Ranking Software Inspection Results using Execution Likelihood [*§]

Cathal Boogerd
Software Evolution Research Lab
Delft University of Technology
The Netherlands
c.j.boogerd@ewi.tudelft.nl

Leon Moonen
Software Evolution Research Lab
Delft University of Technology and CWI
The Netherlands
Leon.Moonen@computer.org

Abstract

Static checking tools are useful as an automated software inspection step that can easily be integrated in the development cycle and assist in creating high quality, reliable and secure code. However, an often quoted disadvantage of these tools is that they generate an overly large number of warnings, including many false positives due to their approximate analysis techniques. This information overload effectively limits their usefulness.

In this paper we present ELAN, a technique that helps the user prioritize the information generated by a software inspection tool, based on computing the likelihood that execution reaches the locations for which warnings are reported. This analysis is orthogonal to ranking techniques known from literature, such as severity levels and statistical analysis to filter false positives. We evaluate feasibility of our technique using a number of open source and Philips-specific case studies and assess the quality of our predictions by comparing them to actual values obtained by profiling software embedded in a TV.

1. Introduction

Software inspection [14] is widely recognized as an effective technique to assess and improve software quality and reduce the number of defects [28, 20, 39, 27, 40]. Software inspection involves carefully examining the code, design, and documentation of software and checking them for aspects that are known to be potentially problematic based on past experience.

In this paper, we focus on tools that perform automatic inspection. Such tools are interesting since automatic detection of defects and anomalies allows early (and repeated) detection of project deterioration and helps to ensure software quality, security and reliability. Early feedback enables early corrections, thereby lowering development costs and increasing chances of success.

Most defect detection techniques are built around some form of static analysis of the code. In its simplest form this can be the warnings generated by a compiler set to its pedantic mode. In addition, various dedicated static program analysis tools are available that assist in defect detection and writing reliable and secure software. A well-known example is the C analyzer LINT [23]. These tools form a complementary step in the development cycle and have the ability to check for more sophisticated program properties than can be examined using a normal compiler; moreover, they can often be customized, and as such benefit from specific domain knowledge.

However, such static analyses come with a price: in the case that the algorithm cannot ascertain whether the source code at a given location obeys a desired property or not, it will make the safest approximation and issue a warning, regardless of the correctness. Consequently, this conservative behavior can lead to *false positives*, incorrectly signaling a problem with the code. Kremenek and Engler [25] observed that program analysis tools typically have false positive rates ranging between 30–100%. In addition, the increased scrutiny with which the code is examined can lead to an explosion in the *number* of warnings generated, especially when the tool is introduced later in the development process or during maintenance, when a significant code base already exists.

To cope with the large number of warnings, users have to resort to using all kinds of (manual) filtering processes, often based on the perceived impact of the underlying fault. Even worse, our experience indicates that the information overload often results in complete rejection of the tools, especially in cases where the first few defects reported by the tool turn out to be false positives.

Goal In this paper, we set out to help the user of automated code inspection tools to deal with this information overload. Instead of aiming to improve a particular defect detection technique to reduce its false positives, we strive for a *generic prioritization approach* that can be applied to the results of any software inspection tool and assists the user in selecting

[*] This work has been carried out in collaboration with Philips Semiconductors as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs.

[§] This paper extends our earlier work on prioritizing software inspection results [5] with a detailed Philips specific case study which invesigates the correlation between execution likelihoods predicted by our approach and actual values obtained by dynamic profiling of software embedded on a TV.

the most relevant warnings.

To this end, we propose *ELAN* (Execution Likelihood ANalysis), a technique which orders inspection results based purely on a prediction of the likelihood that execution reaches the reported defect. The rationale behind this approach is that the violating code needs to be *executed* in order to trigger the undesired behavior. As such, the execution likelihood can be considered a *contextual measure* of severity, rather than the severity based on defect types which is usually reported by software inspection tools. The ELAN algorithm is kept simple on purpose: scalability is an issue, as we want to be able to prioritize inspection results for large industrial systems.

Industrial Context The context of this work is the TRADER project in cooperation with Philips Semiconductors, in which we investigate and develop methods and tools for ensuring reliability of consumer electronics devices.

Modern consumer electronics such as mobile phones, televisions, and audio/video media centers, increasingly rely on embedded software for their operation. In the past, functionality of such devices was mostly implemented in hardware, but nowadays the features of these devices are made easily extensible and adaptable by means of software. As a consequence, the amount of software embedded in consumer electronics has grown tremendously.

