCS)*. This approach becomes quite imprecise (it is still correct) in presence of recursion: Consider a call string of maximal length *k*. If this call string is propagated into a recursive procedure, it may be propagated *k* times into the called recursive procedure. The result is a call string that only contains call sites in the recursive procedure. This call string may then be propagated *k* times back into the calling recursive procedure. Now, the resulting call string is empty and all procedures that call the recursive procedure are marked with the empty call string. The result is a reduced precision.

Let $G = (V, E)$ *be the given IPDG*
Let $s \in V$ *be the given slicing criterion*
$W = \{(s, \varepsilon)\}$
while W *is not empty* do
  remove one element (w, c) *from* W
  foreach $v \to w \in E$ do
    if v *has not been marked with a context* c'
      for which match(c, c') *holds* then
      if $v \to w$ *is a parameter-in or call edge* then
        Let s_v *be the call site of* v
        if $c = \varepsilon \vee \text{car}(c) = s_v$ then
          if s_v *is marked as recursive*
          and v *has not been marked with a context* c'
            for which match(c, c') *holds* then
            $W = W \cup \{(v, c)\}$, *mark* v *with* c
          $c'' = \text{up}(c)$
          $W = W \cup \{(v, c'')\}$, *mark* v *with* c''
        elsif $v \to w$ *is a parameter-out edge* then
          Let s_w *be the call site of* w
          if s_w *is marked as recursive*
          and $\text{car}(c) = s_w$ then
            $W = W \cup \{(v, c)\}$, *mark* v *with* c
          else
            $c' = \text{down}(c, s_w)$
            $W = W \cup \{(v, c')\}$, *mark* v *with* c'
        else
          $W = W \cup \{(v, c)\}$, *mark* v *with* c
return *the set of all visited vertices*

Figure 5. Folded Context-Sensitive Slicing

2.3. Folded Context Slicing

As we have seen, a way to make the call strings finite is to remove cycles. Instead of removing them explicitly we now present a different approach. First, we build the *calling context graph*. This is a graph where the vertices are the call sites and the edges (v, w) represent calls that lead from call site v into a procedure that contains call site w. In that graph, the strongly connected components are folded into one single vertex and the call sites of such a component are replaced by one single call site. During that process all call sites are marked if they are recursive. The definitions of down etc. are only adapted, not changed. For example:

$$\text{down}(c, [s]) \to \left\{ \begin{array}{l|l} \text{cons}([s], c) & c = [s_1][s_2]\ldots[s_l] \wedge l < k \\ [s][s_1]\ldots[s_{k-1}] & l = k \end{array} \right.$$

If we use the modified down etc. within the algorithm of Figure 3, we obtain the same incorrectness as Agrawal. Figure 5 presents a corrected algorithm, modified as follows:

1. If the algorithm descends into a called recursive function, it propagates *two* call strings: the call string generated by *up* and the actual call string (because of the possible recursion).
2. If the algorithm ascends into a calling recursive function, it propagates the actual call string if the actual call site is already present in (the first element of) the call string or the call string generated by *down* otherwise.

This modified algorithm is general enough to also be used in an unlimited version, which is defined by:

$$\text{down}(c, [s]) \to \text{cons}([s], c)$$

We call these variants *unlimited folded context slicing (UFCS)* and *k-limited folded context slicing (kFCS)*. The evaluation will show that the unlimited variant is unpractical due to combinatorial explosion of the set of call strings.

3. Evaluation I: Slicing

Unlike Agrawal and Guo, who only compared context-insensitive slicing (CIS) with ECS in [1], we have implemented all slicing algorithms of the previous sections to evaluate them fully. We used our infrastructure to analyze C programs [10] to implement all slicing (and chopping) algorithms with a flow-insensitive but context-sensitive alias analysis similar to [5]. The details of the analyzed programs are shown in Figure 6. The programs stem from three different sources: ctags, patch and diff are the GNU programs. The rest are from the benchmark database of the PROLANGS Analysis Framework (PAF) [17]. The 'LOC' column shows lines-of-code (measured via `wc -l`), the 'ENTR' column the amount of procedures (the number of entry vertices in the PDG) and the 'nodes' column shows the number of vertices in the IPDG. Like Agrawal and Guo, we selected the formal-in vertices as slicing criteria to make the results comparable; the amount of resulting slicing criteria (the number of formal-in vertices) is shown in column 'slices'. The limited size of the programs is due to the amount of slices that had to be done, which caused a quadratic runtime for one complete test of a program.

