Evaluating Key Statements Analysis

David Binkley[1], **Nicolas Gold**, **Mark Harman**, **Zheng Li** and **Kiarash Mahdavi**
King's College London
CREST Centre
Department of Computer Science
Strand, London WC2R 2LS, England, United Kingdom.

Abstract

Key Statement Analysis extracts from a program, statements that form the core of the program's computation. A good set of key statements is small but has a large impact. Key statements form a useful starting point for understanding and manipulating a program.

An empirical investigation of three kinds of key statements is presented. The three are based on Bieman and Ott's principal variables. To be effective, the key statements must have high impact and form a small, highly cohesive unit. Using a minor improvement of metrics for measuring impact and cohesion, key statements are shown to capture about 75% of the semantic effect of the function from which they are drawn. At the same time, they have cohesion about 20 percentage points higher than the corresponding function. A statistical analysis of the differences shows that key statements have higher average impact and higher average cohesion ($p < 0.001$).

Keywords: Program Slicing, Principal Variables, Impact, Cohesion

1 Introduction

Key Statement Analysis (KSA) identifies statements that are *key* to a given module [9]. The key statements identified support quicker focus of attention on critical code [7]. This helps engineers perform the kinds of tasks that arise, for example, during software maintenance and program comprehension [6, 12, 13, 15].

The computation of key statements denotes a departure from traditional slicing, which seeks to extract an executable sub program. Our previous research has found that, at least within the static slicing paradigm, the extracted slices can be rather large; on average they are about one third of the whole program [3]. Such large slices may retain some value for applications in reuse and reverse engineering, but for the purpose of supporting human analysis and insight elicitation from source code, large chunks of extracted code are comparatively unhelpful.

[1]On sabbatical leave from Loyola College in Maryland.

By contrast, key statements are *not* executable subsets of programs. Rather, they are specific statements, through which the largest part of the program's dependence appears to flow. They can be thought of as high impact statements, or statements that capture something of the essence of the overall computation; at least, they can be said to capture the 'essence of dependence' by definition. This paper introduces a new algorithm for KSA, and evaluates its ability to locate statements that capture this 'essence of dependence' in terms of the key statements' cohesion and impact.

To be effective, KSA must identify statements of high impact that themselves form a small cohesive unit. To evaluate the *keyness* of the key statements from module m, two metrics are used. The first, *impact*, considers the influence of the key statements as compared to that of all the statements of m. A good set of key statements is expected to have an impact similar to that of m. The second metric, *cohesion*, considers the influence of each key statement compared to that of all the key statements taken collectively. All statements in a set of key statements are expected to have high impact; thus, the set will tend to have high cohesion.

KSA can extract key statements from a variety of software modules such as functions, classes, files, and concepts bindings [7]. This paper investigates the effectiveness of KSA applied to the functions as found in a collection of ten C programs. The study considers the following four research questions, analysis of which forms the four primary contributions of the paper:

1. Is the number of key statements considerably smaller than the number of statements found in the function from which they are extracted?
2. Do key statements have similar sized impact on the computation of the program as that of the overall function from which they are extracted?
3. Do key statements form a more cohesive unit than those of the function from which they are extracted?
4. Do (large) dependence clusters [4] affect key statement identification?

The rest of this paper is organized as follows: Section 2 describes background on program slicing, the operation that underlies KSA. This operation is described in Section 3. Then Section 4 lays out the design of the experiment including the two metrics, impact and cohesion, used in the evaluation. Section 5 presents and discusses the results of the study. The final section concludes the paper.

2 Background

This section describes the two kinds of static program slicing [16, 10] used in the paper: *backward slicing* and *forward slicing*. Both are computed as a reachability analysis over a program's *System Dependence Graph* (SDG) [10]. The vertices of an SDG are essentially the nodes of the program's control-flow graph. An SDG's edges represent the control and data-flow *dependences* between the vertices.

A backward slice of an SDG is taken with respect to one or more selected vertices of the SDG and includes the vertices for those parts of the program that potentially effect the behaviour of the selected vertex. A forward slice is also computed with respect to one or more SDG vertices; it includes those parts of the program potentially affected by the computation represented at the selected vertex. The following two definitions formalize these two kinds of slicing.

Definition 1 (Backward Slice [2])
A *backward slice* of program P taken with respect to one or more vertices V from P's SDG, denoted $\overleftarrow{\mathcal{S}}(P, V)$, includes those vertices denoting computations that transitively effect the computation represented at V. The corresponding program elements form an executable subprogram, *S*, such that *S* behaves identically to *P* with respect to the sequence of values computed at each of the statements represented in V.

Definition 2 (Forward Slice [10])
A *forward slice* of program P taken with respect to one or more vertices V from P's SDG, denoted $\overrightarrow{\mathcal{S}}(P, V)$, includes those statements and predicates of P that are transitively affected by the computation represented at V.

In this paper, backward slices will be constructed int*ra*procedurally, while forward slices will be constructed int*er*procedurally. This is done to capture by simple intraprocedural backward slicing, within a procedure/function of interest, the global whole–program forward impact of that procedure. The work described in this paper seeks to use slicing to capture the (hopefully few) key statements that denote the largest part of this whole–program impact.

3 Key Statement Analysis

This section summarizes the key statement identification process [9, 7], which is based on the observation that statements in multiple backward slices contribute to multiple computations; thus, they are potential *key* statements within a program. To identify key statements the backward slices are taken with respect selected *principal variables*. The algorithm starts by identifying the *principal variables* for each function. Bieman and Ott defined these variables as those that play a "more important role in a program" [1, 11]. Formally, they defined variable v of function F is a principal variable iff

- v is a global variable assigned in F. The global principal variables is denoted $\mathbb{PV}_G$.
- v is used in an output statement (*e.g.*, print or write) in F. The output principal variables is denoted $\mathbb{PV}_O$.