For example, a modern television contains several million lines of C code and this amount is growing rapidly with new functionality, such as electronic program guides, increased connectivity with other devices, audio and video processing and enhancements, and support for various video encoding formats. During the development process, this code is routinely inspected using QA-C, one of the leading commercial software inspection tools currently on the market. Nevertheless, the developers have experienced problems handling the information overload mentioned earlier which motivated the research described in this paper.

Overview The remainder of this paper is organized as follows: section 3 presents our prioritization approach, section 4 details the ELAN algorithm. An overview of the experiments can be found in section 5. For a survey of related work, discussion of more advanced algorithms, and an in-depth treatment on the experiments we refer to the full paper.

2. Related Work

Automatic Code Inspection There are a number of tools that perform some sort of automatic code inspection. The most well-known is probably the C analyzer Lint [23] that checks for type violations, portability problems and other anomalies such as flawed pointer arithmetic, memory (de)allocation, `null` references, and array bounds errors. LClint and splint extend the Lint approach with annotations added by the programmer to enable stronger analyses [12, 13]. Various tools specialize in checking security vulnerabilities. The techniques used range from lightweight lexical analysis [33, 38, 31] to advanced and computationally expensive type analysis [22, 18], constraint checking [35] and model checking [8]. Some techniques deliberately trade formal soundness for performance in order to scale to the analysis of larger systems [11, 6, 17] whereas others focus on proving some specific properties based on more formal verification techniques [2, 7, 9].

Several commercial offerings are available for conducting automated automatic code inspection tasks. Examples include QA-C,[1] K7,[2] CodeSonar,[3] and Prevent.[4] The latter was built upon the MECA/Metal research conducted by Engler et al. [42, 11]. Reasoning[5] provides a defect analysis *service* that identifies the location of potential crash-causing and data-corrupting errors. Besides providing a detailed description of defects found, they report on *defect metrics* by measuring a system's defect density and its relation to industry norms.

Ordering Inspection Results The classic approach most automated code inspection tools use for prioritizing and filtering results is to classify the results based on *severity levels*. Such levels are (statically) associated with the *type* of defects detected; they are oblivious of the actual code that is being analyzed and of the location or frequency of a given defect. Therefore, the ordering and filtering that can be achieved using this technique is rather crude. Our approach is based on the idea that this can be refined by taking into account certain properties of the identified defect with respect to the complete source code that was analyzed.

A technique that is more closely related to our approach, is the z-ranking technique by Kremenek and Engler [25]. They share our goals of prioritizing and filtering warnings based on their properties with respect to analyzed code but do so based on the frequency of defects in the results. Their approach aims to determine the likelihood that a given warning is a false positive. It is based on the idea that, typically, the density of defects in source code is low. Thus, when checking source code for a certain problem, there should be a large number of locations where that check is not triggered, and relatively few locations where it is triggered. Conversely, if a check results in many triggered locations and few non-triggered ones, these triggers are more likely to be false positives. This notion is exploited by keeping track of success and failure frequencies, and calculating a numeric score by means of a statistical analysis. The warning reports can then be sorted accordingly.

Besides severity levels and z-ranking, we are not aware of any other work that deals with ordering inspection results.

Static Profiling Static profiling is used in a number of compiler optimizations or worst-case execution time (WCET)

[1] `www.programmingresearch.com`
[2] `www.klocwork.com` [3] `www.grammatech.com`
[4] `www.coverity.com` [5] `www.reasoning.com`

analyses. By analyzing program structure, a prediction is made as to which portions of the program will be most frequently visited during execution. Since this heavily depends upon branching behavior, some means of branch prediction is needed. This can range from simple and computationally cheap heuristics to expensive data flow based analyses such as constant propagation [24, 29, 3], symbolic range propagation [32, 4, 26], or even symbolic evaluation [15]. Although there has been a lot of research on branch prediction, there are only a few approaches that take this a step further and actually compute a complete static profile [41, 36]. For branch prediction, these use heuristics similar to the ones we employ. The difference with respect to our work is that they try to estimate the, computationally expensive, execution *frequency*, whereas we compute execution *likelihood* which is less involved. Moreover, in contrast with these approaches, we do analyze the complete program, but perform analysis in a *demand-driven* manner: i.e., only for the locations associated with the warning reports.

Testability Voas et al. [34] define *software testability* as "the probability that a piece of software will fail on its next execution during testing if the software includes a fault". They present a technique, dubbed *sensitivity analysis*, that analyses execution traces obtained by instrumentation and calculates three probabilities for every location in the program. Together they give an indication of the likelihood that a possible fault in that location will be exposed during testing. The first of these three, *execution probability*, is similar to our notion of execution likelihood, the chance that a certain location is executed. The other two are the *infection probability*, i.e. the probability that the fault will corrupt the data state of the program, and the *propagation probability*, the likelihood that the corrupted data will propagate to output and as such be observable.