3.1. Precision of kLCS and kFCS

How precise can we be with the new algorithms *k*LCS and *k*FCS? How do we measure precision? As context-insensitive slicing (CIS) is the most simple (and imprecise) algorithm we consider it as 0% precision. Slicing with summary edges (SIS) is precise in respect to realizable paths and we consider that as 100% precision. In Figure 7, we present the precision of *k*LCS and *k*FCS. The CIS column gives the average size of a CIS slice and the SIS column the average size of a SIS slice. There is already a big variation: The average CIS slice is between three and 300% larger than the average SIS slice. For many programs, we already have a high precision for small k in both *k*LCS and *k*FCS. In some cases (shown in bold), the precision of *k*FCS

		LOC	ENTR	nodes	slices	edges	summary	%	time
A	gnugo	3305	38	3875	281	10657	2064	16	0.03
B	ansitape	1744	76	6733	1082	18083	12746	41	0.15
C	assembler	3178	685	13393	2401	97908	114629	54	3.58
D	cdecl	3879	53	5992	697	17322	9089	34	0.08
E	ctags	2933	101	10042	1621	24854	20483	45	0.24
F	simulator	4476	283	9143	1019	22138	5022	18	0.06
G	rolo	5717	170	37839	6540	264922	170108	39	5.53
H	compiler	2402	49	15195	1017	45631	58240	56	0.80
I	football	2261	73	8850	818	30474	17605	37	0.35
J	agrep	3968	90	11922	1403	35713	12343	26	0.19
K	bison	8313	161	25485	3744	84794	29739	26	0.72
L	patch	7998	166	20484	3099	104266	83597	44	4.39
M	flex	7640	121	38508	5191	235687	144496	38	4.19
N	diff	13188	181	46990	10130	471395	612484	57	28.2

Figure 6. Details of the test programs

	CIS	SIS	1LCS	2LCS	3LCS	4LCS	5LCS	6LCS	1FCS	2FCS	3FCS	4FCS	5FCS	6FCS
A	1861	1798	100	100	100	100	100	100	100	100	100	100	100	100
B	2909	1645	7	74	74	74	74	100	7	74	74	74	74	100
C	6458	4286	48	48	48	100	100	100	48	48	48	100	100	100
D	1039	880	99	99	100	100	100	100	99	99	100	100	100	100
E	3207	2010	100	100	100	100	100	100	100	100	100	100	100	100
F	5455	3212	83	83	100	100	100	100	83	83	100	100	100	100
G	12819	7766	46	57	84	92	92	92	46	57	84	92	92	**92**
H	7474	6731	22	26	26	26	26	26	22	**30**	**32**	**46**	**73**	**73**
I	3081	2593	100	100	100	100	–	–	100	100	100	100	100	100
J	3521	3183	41	49	59	100	100	100	41	49	59	100	100	100
K	7215	1859	74	96	96	97	97	97	74	96	96	**98**	**100**	**100**
L	9680	7965	92	99	99	100	100	100	92	99	99	100	100	100
M	14558	6172	29	29	29	29	29	29	*29*	**30**	**30**	**32**	**98**	**100**
N	19641	9179	88	98	98	98	98	–	88	98	**99**	**99**	**100**	**–**

Figure 7. Precision of *k*LCS and *k*FCS (avg. size)

is higher than *k*LCS. In only one case (flex, M) the precision of 1FCS is slightly less than 1LCS (less than 0.1%, shown in italics). We limited the amount of memory and time in which all slices have to be calculated: The tests marked with "–" needed more than 300MB core memory or didn't finish in fewer than eight hours on 1GHz machines with 512MB.