In addition to $\mathbb{PV}_G$ and $\mathbb{PV}_O$, the empirical study also considers the set of principal variables $\mathbb{PV}_G \bigcup \mathbb{PV}_O$.

In the experiments the sets $\mathbb{PV}_G$ and $\mathbb{PV}_O$ are extracted from the program's SDG. The set of global variables potentially modified by a call to a function f, is easily extracted because each such global variable has a special *global-formal-out* vertex in the subgraph representing f. The set $\mathbb{PV}_O$ is also easily determined from the SDG as each output variable appears in an *actual-in* vertex (*i.e.*, is used as an actual parameter). The set $\mathbb{PV}_O$ includes those variables found in the actual-in vertices of calls to output functions such as printf and write.

Figure 1 presents the algorithm used to identify key statements based on an algorithm proposed by Harman et al. [7]. It takes as input a function f from a program P and a set of principal variables p. It computes the set of key statements, denoted $\mathbb{KS}$. The algorithm upon which that of Figure 1 is based computes the key statements as those found in the intersection of the backward slices taken with respect to the final use of each principal variable. In the figure, Finaluse(f, v) is used to denote the final use of variable v of function f.

This paper introduces an improved KSA algorithm that maintains the same impact and cohesion while reducing the size of the set of key statements. It exploits the observation that statements in the forward slice of a key statement can be removed from the set of key statements. Note that a function f may have no key statements.

Example. Figure 2 shows an example of the key statement computation based on the set of principal variables $\mathbb{PV}_O$. The example function computes the area and volume of a cylinder given its radius. From Lines 8 and 9, $\mathbb{PV}_O$ includes {area, volume}. The intersection of the backward slices taken with respect to the final uses of these two principal variables includes only Line 4 (shown boxed in Figure 2). Thus, Line 4 is the sole key statement. Here the intuition behind this statement's *keyness* is that the variable under_surface contributes to the computation of *both* area and volume variables.

4 Evaluation

A 'good' set of key statements ought to have a large impact while itself being a small cohesive group. Two metrics

```
function KSA (Program P, Function f, PVs p)
  returns: a set of Statements from f
{
  for each variable v in p
    let slice_i = ←S(P, Finaluse(f, v))
  endfor
  let KS = Statements(f) ∩ {⋂_i slice_i}
  for each (remaining) statement s_i in KS
   for each key statement s_j ≠ s_i in →S(f, s_i)
       remove s_j from KS
   endfor
  endfor
  return KS
}
```

Figure 1. Key Statement Analysis Algorithm for identifying those statements that effect all of a function's principal variables.

Line	void cylinder(int r)
1	{
2	diameter = 2*r;
3	perimeter = PI * diameter;
4	under_surface = PI * r * r;
5	side_surface = perimeter * h;
6	area = 2 * under_surface + side_surface;
7	volume = under_surface * h;
8	printf("Area is %f\n", **area**);
9	printf("Volume is %f\n", **volume**);
10	}

Figure 2. An example key statement computation in which the computation of principal variables area **and** volume **are both influenced by the computation of** under_surface **at Line 4, making it a key statement.**

are used to evaluate the *key-ness*: Impact and Cohesion. This section introduces and motivates these two. It then formalizes the measurement techniques used in the experiment and finally outlines the experimental design.

Following the work of Black [5], and Yau and Collofello [17] where forward dependence analysis is used to identify the ripple effect of a change, forward slicing is used in the definition of the two metrics as a measure of the impact of a statement. Roughly, the larger the forward slice, the larger the impact of the vertex (or set of vertices) from which the slice was computed. Thus, forward slicing can be used to measure the impact of one or more statements.

In the following, the forward slice notation, $\overrightarrow{S}(P,S)$, is overloaded such that when S is a single statement it is assumed to be the singleton set containing that statement; where S is a function it is taken to denote the set of all statements found in the function.

4.1 Metric 1: Impact

Impact measures the relative *influence* of a set of key statements S compared to the *influence* of the function from which they were drawn. Semantically, this *influence* can be defined as those statements computing with a *tainted* value when the statements if S introduce the taint [14]. A safe approximation to this influence can be computed using the forward slice taken with respect to the set of SDG vertices that represent the statement in S.

For a single statement s from function F of program P, Impact is the ratio of the size of the forward slice taken with respect to s to the size of the forward slice taken with respect to the statements of F:

$$\text{Stmt-Impact}(s,F,P) = \frac{|\overrightarrow{S}(P,s)|}{|\overrightarrow{S}(P,F)|}$$

For a set of statements, the average impact is used in the evaluation. Given a set of (key) statement S from function F of program P Impact(S,F,P) is defined as follows:

$$\text{Impact}(S,F,P) = \frac{1}{|S|}\sum_{s\in S}\text{Stmt-Impact}(s,F,P) \quad (1)$$

$$= \frac{1}{|S|}\sum_{s\in S}\frac{|\overrightarrow{S}(P,s)|}{|\overrightarrow{S}(P,F)|}$$

$$= \frac{\sum_{s\in S}|\overrightarrow{S}(P,s)|}{|S|\times|\overrightarrow{S}(P,F)|}$$

Because any set of key statements S is always a subset of the function from which they are drawn, F, the value of Impact(S,F,P) ranges between zero and one, where zero occurs when there are no key statements and one when the slice on each key statement is identical to the slice taken with respect to the entire function. Conceptually, the higher the value of Impact the more *key* the key statements. In the sequel Impact is used in place of Impact(S,F,P) when the statements, function, and program are clear from the context.