Although the concepts involved are very similar to our own, the application and analysis method differ greatly: a location that is unlikely to produce observable changes if it would contain an error should be emphasized during testing, whereas we would consider that location to be one of low priority in our list of results. In addition, Voas approximates these probabilities based on dynamic info whereas we try to make predictions purely statically. Finally, infection- and propagation probability apply to locations that contain faulty code, while the inspection results we are dealing with may also reveal security vulnerabilities and coding standard violations that do not suit these two concepts.

3. Approach

The approach we propose for prioritizing code inspection results is based on the workflow depicted in figure 1. The process consists of the following steps (starting at the top-left node):

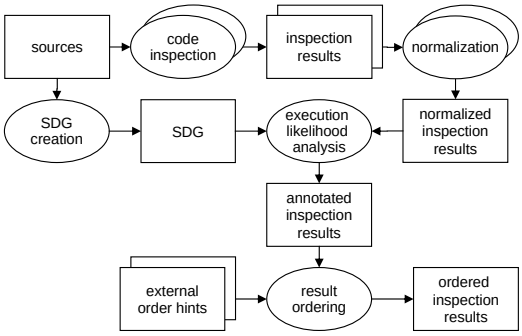

Figure 1. ELAN prioritization approach

1. The source code is analyzed using some code inspection tool, which returns a set of inspection results.

2. The inspection results are normalized to the generic format that is used by our tools. The format is currently very simple and contains the location of the warning in terms of file and line number, and the warning description. We include such a normalization step to achieve independence of code inspection tools.

3. We create a graph (SDG) representation of the source code that is inspected. Nodes in the graph represent program locations and edges model control- and data flow.

4. For every warning generated by the inspection tool, the following steps are taken:

   (a) Based on the reported source location, the analyzer looks for the corresponding vertex in the graph.

   (b) It then proceeds to calculate the execution likelihood of this location/vertex by analyzing the structure of the graph, and annotates the warning with this likelihood.

5. The inspection results are ordered by execution likelihood, possibly incorporating external hints such as severity levels or z-ranking results.

System Dependence Graph Central to our approach is the computation of the execution likelihood of a certain location in the program. In other words, we need to find all possible execution paths of the program that include our location of interest, and make predictions for the conditions (or branches) found along those paths. To this end, we use the program's *System Dependence Graph* (SDG) [21], which is a generalization of the *Program Dependence Graph* (PDG).

In short, the PDG is a directed graph representing control- and data dependences within a single routine of a program (i.e. *intraprocedural*), and the SDG ties all the PDGs of a program together by modeling the *interprocedural* control- and

data dependences. A PDG holds vertices for, amongst others, assignment statements, control predicates and call sites. Conditions in the code are represented by one or more *control points* in the SDG. In addition, there is a special vertex called *entry vertex*, modeling the start point of control for a function. In the remainder, we will use the terms vertex, program point and location interchangeably. The edges between nodes represent the control- and data dependences. Our approach currently does not consider information from dataflow analysis, so we limit our discussion to control dependences. The most relevant causes for these dependencies are:

- there is a control dependence between a predicate vertex v and a second vertex w if the condition at v determines whether execution reaches w;
- there is a control dependence between a function's entry point and its top-level statements and conditions;
- there is a control dependence between a call site and its corresponding function entry point.

Clearly, we can find all possible acyclic execution paths by simply traversing the SDG with respect to these control dependences. However, traversing the complete SDG to find all paths to a single point is not very efficient. To better guide this search, we base our traversals on *program slicing*.

Slicing The slice of a program P with respect to a certain location v and a variable x is the set of statements in P that may influence the value of variable x at point v. Although we are not actually interested in dataflow information, this slice must necessarily include all execution paths to v, which is exactly what we are looking for. By restricting ourselves to control flow information, we can rephrase the definition as follows: the *control-slice* of v in P consists of all statements in P that determine whether execution reaches v. Calculating the execution likelihood of v is now reduced to traversing all paths within this slice, and this forms the basis for the algorithm discussed next.

4. Execution Likelihood ANalysis

This section will introduce the algorithm calculating the execution likelihood for a single program point. Given the SDG of a project, computation entails traversing the graph in reverse postorder, obtaining probabilities by predicting branch probabilities and combining all the paths found from the main entry point to our point of interest. For simplicity, we assume that the project contains a main function that serves as a starting point of execution, although this is not a strict prerequisite, as we will see later on.