3.2. Speed of kLCS and kFCS

In Figure 8 we see the runtimes of the test cases for the different slicing algorithms (in sec.). The given times are for the complete set of slicing criteria. To get the average time for a single slice one has to divide by the number given in the 'slices' column of Figure 6. We haven't given the average time, because most numbers would be sub-second. In the first two columns of Figure 8 we can see that the time needed to do SIS slices is less than the time to do CIS slices if the SIS slices have a much higher precision: the quadratic complexity of SIS is more than compensated by the smaller amount of vertices that are visited.

A typical problem of call string approaches is the combinatorial explosion of the generated call strings. This is also present in *k*LCS (see A, B, E, H, I, N) and *k*FCS (see E, H, N). However, due to the increased precision and the therefore smaller amount of visited vertices we also experienced reduced runtimes for higher *k*, see C, D, K for *k*LCS and B, C, D, K for *k*FCS. In many cases *k*FCS is slightly slower than *k*LCS resulting from the overhead propagating *two* call strings. In other cases it is much faster than *k*LCS—it is also less likely to suffer combinatorial explosion. In some situations *k*FCS is much slower than *k*LCS. This is not related to a problem with *k*FCS but stems from the higher precision of *k*FCS in these cases (see Figure 7 for comparison).

A further increased *k* (not shown here) causes many test cases to suffer combinatorial explosion up to a point where slicing is not any longer possible with *k*LCS or *k*FCS—like

	CIS	SIS	1LCS	2LCS	3LCS	4LCS	5LCS	6LCS	1FCS	2FCS	3FCS	4FCS	5FCS	6FCS
A	1,13	1,23	3,05	5,99	14,8	54,1	267	1357	2,65	4,36	8,65	11,0	11,4	12,5
B	7,37	5,56	17,8	18,8	29,7	73,6	240	845	17,8	18,0	26,1	49,8	82,5	77,0
C	81,5	118	158	206	228	171	177	143	160	207	230	174	178	144
D	1,65	1,76	3,70	5,18	8,49	11,5	10,6	8,30	3,75	5,13	8,48	11,6	10,8	8,18
E	11,1	9,83	17,2	24,0	38,3	148	352	574	17,3	24,3	41,6	137	343	579
F	12,4	7,88	27,4	97,4	97,4	121	176	206	27,4	87,5	98,2	121	178	208
G	480	447	768	1033	1174	1511	2007	2738	749	964	1077	1355	1804	2600
H	21,3	31,0	68,9	200	470	1461	5123	19140	48,0	200	487	2030	5220	8540
I	6,01	7,33	14,3	29,0	75,6	621	–	–	13,9	25,2	29,6	41,1	46,3	45,3
J	12,8	11,6	30,3	39,1	52,0	39,7	59,5	120	29,9	38,3	49,8	31,5	31,9	30,6
K	83,5	30,4	108	162	323	373	335	284	109	165	334	430	397	374
L	119	133	227	313	382	402	398	424	229	315	384	404	400	426
M	287	151	586	918	1289	1672	1899	2259	591	1088	2127	4571	2102	2420
N	1824	1307	1170	1580	2737	6675	20849	–	1109	1341	1946	3182	9227	–

Figure 8. Runtimes of *k*LCS and *k*FCS (sec.)

already test case I for $k > 4$. The previously described effect of higher speed for higher *k* is never compensating the combinatorial explosion and we cannot share the experience of Agrawal and Guo [1] who successfully used an unlimited call string slicing algorithm. Our experience is backed up by Atkinson and Griswold in [2], who reported the same for the control flow graph approach.

If we compare the runtimes of *k*LCS and *k*FCS with SIS, we see that even for $k = 1$ the runtimes of *k*LCS and *k*FCS are much higher than those of SIS. This leads to our persuasion that as long as summary edges are calculable, one should use SIS for slicing. This stays the same if we consider the overhead to calculate the summary edges: the generation of summary edges is part of the analysis to generate the PDG, which is expensive even without summary edge generation. The overhead is only unaffordable in situations where only one or two slices have to be done; in those situations the use of control flow graph based slicing is probably better anyway. If summary edges are not available, one must revert to *k*-limiting algorithm, where *k*FCS is preferable to *k*LCS. The *k*-limiting algorithms are also more important in chopping, because the known context-sensitive chopping algorithms are expensive.