4.2 Metric 2: Cohesion

Whereas impact measures the outward influence of the key statements, Cohesion measures their inward connectedness. It compares the influence of each key statement to the influence of all the key statements taken collectively. This is similar to Bieman and Ott's notion of *Coverage* [1] and Weiser's notion of *Overlap* [16]. High cohesion is obtained when each of the key statements has a similar influence. For a fixed impact, the range of cohesion values is from 1/n (n is the number of key statements), where each key statement has a completely separate influence, to one,

when each key statement has exactly the same influence. Combined high Cohesion means the key statements have similar influence and high Impact means that they have a large influence.

Cohesion is computed as the ratio of the average slice size for the key statements to the size of the slice taken with respect to all the key statements collectively. More formally, for the set of key statements S from program P, Cohesion(P, S) is defined as follows:

$$\mathsf{Cohesion}(P,S) = \frac{\frac{1}{|S|}\sum_{s\in S}|\overrightarrow{\mathcal{S}}(P,s)|}{|\overrightarrow{\mathcal{S}}(P,S)|} \quad (2)$$

$$= \frac{\sum_{s\in S}|\overrightarrow{\mathcal{S}}(P,s)|}{|S|\times|\overrightarrow{\mathcal{S}}(P,S)|}$$

As with Impact, Cohesion is used in place of Cohesion (P, S) then the program and set of key statements are clear from the context.

The definition of Cohesion improves on Weiser's original slice-based metric Overlap [16] when the intersection of the set of slices is empty. Weiser's metric is defined as the average ratio of the size of the intersection of all slices to the size of each slice. When the intersection is empty, this metric takes the value zero. The difference is illustrated by the three examples shown in Table 1.

In the table, each column represents a slice, in which a '1' denotes a statement in the slice and a '0' a statement not in the slice. In Example 1, the three slices cover all six statements without any pairwise-overlap; in Example 2, the three slices cover only four of the statements, but SL_1 and SL_2 completely overlap; finally, in Example 3, three slices cover the same two statements and fully overlap. Weiser's definition of Overlap has the same value for Examples 1 and 2 and thus does not distinguish between, the no overlap and partial overlap cases.

By contrast, Cohesion, as defined above, distinguishes these two examples. At the same time, when Overlap imposes an order as it does between both Examples 1 and 2 when compared to Example 3, then Cohesion imposes the same order.

4.3 Measurement

While commonly used, Lines of Code (LoC) forms a somewhat crude measure owing to the lack of a standard definition and thus the influence that formatting style has on most definitions can be large. For this reason the empirical investigation uses SDG vertices, which provide a more consistent measure. However, to simplify the presentation, the examples will continue to refer to source lines rather than the associated vertices.

4.4 Experimental Design

The experimental design first presents the programs studied and then describes the source code analysis tools used. The ten C programs employed as subjects are described in Table 2. The table provides each program's size in LoC, SDG vertices, and the number of user defined functions. It also provides a brief description of each subject. The penultimate column separates the five programs known to be free of large dependence clusters from the five known to contain large dependence clusters.

The analysis constructs the SDG for each program and then slices it using CodeSurfer, Grammatech's deep-structure analysis tool [8]. The API for CodeSurfer includes functions for backward and forward slicing as well as access to the mapping from SDG vertices to the source code. In addition to slicing, the SDG vertices associated with global variables and function parameters are used to compute the sets $\mathbb{PV}_G$ and $\mathbb{PV}_O$.

5 Results and Discussion

This section first discusses the results related to the principal variables discovered. It then considers the key statements identified from the three kinds of principal variables and compares their Impact and Cohesion. This is followed by a comparison of the number of key statements compared to the number of statements in to the function from which they are extracted. The section finally returns to the four research questions from the introduction and interprets empirical results in terms of these questions.

5.1 Principal Variables

Table 3 presents the number of functions in each program and then the number that contain each of the three kinds of principal variables. In total 373 of 1,444 functions contain principal variables. The first observation evident from the data is that there are a large number of functions without any principal variables as Bieman and Ott define them [1]. These functions, by definition, will have no key statements. Inspection of the 1,071 functions without principal variables reveals that 749 of the 1,071 return a result computed from their input parameters. While not a *variable* per se, this suggests that one direction for future investigation is the consideration of additional kinds of principal 'variables' to augment those defined by Bieman and Ott. For example, expressions returned by a function.

Also evident from the table, the programs space and oracolo2 define no global variables, leaving $\mathbb{PV}_G$ empty. Finally, $\mathbb{PV}_G$ appears to be smaller for subject programs without large dependence clusters. This is potentially explained by the likelihood of a correlation between programs defining a large number of global variables and those with large dependence clusters.