Basic algorithm To calculate the execution likelihood e_v for a vertex v in a programs SDG P, we perform the following steps:

1. Let B_v be the control-slice with respect to v. The result is a subgraph of P that consists of the vertices that influence whether control reaches v.
2. Starting from the main entry point v_s, perform a depth-first search to v within B_v, enumerating all the paths (sequences of vertices) to v. This is a recursive process; traversal ends at v, then the transition probabilities are propagated back to v_s, where the transition probability $p_{w,v}$ is the a posteriori probability that execution reaches v via w. For any given vertex w visited within the traversal, this is calculated in the following manner:
   (a) If w is v, skip all steps, $p_{v,v}$ is 1.
   (b) For every control-dependence successor s of w in B_v, determine $p_{s,v}$
   (c) Determine the probability that control is transferred from w to any of its successors s. We do this by first grouping the probabilities $p_{s,v}$ by the label of the edge needed to reach s from w. E.g., when w represents the condition of an if statement, we group probabilities of the true and false branches of w together. For every group we determine the probability that at least one of the paths found is taken, we denote this set S_w.
   (d) If w is not a control point, there will be just one element in S_w, and its probability is $p_{w,v}$.
   (e) If w is a multiway branch (switch), all its cases are thought to be equally likely, and as such $p_{w,v}$ can be obtained by adding all probabilities in S_w and dividing them by the number of cases.
   (f) If w represents the condition of an if statement, we consider this a special case of a switch: each of its branches is thought equally likely, so $p_{w,v}$ is obtained by adding both elements of S_w and dividing them by 2.
   (g) If w is a loop, it is assumed that the loop will be executed at least once. S_w consists of one element representing the probability of the loop body, and $p_{w,v}$ is taken to be equal to this value.
3. When recursion returns at our starting point v_s, we have calculated the transition probability from v_s to v, which is our desired execution likelihood e_v.

As stated earlier, the algorithm is written with the computation of execution likelihood in a project with a single startpoint of execution (i.e. the main function) in mind. If we are dealing with a partially complete project, or we are in any other way interested in the execution likelihood with a different starting point, the approach can be easily modified to suit that purpose. Instead of using program slicing, we use a related operation called *program chopping*. The chop of a program P with respect to a source element s and a target element t gives us all

Project Name	ncKLoC	# nodes in SDG	# non-global nodes in SDG	# CPoints in SDG	CC/fn			CPoint/LoC			ELAN accuracy (w. 5% margin)	
					avg	stddev	max	avg	stddev	max	"is executed"	"is not executed"
Uni2Ascii	3	10368	5022	138	29.3	33.4	1004	0.23	0.21	0.91	100 - 100	67 - 80
Chktex	8	30422	10149	769	9.8	18.2	412	0.19	0.13	0.85	92 - 93	47 - 45
Link	17	88766	33647	3009	5.6	9.4	358	0.16	0.15	1.0	92 - 94	33 - 37
Antiword	24	119391	35371	2787	8.0	11.0	596	0.13	0.10	0.77	100 - 100	69 - 69
Lame	53	93812	39937	3673	7.9	25.9	904	0.12	0.19	3.25	100 - 100	68 - 69
Infratst	67	600575	92659	7678	5.9	12.1	567	0.22	0.34	6.4	50 - 38	73 - 73

Table 1. Case study programs, their metrics and accuracy of ELAN predictions w.r.t. runtime data

elements of P that can transmit effects from s to t. Notably, the chop of P between its main entry point and any other point v is simply the slice of P with respect to v. Notice that if we let B_v be the chop with respect to v_s and v, we can take any starting point v_s and end up with the desired conditional execution likelihood.

While traversing the graph, transition probabilities for the paths taken are cached. As such, when traversing towards v, for a given $w \in B_v$ we only need to calculate $p_{w,v}$ once. This approach necessarily only works within one traversal, i.e. when computing the execution likelihood of one point, because the prequel to some subpath may differ between traversals. However, when computing the likelihood for multiple locations within one program in a row, it is likely that at least part of the traversal results can be reused. For any point v in a procedure f, we can split the transition probabilities into one from v_s to the entry point s_f of f, and the transition probability from s_f to v. Effectively, this means that for any point in f we only compute p_{v_s,s_f} once.