4. Chopping

We also applied our approach to the problem of chopping. In [8] Jackson and Rollins defined a restricted form of chopping: a chop for a chopping criterion (s,t) is the set of vertices that are part of an influence of the (source) vertex *s* onto the (target) vertex *t*. This is basically the set of vertices which are lying on a path from *s* to *t* in the PDG. Jackson and Rollins restricted *s* and *t* to be in the same procedure and only traversed control dependence edges, data dependence edges and summary edges but not parameter or call edges. The resulting chop is called a *same-level truncated chop*, "truncated" because the vertices of called procedures are not included. In [14] Reps and Rosay presented more variants of precise chopping. A same-level *non-truncated* chop is like the truncated chop but includes the vertices of called procedures. They also present *different-level* truncated and non-truncated chops (which they call *interprocedural*), where the vertices of the chopping criterion are allowed to be in different procedures. Due to space limitations, we are not considering the different-level variants here. We are focusing on *non-truncated same-level chopping*, because it is the most challenging problem to formulate this variant based on call strings: Due to the limitations of length limited call strings, the same-level property can get lost during graph traversal.

4.1. Context-Insensitive Chopping

In the intraprocedural or context-insensitive case a chop for a chopping criterion (s,t) is basically the intersection of a backward slice for *t* with a forward slice for *s*. An algorithm would not use intersection, as set operations are expensive for large sets like slices. Instead, the algorithm presented in Figure 9 uses a two-phase approach: in the first phase, the backward slice is done and in the second phase the forward slice is done, where only vertices which have been visited during the backward slice phase are considered. We call this algorithm *context-insensitive chopping (CIC)*.

4.2. Chopping with Summary Edges

The precise context-sensitive chopping algorithm, which uses summary edges, is depicted in Figure 10. It basically starts with an intraprocedural chop which is done with the algorithm in Figure 9. There is a slight modification to that

```
Let G = (V,E) be the given IPDG
Let (s,t) ∈ V × V be the given chopping criterion
W_B = {t}, mark t as visited in the backward phase
while W_B is not empty do
   remove one element w from W_B
   foreach v → w ∈ E do
      if v is not yet marked then
         W_B = W_B ∪ {v}, mark v as visited in the backward phase
if s has been marked in the backward phase
and s has not yet been marked in the forward phase then
   W_F = {s}, mark t as visited in the forward phase
   while W_F is not empty do
      remove one element w from W_F
      foreach w → v ∈ E do
         if v has been marked in the backward phase then
            W_F = W_F ∪ {v}, mark v as visited in the forward phase
return the set of all in the forward phase visited vertices
```

Figure 9. Context-Insensitive Chopping

algorithm: as it has to do a chop only inside one single procedure, it is not allowed to traverse parameter or call edges (but it is allowed and required to traverse summary edges). From the calculated chop all pairs of actual-in/-out vertices which are connected by a summary edge are extracted and the initial worklist is filled with the pairs of the corresponding formal-in/formal-out vertices. Now for every pair in the worklist a new intraprocedural chop is generated and added to the starting chop. Again, all summary edges are extracted and the pairs of the corresponding formal-in/formal-out vertices are added to the worklist if they have not been added before. This is repeated as long as there are elements in the worklist. The now extended chop is the resulting precise interprocedural chop. The algorithm generates a *same-level non-truncated* chop: both vertices of the chopping criterion have to be in the same procedure. We call this algorithm *summary information chopping (SIC)*. The intraprocedural version of the algorithm in Figure 9 is a same-level *truncated* chop.