In total there are 2,494 principal variables in the ten programs studied. These include 953 global principal variables and 1,541 output principal variables. Figure 3 shows three scatter plots, one for each kind of principal variable. Each plot shows the number of principal variables per function

Table 1. Example Cohesion **computation illustrating how** Cohesion **provides a range of values in cases where Weiser's overlap metric (upon which** Cohesion **is based) does not. At the same time** Cohesion **preserves the ordering when overlap identifies one.**

	Example 1			Example 2			Example 3		
Statement	SL_1	SL_2	SL_3	SL_1	SL_2	SL_3	SL_1	SL_2	SL_3
1	1	0	0	1	1	0	1	1	1
2	1	0	0	1	1	0	1	1	1
3	0	1	0	0	0	0	0	0	0
4	0	1	0	0	0	0	0	0	0
5	0	0	1	0	0	1	0	0	0
6	0	0	1	0	0	1	0	0	0
Weiser's Overlap	$\frac{1}{3}(\frac{0}{2}+\frac{0}{2}+\frac{0}{2})=0$			$\frac{1}{3}(\frac{0}{2}+\frac{0}{2}+\frac{0}{2})=0$			$\frac{1}{3}(\frac{2}{2}+\frac{2}{2}+\frac{2}{2})=1$		
Cohesion	$\frac{1}{3}(\frac{2}{6}+\frac{2}{6}+\frac{2}{6})=\frac{1}{3}$			$\frac{1}{3}(\frac{2}{4}+\frac{2}{4}+\frac{2}{4})=\frac{1}{2}$			$\frac{1}{3}(\frac{2}{2}+\frac{2}{2}+\frac{2}{2})=1$		

Table 2. Experimental Subjects.

Program	LoC	Vertices	Number of User Defined Functions	Large Dependence Cluster	Description
acct-6.3.2	3,204	9,775	50	No	Process monitoring tools
EPWIC-1	7,943	19,545	124	No	Image compressor
space	9,126	20,018	136	No	ESA space program
oracolo2	9,477	19,066	135	No	Antennae array set-up
CADP	12,762	48,577	450	No	Protocol toolbox
userv-1.0.1	6,616	95,076	114	Yes	Access control utility
indent-2.2.6	8,259	30,311	48	Yes	Text Formatter
bc-1.06	10,449	40,575	94	Yes	Calculator
diffutils-2.8	10,743	33,231	91	Yes	File comparison utilities
findutils-4.2.25	28,887	105,535	202	Yes	File finding utilities
Total	107,466	421,709	1,444		

Table 3. Counts of the number of functions that contain each of the three kinds of principal variables.

	Functions with Principal Variables			
Program	Total	$\mathbb{PV}_G \bigcup \mathbb{PV}_G$	$\mathbb{PV}_O$	$\mathbb{PV}_G$
acct-6.3.2	50	24	24	2
EPWIC-1	124	22	18	9
space	136	41	41	0
oracolo2	135	41	41	0
CADP	450	65	57	9
userv-1.0.1	114	38	29	11
indent-2.2.6	48	26	14	16
bc-1.06	94	44	16	36
diffutils-2.8	91	34	30	9
findutils-4.2.25	202	38	10	29
Total	1,444	373	280	121

on the y-axis and each program on the x-axis. In all three plots, most functions have between 1 to 25 principal variables. Comparing the charts for $\mathbb{PV}_O$ and $\mathbb{PV}_G$ (the upper two scatter plots), the number of output principal variables exist in a tighter range of values. Thus, over the ten programs considered, the output of variables is more consistent than the modification of global variables. The lower plot shows the distribution for $\mathbb{PV}_{Union}$. As the union of $\mathbb{PV}_O$ and $\mathbb{PV}_G$, this scatter plot shows where $\mathbb{PV}_O$ or $\mathbb{PV}_G$ dominate the union. Table 4 presents the mean and standard deviation for the *Union* data. It is interesting to note that the mean number of principal variables does not appear to be affected by the presence of large dependence clusters. However, the presence of large dependence clusters does appear to produce a higher standard deviation (at least with some programs).

This large standard deviation is partially caused by two clear outliers. Interestingly, as seen in the scatter plots, one each attributed to $\mathbb{PV}_O$ and $\mathbb{PV}_G$. The two, having over

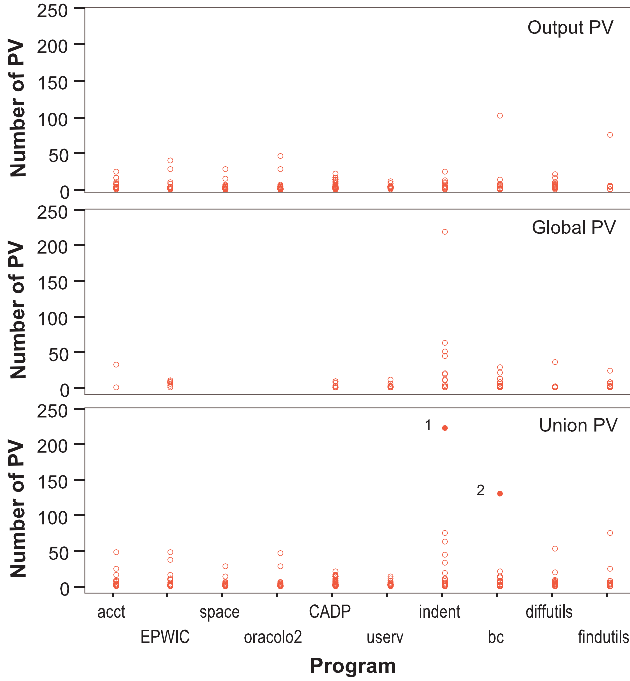

Figure 3. The number of principal variables per function for each of the three kinds of principal variables.

Table 4. The mean and standard deviation of the number of principal variables in $\mathbb{PV}_{Union}$.

Program	Functions	Mean	Std. Deviation
acct-6.3.2	24	7.13	10.49
EPWIC-1	22	8.59	12.23
space	41	3.12	4.85
oracolo2	41	3.90	8.21
CADP	65	5.89	5.22
userv-1.0.1	38	4.00	2.81
indent-2.2.6	26	21.08	45.46
bc-1.06	44	7.73	19.55
diffutils-2.8	34	6.35	9.20
findutils-4.2.25	38	5.42	12.53
Total	373	6.68	15.96

100 principal variables, are shown as the solid points and labeled '1' and '2' in the lower scatter plot. Looking at the source code, Point '1' comes from the main function of the program indent. Inspection of this function reveals that it is a large function with 1,657 LoC and 222 principal variables (219 of which come from $\mathbb{PV}_G$). As comments in the code indicate, the program's author chose to communicate values from main to the rest of the program using global variables; thus, creating a large pool of globals many of which are modified.