Another important contributor to performance is that our algorithm is *demand-driven*: it only computes execution likelihoods for locations of interest, instead of computing results for every location in the program. Given that the number locations with issues reported by an inspection tool will typically be much smaller than the total number of vertices in the graph, this is a sensible choice. Together with our deliberately simple heuristics, it forms the basis for a scalable approach.

Refined branch prediction heuristics To gain more insight into the performance/accuracy trade-off, we extend the algorithm with some of the branch prediction heuristics of Ball and Larus [1], used in the manner discussed by Wu and Larus [41]. They tested the heuristics empirically and used the observed accuracy as a prediction for the branch probability. For example, they observed that the value check heuristic predicts 'branch not taken' accurately 84% of the time. Therefore, when encountering a condition applicable to this heuristic, 16 and 84 are used for the 'true' and 'false' branch probabilities, respectively. Whenever more than one heuristic applies to a certain control point, the predictions are combined using the Dempster-Shafer theory of evidence [19], a generalization of Bayesian theory that that describes how several independent pieces of information regarding the same event can be combined into a single outcome.

The heuristics replace the simple conventions used in steps (d) through (g) discussed above. The behavior with regard to multiway branches has not been changed, and in cases where none of the heuristics apply the same conventions are used as before. A brief discussion of the application of the different heuristics follows below, their associated branch prediction probabilities can be found in table 2. The table lists the probability a condition that satisfies the heuristic will evaluate to *true*. We should remark that these numbers are based on empirical research on different programs [1] than used in our experiments. However, Deitrich et al. [10] provide more insight into their effectiveness and applicability to other systems (and discuss some refinements specific to compilers).

The refined heuristics are:

Loop branch heuristic: This heuristic has been modified to apply to any loop control point. The idea is that loop branches are very likely to be taken, similar to what was used earlier. The value is used as multiplier for probability of the body.

Pointer heuristic: Applies to a condition with a comparison of a pointer against `null`, or a comparison of two pointers. The rationale behind this heuristic is that pointers are unlikely to be `null`, and unlikely to be equal to another pointer.

Value check heuristic: This applies to a condition containing a comparison of an integer for less than zero, less than or equal to zero. This heuristic is based on the observation that integers usually contain positive numbers.

Loop exit heuristic: This heuristic has been modified to apply to any control point within a loop that has a loop exit statement (i.e. break) as its direct control predecessor. It says that loop exits in the form of break statements are unlikely to be reached as they usually encode exceptional behavior.

Return heuristic: Applies to any condition having a return statement as its direct successor. This heuristic works because typically, conditional returns from functions are used to exit in case of unexpected behavior.

Implementation ELAN has been implemented as a plugin for Codesurfer,[6] a program analysis tool that can construct dependence graphs for C and C++ programs. As our approach is

[6] `www.grammatech.com`

Heuristic	Probability	Heuristic	Probability
Loop branch	0.88	Pointer	0.40
Value check	0.16	Loop exit	0.20
Return	0.28		

Table 2. Heuristics and associated probabilities

based on the SDG, the way in which this graph is constructed directly affects its outcome, especially in terms of accuracy. It should be noted, therefore, that the graphs both have missing dependences (false negatives) and dependences that are actually impossible (false positives). For example, control- or data dependences that occur when using setjmp/longjmp are not modeled. Another important issue is the accuracy of dependences in the face of pointers, think for example of modeling control dependences when using function pointers. To improve this accuracy, a flow insensitive and context insensitive points-to algorithm [30] is employed to derive safe information for every pointer in the program.

5. Experiments

This section reports on experiments designed to evaluate the accuracy and performance of our technique. Recall that the execution likelihood predictions are used in ranking different locations; we therefore compare rankings based on predictions with rankings based on measurements in actual program runs. Apart from this *ordering*, we also evaluate accuracy by checking correspondence of the actual prediction *values* with their measured counterparts. Both experiments are discussed in section 5.1; in section 5.2 we analyze performance by timing the aforementioned experiments and relate the analysis speed to the size of the SDG involved.

5.1. Correlating predictions with runtime

We will start our discussion by reviewing the experimental setup. Table 1 lists some source code properties for the different case studies, respectively the size in lines of code (LoC, not counting comment or empty lines), the size of the SDG in vertices, the size of the SDG without vertices for global parameters, the total number of control points in the program, the cyclomatic complexity (per function), and the control points related to number of lines in the program (again per function). The apparent discrepancy between the size in LoC and the size of the SDG in vertices can be largely attributed to Codesurfer's modeling of global variables in the SDG. These are filtered out in the 'non-global' column, which shows a better correspondence with the size in LoC.