4.3. Mixed Context-Sensitivity Chopping

Chopping with summary edges is expensive due to the high complexity. A simple improvement is to combine context-sensitive slicing with summary edges with context-insensitive chopping. We are using the same two-phase approach as in context-insensitive chopping, but with context-sensitive slicing using summary edges instead of context-insensitive slicing. Also, as we are doing same-level chopping, we only have to descend into called procedures. This algorithm, called *Mixed context-sensitivity chopping (MCC)*, is shown in Figure 11. It is surprisingly precise as we will see in Section 5.

```
Let G = (V,E) be the given SDG
Let (s,t) ∈ V × V be the given chopping criterion
Let C be the intraprocedural chop for (s,t)
W = {v → w | v,w ∈ C, v → w is a summary edge}
while W is not empty do
   remove one element v → w from W
   Let v' be the to v corresponding formal-in vertex
   Let w' be the to w corresponding formal-out vertex
   if v',w' has not been marked then
      mark v',w' as visited
      Let C' be the intraprocedural chop for (v',w')
      C = C ∪ C'
      W = W ∪ {v → w | v,w ∈ C', v → w is a summary edge}
return C
```

Figure 10. Chopping with Summary Edges

```
Let G = (V,E) be the given SDG
Let (s,t) ∈ V × V be the given chopping criterion
W_B = {t}, mark t as visited in the backward phase
while W_B is not empty do
   remove one element w from W_B
   foreach v → w ∈ E not a parameter-in or call edges do
      if v is not yet marked then
         W_B = W_B ∪ {v}, mark v as visited in the backward phase
if s has been marked in the backward phase
and s has not yet been marked in the forward phase then
   W_F = {s}, mark t as visited in the forward phase
   while W_F is not empty do
      remove one element w from W_F
      foreach w → v ∈ E not a parameter-out edge do
         if v has been marked in the backward phase then
            W_F = W_F ∪ {v}, mark v as visited in the forward phase
return the set of all in the forward phase visited vertices
```

Figure 11. Mixed Context-Sensitivity

4.4. Limited/Folded Context Chopping

Now, We can adapt the *k*LCS algorithm to chopping (Figure 12). Again, we use a two-phase approach: First, the backward slice is done and all vertices are marked with the encountered call strings. Second, basically a forward slice is computed, only considering vertices which have been marked with a matching call string in the first phase. They also have still to be unmarked with any matching context from the second phase. The definitions of down, up and match are the same as in *k*LCS. We call this algorithm *k-limited context chopping (kLCC)*. In the same style we can adapt the *k*FCS algorithm to chopping. As this is straightforward (and due to space limitations) we are not presenting the algorithm here (but it has been implemented and will be evaluated in Section 5). We call this algorithm *k-limited folded context chopping (kFCC)*.

```
Let G = (V,E) be the given IPDG
Let (s,t) ∈ V × V be the given chopping criterion
W_B = {(t,ε)}, mark t with (B,ε)
while W_B is not empty do
  remove one element (w,c) from W_B
  foreach v → w ∈ E do
    if v has not been marked with (B,c')
      for which match(c,c') holds then
      if v → w is a parameter-in or call edge then
        Let s_v be the call site of v
        if c = ε ∨ car(c) = s_v then
          c' = up(c)
          W_B = W_B ∪ {(v,c')}, mark v with (B,c')
      elsif v → w is a parameter-out edge then
        Let s_w be the call site of w
        c' = down(c,s_w)
        W_B = W_B ∪ {(v,c')}, mark v with (B,c')
      else
        W_B = W_B ∪ {(v,c)}, mark v with (B,c)
if s has been marked with (B,ε) then
  W_F = {(s,ε)}, mark s with (F,ε)
  while W_F is not empty do
    remove one element (w,c) from W_F
    foreach w → v ∈ E do
      if v has been marked with (B,c')
        for which match(c',c) holds then
        if v has not been marked with (F,c')
          for which match(c,c') holds then
          if w → v is a parameter-out edge then
            Let s_v be the call site of v
            if c = ε ∨ car(c) = s_v then
              c' = up(c)
              W_F = W_F ∪ {(v,c')}, mark v with (F,c')
          elsif w → v is a parameter-in or call edge then
            Let s_w be the call site of w
            c' = down(c,s_w)
            W_F = W_F ∪ {(v,c')}, mark v with (F,c')
          else
            W_F = W_F ∪ {(v,c)}, mark v with (F,c)
return the set of all in the forward phase visited vertices
```