Point '2', the second outlier, is from the function yyparse which contains 1,190 LoC and 131 principal variables. Here, 102 are from $\mathbb{PV}_O$ and 29 from $\mathbb{PV}_G$. From an inspection of the source code for this Bison-generated parser, it is evident that there are a significant number output statements devoted to generating error messages associated with various error conditions that the parser may encounter.

5.2 Key Statements

Three types of key statements are computed based on the three kinds of principal variables: from $\mathbb{PV}_G$ *global key statements*, denoted $\mathbb{KS}_G$, from $\mathbb{PV}_O$ *output key statements*, denoted $\mathbb{KS}_O$, and from $\mathbb{PV}_G \bigcup \mathbb{PV}_O$ *union key statements*, denoted $\mathbb{KS}_{Union}$, respectively. Following some statistics on the number of each kind of key statement, the data related to the Impact and Cohesion are presented.

To gain an initial feel for the data, the size of the set of key statements is first compared to the size of the associated function. Here a lower value is preferred. For example, the ratio for 2 key statements in a 5 statement function is 40%, while 2 key statements in a 10 statement function is only 20%.

For each program, Figure 4 presents the average number of key statements per function expressed as percentage of the function's size. The numeric percentages are presented in Table 5. Overall, the average for each kind is similar, being around 25% of the function.

Two relevant observations about the key statements can be made from this graph. First, several of the averages for $\mathbb{KS}_G$ show what appears, at first sight, to be an interesting anomaly whereby the percentage for $\mathbb{KS}_{Union}$ is lower than that for $\mathbb{KS}_G$ (a similar pattern exists between $\mathbb{KS}_O$ and $\mathbb{KS}_{Union}$). This occurs when the backward slice taken with respect to each of the global principal variables in $\mathbb{PV}_G$ includes similar statements, while the backward slice taken with respect to the variables in $\mathbb{PV}_O$ includes different statements. Thus, when considering $\mathbb{PV}_G \bigcup \mathbb{PV}_O$, a smaller set of key statements (*i.e.*, a smaller intersection) is identified. That this pattern is not more pronounced supports the notion that the key statements identified are true.

The second observation is that, for all three types of key statements, the absence of large dependence clusters appears to correspond to smaller sets of key statements. Furthermore, large dependence clusters seem to accompany lower variability in the average. This is most pronounced for $\mathbb{KS}_G$. This suggests that large dependence clusters are masking the variability seen in the cluster-free programs. This provides evidence that large dependence clusters affect KSA and further that they do so through global variables.

The remainder of this section considers the Impact and Cohesion for each of the three types of key statements. To begin with, for the key statements to truly form the nucleus of a function's computation then they should have an impact similar to that of the function from which they are extracted.

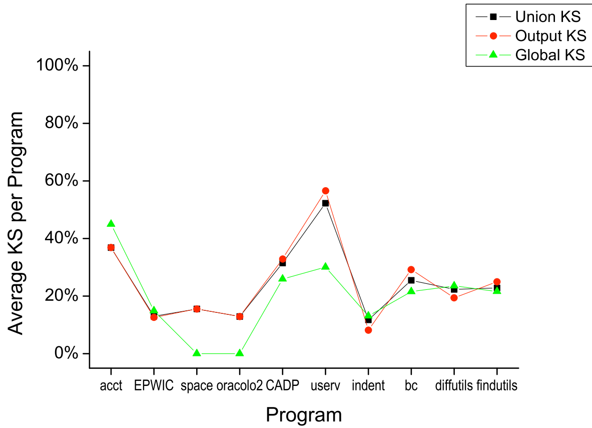

Figure 4. The average percentage of a function's statements that are key statements. (In this and similar charts, lines connecting points are included only as a visual aid; strictly speaking, no intermediate values exist.)

Table 5. The number of key statements as a percentage of function size.

Program	$\mathbb{KS}_{Union}$	$\mathbb{KS}_O$	$\mathbb{KS}_G$
acct-6.3.2	36.83%	36.83%	45.00%
EPWIC-1	12.99%	12.63%	14.91%
space	15.60%	15.60%	0.00%
oracolo2	12.92%	12.92%	0.00%
CADP	31.50%	32.93%	25.94%
userv-1.0.1	52.23%	56.60%	30.10%
indent-2.2.6	11.74%	8.21%	13.14%
bc-1.06	25.53%	29.21%	21.58%
diffutils-2.8	22.34%	19.46%	23.57%
findutils-4.2.25	22.87%	24.98%	21.63%
Total	25.17%	25.39%	21.90%

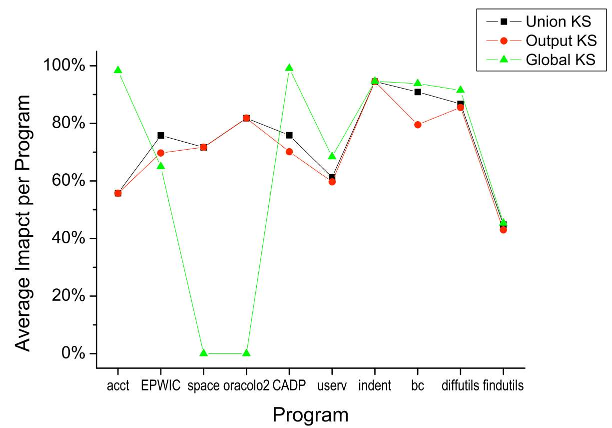

Figure 5. The average Impact for $\mathbb{KS}_{Union}$, $\mathbb{KS}_O$, and $\mathbb{KS}_G$.