The programs were picked such that it would be easy to construct 'typical usage' input sets, and automatically perform a large number of test runs. For every case, at least 20 different test runs were recorded. Every one of the programs was subjected to the following steps:

1. Build the project using Codesurfer. This involves the normal build and building the extra graph representations used by our technique.

2. Build the project using gcc's profiling options, in order to obtain profiling information after program execution.

3. Run the ELAN algorithm for every control-point vertex in the project. This will give a good indication of analysis behavior distributed throughout the program (since it approximates predictions for every basic block). Moreover, this step was performed twice: with and without branching heuristics.

4. Gather a small dataset representing typical usage for the project, and run the program using this dataset as input. For all the program locations specified in step 3, determine the percentage of runs in which it was visited at least once. This last step uses gcov, which post-processes the profile data gathered by gcc's instrumentation.

5. Create two sets of program locations, the first sorted by prediction, the second by actual usage, and compare them using Wall's unweighted matching method [37]. This will give us a *correlation score* for different sections of the two rankings.

To illustrate this correlation score, consider the following example: suppose we have obtained the two sorted lists of program locations, both having length N, and we want to know the score for the topmost m locations. Let k be the size of the intersection of the two lists of m topmost locations. The correlation score then is k/m, where 1 denotes a perfect score, and the expected score for a random sorting will be m/N. In our experiments, scores were calculated for the topmost 1%, 2%, 5%, 10%, 20%, 40% and 80%. The correlation scores for the different programs can be found in table 3, where the first number in every cell is the score without the use of heuristics, and the second is obtained using heuristics.

When considering the ranking data in table 3, we will denote the upper $n\%$ of the ranking as *block n*. We are first and foremost interested in the correlation scores of the top-most blocks of locations. Typically, a randomized ranking would produce correlation scores of 0.01, 0.02 and 0.05 for these top blocks, which is easily outperformed by the ELAN technique. However, there is a problem in using these correlation scores for evaluating the accuracy of the ranking. Consider Antiword, for example: in our test series, 3.8% of the locations involved were always executed, which means that the two topmost blocks in our table will consist entirely of locations with measured value 1. There is no way to distinguish between these locations, and as such, no way to compare rankings. Even if predictions were perfect, locations in our predicted block 1 could be anywhere in block 2 of the other ranking. In

Portion	Antiword	Chktex	Lame	Link	Uni2Ascii	TV
1%	.45 - .45	.25 - .50	.14 - .14	.22 - .22	1.0 - 1.0	.13 - .13
2%	.44 - .44	.22 - .33	.11 - .11	.11 - .11	1.0 - 1.0	.11 - .10
5%	.61 - .61	.30 - .26	.25 - .24	.30 - .29	.67 - .67	.15 - .15
10%	.50 - .46	.34 - .32	.26 - .26	.59 - .55	.29 - .29	.17 - .17
20%	.42 - .45	.24 - .26	.34 - .34	.57 - .57	.67 - .67	.35 - .35
40%	.52 - .49	.40 - .41	.46 - .45	.60 - .61	.57 - .57	.63 - .64
80%	.78 - .79	.76 - .76	.83 - .83	.83 - .83	.79 - .79	.96 - .96

Table 3. Correlation scores

Portion	Antiword	Chktex	Lame	Link	Uni2Ascii	TV
1%	.91 - 1.0	1.0 - 1.0	.71 - .75	.87 - .91	1.0 - 1.0	.30 - .33
2%	.73 - .89	.89 - .89	.35 - .47	.89 - .94	1.0 - 1.0	.23 - .23
5%	.68 - .83	.83 - .78	.35 - .36	.89 - .90	1.0 - 1.0	.16 - .16
10%	.65 - .72	.72 - .72	.40 - .43	.89 - .90	1.0 - 1.0	.13 - .14
20%	.49 - .59	.62 - .61	.39 - .40	.81 - .82	.87 - .87	.08 - .08
40%	.40 - .40	.41 - .41	.30 - .29	.70 - .72	.57 - .57	.06 - .06
80%	.28 - .47	.47 - .46	.35 - .35	.57 - .56	.44 - .44	.09 - .09