Figure 12. Explicitly Context-Sensitive

4.5. An improved precise algorithm

Now, we present an improved precise chopping algorithm which has a much higher speed. The SIC algorithm of Section 4.2 calculates a new chop for every pair of formal-in and formal-out vertices that have a summary edge between the corresponding actual-in and -out vertices included in the chop. Our observation is as follows: if two summary edges of one call site are included in the chop, we do not have to do a chop for the corresponding pairs of formal-in/formal-out vertices separately. Instead, we can do a single chop between the set of corresponding formal-in vertices and the set of corresponding formal-out vertices.[1] The algorithm in Figure 13 is based on the merging of summary edges dependent of their call site and is therefore called *summary-merged chopping (SMC)*. After calculating the starting chop, we collect all new summary edges of visited call sites. Then we do a new chop for every call site between the set of the corresponding formal-in vertices and the set of the corresponding formal-out vertices. The resulting vertices are added to the starting chop and we repeat the procedure with the new resulting summary edges until there are no more new summary edges left. This causes low runtimes as we are doing a much smaller number of chops.

```
Let G = (V,E) be the given SDG
Let (s,t) ∈ V × V be the given chopping criterion
Let C be the intraprocedural chop for (s,t)
W = {v → w | v,w ∈ C, v → w is a summary edge}
while W is not empty do
  S = ∅, T = ∅
  foreach call site c in C do
    foreach w → v ∈ W, c call site of w → v do
      Let v' be the to v corresponding formal-in vertex
      Let w' be the to w corresponding formal-out vertex
      if v',w' has not been marked then
        mark v',w' as visited
        S = S ∪ v', T = T ∪ w'
    Let C' be the intraprocedural chop for (S,T)
    C = C ∪ C'
    W = {v → w | v,w ∈ C', v → w is a summary edge}
return C
```

Figure 13. Merging Summary Edges

5. Evaluation II: Chopping

We have implemented all previously presented chopping algorithms (CIC, SIC, SMC, MCC, *k*LCC and *k*FCC) to fully evaluate them. As chopping criteria we have chosen every tenth of all same-level pairs of formal-in/formal-out vertices. Again, we measured the time needed to do the complete set of chops, as the average time to calculate one chop is sub-second.

[1] Sketch of Proof: Let S be the set of all summary edges of one single call site which are all included in the starting chop. Let $(i_1,o_1) \in S$ and $(i_2,o_2) \in S$ be a randomly chosen pair of summary edges. Let (i'_1,o'_1) and (i'_2,o'_2) be the corresponding pairs of formal-in/formal-out vertices. Let C_1 be the chop for (i'_1,o'_1) and C_2 for (i'_2,o'_2). Let C be the chop for $(\{i'_1,i'_2\},\{o'_1,o'_2\})$. A problem can only be caused by a vertex that is included in C but not in $C_1 \cup C_2$. For any $c \in C$, $c \notin C_1$, $c \notin C_2$ one of the paths $i'_1 \rightarrow^* c \rightarrow^* o'_2$ or $i'_2 \rightarrow^* c \rightarrow^* o'_1$ must exist. Therefore, a summary edge (i_1,o_2) or (i_2,o_1) must exist. These edges now must be included in the starting chop (and thus also in S) as i_1,i_2,o_1,o_2 are all in the starting chop. Because of these summary edges we have to include the chop C_3 for (i'_1,o'_2) or C_4 for (i'_2,o'_1) into the resulting chop and therefore c will be added to the resulting chop anyways.