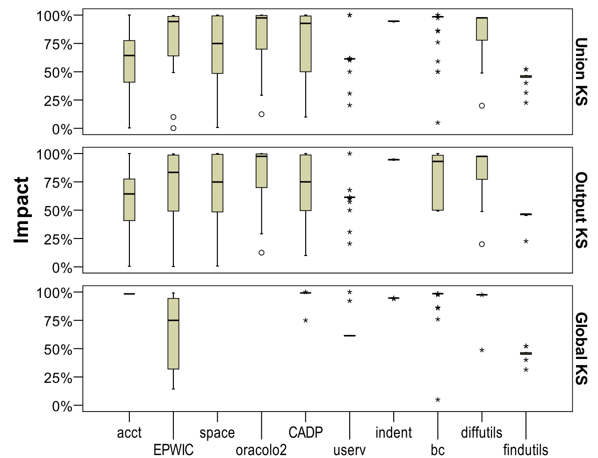

Figure 6. The boxplot showing the distribution of Impact for $\mathbb{KS}_{Union}$, $\mathbb{KS}_O$, and $\mathbb{KS}_G$. In addition to the box and whiskers, when they exist, outliers, denoted ○, and extreme cases, denoted * are shown with each boxplot.

Figures 5 and 6 present the average Impact and its distribution for each kind of key statement. In the experiment, Impact has the value 100% when the key statements have the same impact as the function from which they are extracted. As can be seen in Figure 5, $\mathbb{KS}_G$ tends to have a slightly higher Impact. Overall, the Impact of the key statements exceeds 70% of that of the associated function for 19 of the 28 points (discounting the two programs that have no globals). These include 7 of 10 for $\mathbb{KS}_{Union}$, 7 of 10 for $\mathbb{KS}_O$, and 5 of 8 for $\mathbb{KS}_G$.

The distribution of the Impacts, shown in Figure 6, clearly demonstrates the undesirable effect of large dependence clusters [4]. Consider first, the top row of Figure 6, which shows the box plots for $\mathbb{KS}_{Union}$. In the absence of large dependence clusters, the data have a reasonable distribution and only a few outliers. By contrast, with four of the five dependence–cluster–having programs, the 'box' in essence includes only the median value. These programs also have a considerable number of outliers and extreme values. For $\mathbb{KS}_O$ the data is better, but still shows the negative impact of large dependence clusters. There are fewer globals, but the data still point to dependence clusters generating a greater number of outlier and extreme values. Finally, more than the means, the median values show how in the first five programs (those without large dependence clusters), Impact is more consistent.

The data collected for the second metric, Cohesion is summarized in Figures 7 and 8. Cohesion was only computed for the functions with two or more key statements be-

cause Cohesion is, by definition, always 100% in the case of a single key statement.

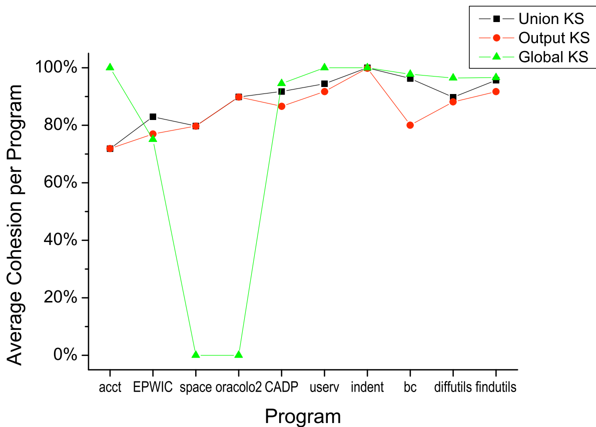

Figure 7. The average Cohesion for $\mathbb{KS}_{Union}$, $\mathbb{KS}_O$, and $\mathbb{KS}_G$.

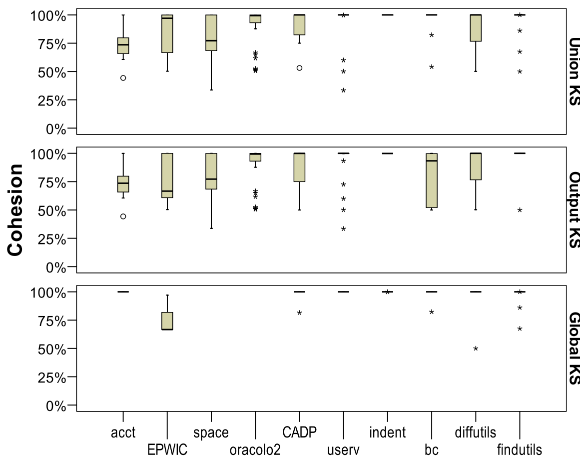

Figure 8. The boxplot showing the distribution of Cohesion for $\mathbb{KS}_{Union}$, $\mathbb{KS}_O$, and $\mathbb{KS}_G$. In addition to the box and whiskers, when they exist, outliers, denoted $\circ$, and extreme cases, denoted * are shown with each boxplot.

Higher Cohesion comes from higher overlap between the forward slices taken with respect to key statements. As seen in Figure 7, Cohesion for $\mathbb{KS}_{Union}$, $\mathbb{KS}_O$, and $\mathbb{KS}_G$ is always greater than 75%. This shows that each type of principal variable tends to generate a highly cohesive set of key statements.