Table 4. Average measured execution likelihood

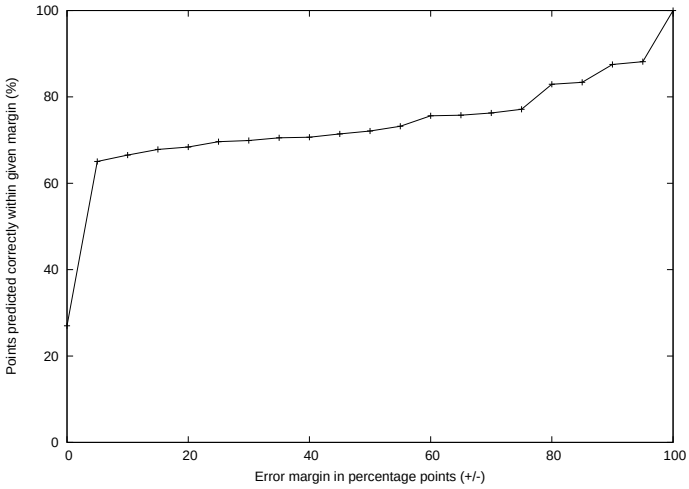

Figure 2. ELAN overall accuracy

practice this will be less of a problem since these values are used in conjunction with, e.g., the reported severity. Still, it shows that is important not to look at just the ordering, but also at the distribution of actual values.

In table 4, the average execution likelihood for all the locations in a certain block are shown; again, the values represent the results without heuristics and with heuristics, respectively. We took the ranking based on our *computed* likelihood and calculated the average *measured* execution likelihood. The values shown in table 4 cannot be compared amongst programs, as they depend on the runtime behavior of the individual program. For example, consider the differences between values for Antiword, which has 3.8% of its locations always executed, and Chktex, which has 33.5% of its locations always executed. What does matter, however, is the distribution within one program: we expect the locations ranked higher to have a higher actual execution likelihood, and, with some exceptions, exactly this correlation can be observed here.

Finally, we illustrate overall accuracy by means of the relation between error margins and accurate predictions, using a diagram similar to the ones in [26]. However, instead of the number of branches predicted accurately, the figure displays the number of locations for which the execution likelihood was predicted accurately within the given error margin. Moreover, the set of test programs used for evaluation differs.

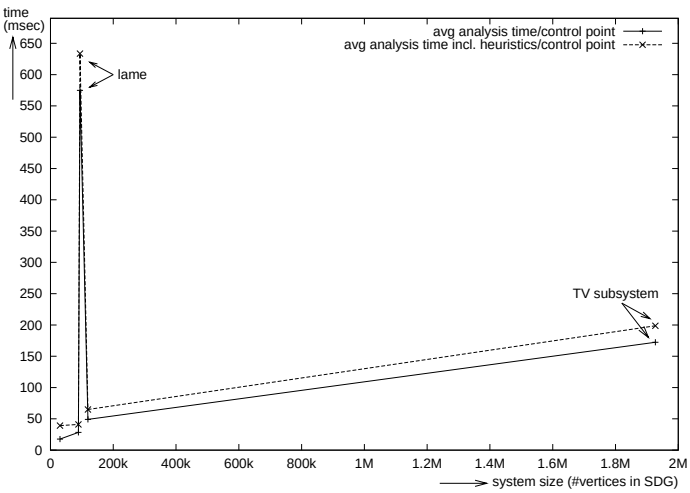

Figure 3. ELAN performance for various case studies

5.2. Performance measurements

The benchmark set used in the previous section consists of programs of different size (cf. table 1), which helps to understand the scalability of the approach. Recall our algorithm, which computes a slice, traverses the subgraph obtained, and derives predictions for conditions. This signifies the importance of the size of the SDG, rather than the number of KLoC. This relationship is illustrated in figure 3. Timing measurements were taken for every experiment in section 5.1, and we calculated the average time taken per location. All measurements were performed on a laptop with an Intel Pentium Mobile 1.6Ghz and 512Mb memory running MS Windows XP Pro.

Also, to understand the effect of the caching mechanism introduced in section 4, we performed the analysis for different subsets of the complete set of program points involved in the earlier experiments. Specifically, we took subsets of differing sizes (e.g. 10%, 20% of the original size) by sampling the original set uniformly, and measured the analysis running time for these subsets. The resulting measurements are displayed in figure 4. Typically, we expect a relation like this to be linear, and the fact that the slope of the curve is decreasing shows that the caching mechanism is indeed doing its job.