	chops	CIC	SIC	SMC	MCC	1LCC	2LCC	3LCC	4LCC	5LCC	6LCC	1FCC	2FCC	3FCC	4FCC	5FCC	6FCC
A	196	1050	262	262	96	6	6	6	6	6	6	6	6	6	6	**6**	**6**
B	2802	1373	259	259	97	9	48	50	52	52	68	9	48	50	52	52	68
C	16098	5177	–	794	99	26	26	26	57	57	57	26	26	26	57	57	57
D	2709	228	83	83	96	28	28	28	28	28	28	28	28	28	28	28	28
E	5303	1000	233	233	98	83	83	83	83	83	83	83	83	83	83	83	83
F	848	3562	700	700	97	63	64	75	75	75	75	63	64	75	75	75	75
G	33994	5573	–	818	99	37	42	48	56	56	56	37	42	48	56	56	**56**
H	2676	5399	1984	1984	99	5	*7*	7	7	–	–	5	7	**8**	**12**	–	–
I	1528	1450	767	767	94	43	43	43	–	–	–	43	43	43	44	44	44
J	4892	551	169	169	95	34	36	38	73	73	73	34	36	38	73	73	73
K	21839	3298	41	41	99	88	99	99	99	99	99	88	99	**99**	**99**	**99**	**99**
L	15672	6370	–	1855	98	42	46	46	46	46	46	42	46	46	46	46	46
M	42442	5265	–	522	99	*29*	30	30	30	30	–	29	**30**	**30**	–	**91**	**93**

Figure 14. Precision of MCC, *k*LCC and *k*FCC (%)

Precision. To measure the precision of the different algorithms, we follow the same approach as in Section 3.3: We consider CIC to have 0% precision and SIC (or SMC) to have 100% precision. The results are shown in Figure 14: The first column contains the amount of chopping criteria (= amount of chops done) and the next three columns give the average size of CIC, SIC and SMC chops. The first thing we see is that there are four cases where it was impossible to complete SIC in the given time boundaries.

The simple MCC algorithm is surprisingly precise: normally it has around 96% precision and is always more precise than 6LCC or 6FCC. This shows that the main source of imprecision in chopping is the loss of the same-level property, which is kept by MCC. The other experiences are in a kind of contrast to the experiences with the slicing algorithms: All other algorithms than SIC, SMC and MCC are much less precise and with increased *k* the precision of *k*LCC and *k*FCC is only increasing slowly.

Speed. The most important observation while comparing the runtimes of SIC and SMC is that SMC is always *much* faster. As an extreme, in test case I SMC only needs 1% of the runtime of SIC. The MCC algorithm is not only surprisingly precise but also fast. For the larger test cases it may be much faster than any of the other algorithms.

The comparison of *k*LCC with *k*FCC in terms of runtime reveals the same results as for *k*LCS and *k*FCS: In many cases *k*FCC is much faster than *k*LCC and is less likely to suffer combinatorial explosion. Also, increased *k* may sometimes result in lower runtime.

In Section 3.4 we saw that SIS is normally not slower than CIS. This is not the case with chopping: In only three test cases CIC is slower than SIC, and in five test cases CIC is even faster than SMC. This is due to the much higher complexity of chopping with summary edges in comparison to slicing with summary edges. If we compare the speed of *k*LCC and *k*FCC against SIC and SMC, we experience again different results as in slicing: nine test cases are faster in chopping with 1LCC or 1FCC than SIC and one is even faster than SMC.

The results for chopping are not as clear as in slicing, but they lead to the same recommendation: due to the high precision SMC is preferable to all other chopping algorithms. Sometimes the call string approaches are faster, but they have a poor precision.

6. Conclusions and Future Work

We have evaluated four different algorithms (two well known and two new) for slicing and six different algorithms (two well known and four new) for chopping. Our experiences can be summarized as follows:

- The two new specialized chopping algorithms SMC and MCC are better than previously known chopping algorithms (CIC, SIC). SMC is preferable to SIC, as it is much faster. The MCC algorithm is surprisingly fast and precise, in some cases an alternative to SMC.
- Both *k*-limited approaches may suffer from the possible combinatorial explosions of call strings.
- Higher precision has definitely a high influence on speed: it may cause lower runtimes due to a smaller set of visited vertices. But the higher precision is gained from an increased *k*-limit, the higher amount of generated call strings may also cause a much higher runtime.
- The use of summary information is the best solution (as long as it is possible). It is both precise and fast.
- If summary edges cannot be used, *k*FCS and *k*FCC are the algorithms of choice, because they do not suffer from combinatorial explosions due to recursion. In