For programs with large dependence clusters, there is a high probability that a given key statement will be within a large cluster and consequently have the same influence as other key statements that are also within the cluster. As such, the Cohesion for such programs is expected to be similar and higher than that of programs free from large dependence clusters. The data in Figure 7 bear this out. In particular, the points for the rightmost five programs are higher than those on the left.

Similar to the box plots for Impact, the box plots for Cohesion show how the presence of dependence clusters all but remove any spread in the data. With the union data, the distribution of Cohesion in four of the programs with dependence clusters is essentially 100%.

In summary, the data for Impact shows that the key statements capture a significant portion of the influence of a function. For almost three quarters of the functions, this was over 70% of the influence. In addition, the key statements have high Cohesion that often, in particular in the presence of large dependence clusters, measures 100%. Having demonstrated in this section, the high Impact and Cohesion of these statements, the next section compares these values to those of the corresponding function.

5.3 Key Statements and their Associated Function

The previous section compared the Impact and Cohesion of the three kinds of key statements with each other. This section compares the Impact and then the Cohesion of the key statements with the Impact and Cohesion of the function from which they are drawn. Because the same metric is computed for the matched pair of key statements and the function from which these statements are drawn, a Wilcoxon Matched-Pairs Signed Ranks Test procedure was applied to test the significance of the difference. In this case, the null hypothesis is there is no difference between the metric for the key statements and associated function.

To begin with, Figure 9 compares the Impact of the key statements from $\mathbb{KS}_{Union}$ taken from function F, Impact($\mathbb{KS}_{Union}, F, P$) that of F itself, Impact(F, F, P). In the figure $\mathbb{KS}_{Union}$ is used as representative of the three. Recall that, in Section 4, for a set of statements S, Impact(S, F, P) is defined as the average impact of the elements of S as compared to the impact of F; thus, Impact(F, F, P), is not always 1, but rather the average impact of the statements from F compared to the Impact of all the statements of F taken collectively. As visually apparent in Figure 9, the key statements always have higher Impact.

Table 6 includes the results of the statistical analysis. For each kind of key statement, it presents the standardized signed-ranks difference (Z) and statistical significance p-value (Sig. (2-tailed)). If the p-value shown in the final column less than 0.05, each null hypothesis can be rejected and the difference taken as statistically significant.

The first three rows of Table 6 concern Impact. For all three kinds of key statements, the p-value is less than 0.05; thus, the null hypothesis can be rejected. Hence, the difference between the Impact of each kind of key statement is significantly different from that of the correspond-

ing function. Visually, Figure 9 suggests that the impact for $\mathbb{KS}_{Union}$ is higher than that of the associated function.

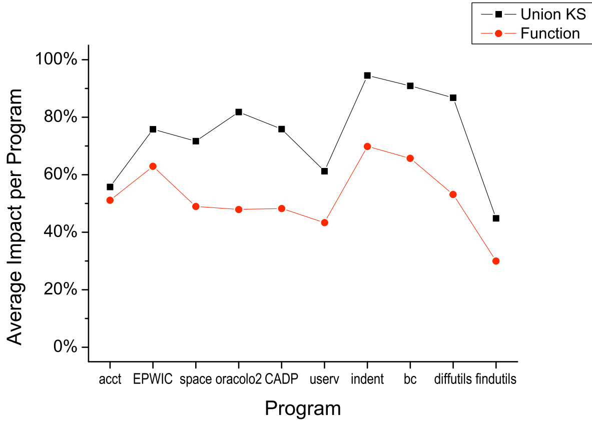

Figure 9. Average Impact of key statements and their associated functions.

Table 6. Wilcoxon Matched-Pairs Signed Ranks Test comparing key statements and their associated function.

	Z	Sig.(2-tailed)
Impact		
$\mathbb{KS}_{Union}$ - Function$_{Union}$	12.819	< 0.001
$\mathbb{KS}_{O}$ - Function$_{O}$	10.618	< 0.001
$\mathbb{KS}_{G}$ - Function$_{G}$	8.515	< 0.001
Cohesion		
$\mathbb{KS}_{Union}$ - Function$_{Union}$	11.920	< 0.001
$\mathbb{KS}_{O}$ - Function$_{O}$	10.303	< 0.001
$\mathbb{KS}_{G}$ - Function$_{G}$	6.791	< 0.001

Turning to Cohesion, for a set of key statements S, Cohesion (P, S) is compared to that of their associated function F, Cohesion (P, F). Figure 10 presents the comparison between the Cohesion for $\mathbb{KS}_{Union}$, which is again used as representative of the three. It is clear that the key statements always have higher Cohesion than the associated functions.

As with Impact, a Wilcoxon Matched-Pairs Signed Ranks Test was used to compare the significance of the difference between the Cohesion (P, S) and Cohesion(P, F). The null hypothesis is that there is no difference in Cohesion. The last three rows of Table 6 include the results for the three types of key statements. For Cohesion, the p value is less than 0.05; thus, the null hypothesis is rejected in each case and the difference taken to be statistically significant. This provides the evidence that the key statements of $\mathbb{KS}_{Union}$ have higher Cohesion than associated function.