	CIC	SIC	SMC	MCC	1LCC	2LCC	3LCC	4LCC	5LCC	6LCC	1FCC	2FCC	3FCC	4FCC	5FCC	6FCC
A	1,35	2,54	0,35	0,47	4,07	7,88	20,5	78,3	384	1964	3,67	5,63	11,3	14,4	15,4	15,8
B	32,4	242	11,8	10,7	88,2	108	185	371	1069	2663	87,8	106	171	296	511	379
C	1145	–	4691	451	2293	3389	4670	3113	2613	2101	2295	3381	4697	3106	2612	2107
D	9,60	31,1	3,44	5,60	20,0	29,6	45,5	68,9	65,0	49,5	20,0	29,7	45,1	68,6	64,9	49,5
E	59,7	511	18,0	30,4	85,0	125	223	787	2061	3581	85,2	125	234	746	2087	3640
F	20,3	14,6	3,45	5,58	40,9	154	159	226	451	530	41,1	155	160	227	454	532
G	3985	–	4172	1171	6647	8721	11884	16844	21892	28437	6486	8260	11081	15173	20264	28753
H	112	9098	153	75,4	420	993	2884	9253	–	–	309	1131	3623	17486	–	–
I	18,1	2003	21,1	13,3	51,3	177	1778	–	–	–	46,2	91,7	122	176	232	214
J	60,8	44,1	9,63	25,2	140	185	268	323	1218	7921	138	179	237	151	159	154
K	1038	247	29,6	183	1127	1460	3137	3846	3167	2472	1134	1476	3191	4104	3459	2858
L	1202	–	2358	609	2302	3134	3967	4285	4119	4082	2301	3134	3974	4288	4114	4099
M	4037	–	2045	957	8452	13062	19314	25121	29800	–	8465	14981	28793	–	22959	24958

Figure 15. Runtimes of *k*LCC and *k*FCC (sec.)

some cases, FCS or FCC is also more precise, however, this may lead to higher runtimes.

- Context-insensitive algorithms cause a loss of precision, which is much higher in chopping than in slicing.
- The context-insensitive algorithms CIS and CIC can be regarded as 0LCS/0FCS and 0LCC/0FCC.

Based on our results, it would be interesting to fully evaluate other problems of program analysis: First, what influence have prerequisites like pointer analysis onto precision and speed of the different slicing/chopping algorithms? (Some results for SIS are presented in [11, 12].) Second, how are applications that use slicing or chopping influenced by the different algorithms? Based on our technique to generate and solve path conditions [16], which makes intensive use of chopping, we will work on both questions in the future.

References

[1] G. Agrawal and L. Guo. Evaluating explicitly context-sensitive program slicing. In *Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[2] D. C. Atkinson and W. G. Griswold. The design of whole program analysis tools. In *Proc. Intl. Conference on Software Engineering*, 1996.

[3] D. C. Atkinson and W. G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proc. Intl. Conference on Software Maintenance*, 2001.

[4] L. Bent, D. Atkinson, and W. Griswold. A comparative study of two whole-program slicers for C. Technical Report CS2000-0643, Univer. of California at San Diego, 2000.

[5] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Lecture Notes in Computer Science, 892*. 1995.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3), 1987.

[7] S. B. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1), 1990.

[8] D. Jackson and E. Rollins. A new model of program dependences for reverse engineering. In *Proc. Symposium on the Foundations of Software Engineering*, 1994.

[9] J. Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, 1998.

[10] J. Krinke and G. Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, 40(11-12), 1998.

[11] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *ESEC / SIGSOFT FSE*, 1999.

[12] A. Orso, S. Sinha, and M. Harrold. Effects of pointers on data dependences. In *Proc. 9th Intl. Workshop on Program Comprehension*, 2001.

[13] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proc. Software Engineering Symposium on Practical Software Development Environments*, 1984.

[14] T. Reps and G. Rosay. Precise Interprocedural Chopping. In *Proc. Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.

[15] T. W. Reps, S. B. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proc. Second Symposium on the Foundations of Software Engineering*, 1994.

[16] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *Proc. Intl. Conference on Software Engineering*, 2002.

[17] B. G. Ryder, W. Landi, B. Philip, A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Prog. Lang. Syst.*, 2001.

[18] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[19] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4), 1984.