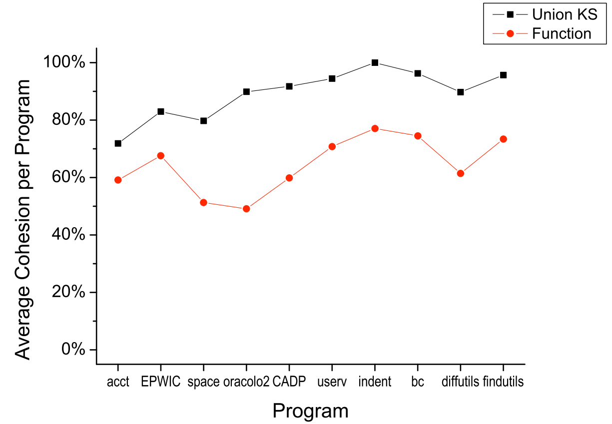

Figure 10. Comparison of the average Cohesion for $\mathbb{KS}_{Union}$ and the associated functions.

5.4 Summary of Results

Having presented the data gathered in the empirical study, this section concludes by returning to the four research questions:

1. *Is the number of key statements considerably smaller than the function from which it is extracted?*
   The data presented in Table 5 presents the average number of key statements in $\mathbb{KS}_{Union}$, $\mathbb{KS}_{O}$, and $\mathbb{KS}_{G}$ as 25.2%, 25.4%, and 21.9%, respectively. This data, visually supported by Figure 4, shows that for all the three kinds of principal variables, a considerable reduction is achieved.

2. *Do key statements have similar impact on the computation of the program as the function from which they are extracted?*
   The data presented in Figure 5 shows that for 19 of 28 cases the Impact of the key statements exceeds 70% of that of the associated function. Furthermore, from Figure 9 and the statistics in Table 6, the average impact of the key statements is higher. Thus, the parts of program impacted by a function but not its key statements are, in some sense, less consequential computations.

3. *Do key statements form a more cohesive unit than those of the function from which they are extracted?*
   Figure 7 shows how the key statements computed for each of the three types of principal variable have similar cohesion. Furthermore, as shown in the last three rows of Table 6 and visually apparent in Figure 10, key statements have a higher Cohesion than the statements of the function from which they are drawn.

4. Finally, *Do (large) dependence clusters affect key statement identification?*
   There is clear support for an affirmative answer to

this question. Starting with Table 3, the general distribution of the principal variables hints at a difference: dependence-cluster free programs tend to have a greater proportion of their principal variables in $\mathbb{PV}_O$, while programs with large dependence clusters tend to have a larger proportion of their principal variables in $\mathbb{PV}_G$. In Figure 4 the average number of principal variables is visually higher in the five programs that contain large dependence clusters. In Figure 5 (and the following box plots), the variability in Impact is higher for programs with large dependence clusters. Finally, large dependence clusters produce more cohesive sets of principal variables, as seen in Figure 7. Thus, large dependence clusters have a clear and largely negative impact on key statement identification.

6 Threats to Validity

There are potential threats to the external and internal validity of this investigation. The only significant threat to external validity is the possibility that the selected programs are not 'typical' programs. For example, most of the programs are open-source. As mature tools were used, the only significant threat to internal validity comes from construct validity, where variables do not adequately capture the concepts they are supposed to measure. To help alleviate this concern, mature well-studied metrics were used.

7 Summary and Future Work

This paper empirically examines the value of Key Statement Analysis (KSA) starting from the principal variables as defined by in the work of Bieman and Ott [1, 11]. KSA used is an improvement over previous approaches to KSA [9, 7]. The keyness of the identified statements is measured using their Impact and Cohesion. The results indicate that the identified key statements have high Impact and Cohesion and thus represent the core of a function's computation.

Key statements are computed as an optimized intersection of the intraprocedural static backward slices on Bieman and Ott's 'principal' variables. The results reveal that this simple intra procedural backward analysis can capture a significant proportion of the forward impact of the function within which the intraprocedural analysis is performed.

Future work will consider an evaluation of different starting points in the KSA algorithm. For example, the value returned by a function call.

8 Acknowledgements

This research work is supported by EPSRC Grant GR/T22872/01, GR/R71733/01. The authors also wish to thank GrammaTech Inc. (http://www.grammatech.com) for providing CodeSurfer, Greg Rothermel for providing some of the case study programs and Simon Poulding for valuable discussions about statistical analysis. Author order is alphabetical.

References

[1] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, Aug. 1994.

[2] D. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 3(1-4):31–45, 1993.

[3] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, 16(2):1–32, 2007.

[4] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In 21st *IEEE International Conference on Software Maintenance*, pages 177–186, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.

[5] S. E. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13:263–279, 2001.

[6] R. E. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[7] N. Gold, M. Harman, D. Binkley, and R. Hierons. Unifying program slicing and concept assignment for higher-level executable source code extraction. *Software Practice and Experience*, 35(10):977–1006, 2005.

[8] Grammatech Inc. The codesurfer slicing system, 2002.

[9] M. Harman, N. Gold, R. Hierons, and D. Binkley. Code extraction algorithms which unify slicing and concept assignment. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, pages 11 – 21, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[10] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[11] L. M. Ott. Using slice profiles and metrics during software maintenance. In *Proceedings of the* 10th *Annual Software Reliability Symposium*, pages 16–23, 1992.

[12] S. C. Shaw, M. Goldstein, M. Munro, and E. Burd. Moral dominance relations for program comprehension. *IEEE Trans. Softw. Eng.*, 29(9):851–863, 2003.

[13] B. Shneiderman. Measuring computer program quality and comprehension. *International Journal of Man-Machine Studies*, 9(4):465–478, 1977.

[14] G. A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[15] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[16] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[17] S. S. Yau and J. S. Collofello. Design stability measures for software maintenance. *IEEE Transactions on Software Engineering*, 11(9):849–856, Sept. 1985.