6. Evaluation

Accuracy When looking at the data of the accuracy experiments, perhaps most striking are the differences between tables 3 and 4. Even though locations with a higher execution likelihood in general seem to be ranked higher, the correlation scores resulting from the comparison with the ranking based on those measured values simply do not measure up. To understand this, we need to look at the locations that will end up high in the ranking: in our test set, the percentage of loca-

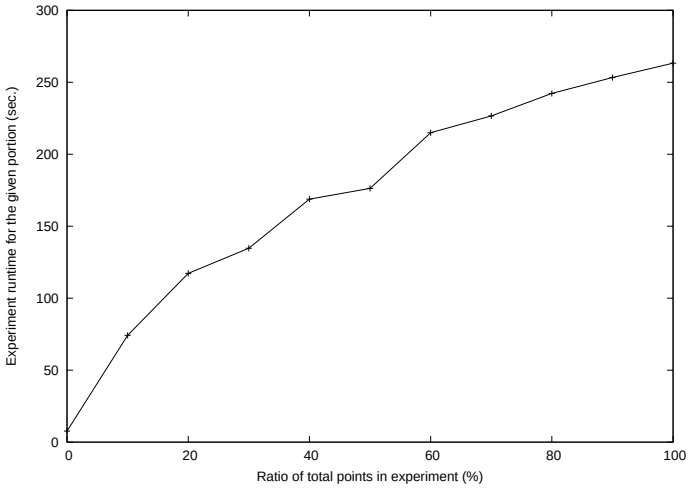

Figure 4. ELAN caching performance illustrated

tions that were always executed ranged from 4% to 34%. This typically means that, even though we may propagate many of the 'interesting' locations towards the top of the list, we cannot distinguish between those in the topmost regions of that list. This explains the seeming discrepancy between correlation score and average execution likelihood.

Another observation is that both accuracy tables, and especially table 4, show a drop in ranking accuracy for the Philips TV software. We believe this can be explained by its intensive use of function pointers. Even though we can statically predict pointer targets to some extent, we will usually end up with a *set* of possible targets, each of which is thought to be equally likely. As only one can be the true target, the algorithm will assign a higher likelihood to the other points than should be the case (and would be, if it were a direct call). A possible solution to this problem would be to make another pass over the source code, replacing the function pointers by direct function invocations. This is feasible because all the function pointers are explicitly stored in a reference table. Although the transformation will not result in an executable system, it is perfectly analyzable using the standard techniques we employ in our approach.

Bearing in mind that we use rather simple mechanics, the results diplayed in table 1 and figure 2 are promising. Notably, 65% of all the locations involved in the experiment were predicted correctly with an error margin of 5 percent point. Moreover, precision seems to be highest for locations with likelihood close to 0 or 1, and this makes the latter an interesting starting point for manual inspection.

Performance There are a number of observations that we can make regarding performance: first of all, the approach seems to scale well to larger software systems, where the version that uses the refined branch prediction heuristics is only slightly outperformed by the simpler one. It goes to show that including more sophisticated analyses can still result in a feasible solution.

Secondly, there is one program that does not conform to this observation: performance on Lame is significantly lower than on any of the others. Manual inspection revealed that the Lame frontend has a function that parses command-line arguments with a great number of short-circuited expressions. Because this occurs early on in the program, it affects many of the locations we are testing. Specifically, this means that computation of 25% of the locations involved requires traversal through this function, and 6.5% of the locations are within this function itself. In a follow-up experiment that left out the suspect function we found that the performance on the rest of the program was according to expectations (30 ms/location). This illustrates the fact that our evaluation experiment is actually somewhat negatively biased, since a relatively large number of locations were taken from a computationally expensive function. However, this kind of biased distribution is unlikely when ordering actual inspection results. In addition, such large short-circuited expressions are atypical for the type of software analysed in the Trader project, leaving little reason for concern at this time.

Finally, note that, because of the caching mechanism, the reported timings are actually worst-case approximations. Because the complete project is covered, adding more warning locations will lead to an efficient use of the cache and therefore a shortened computation time. For instance, one of the benchmarks, involving approximately 2600 points, took 137 seconds. Extending this experiment to an analysis of 13000 points increased the running time to slightly under 300 seconds: just over double the time for a fivefold increase in input, showing that prioritizing a large number of warnings remains feasible.

Orthogonality We remark that our approach is orthogonal to other prioritization and filtering techniques discussed in the related work. However, in combination with these approaches, ELAN can best be applied as final step because the filtering of the earlier stages in combination with our demand driven approach effectively reduces the amount of computations that will need to be done.

Applicability In our test set, we use programs that are one-dimensional in their tasks, i.e. perform one kind of operation on a rather restricted form of input. This limits issues related to the creation of appropriate test inputs, and allows us to focus on evaluating the approach itself. It does mean, however, that we must devote some time to the question how to generalize these results to other kind of programs.

The ELAN approach is based on information implicit in the control structure of the program, and as for the heuristics, in the way humans tend to write programs. This information will always be present in any program. However, there may be

parts of the control structure that are highly dependent on interaction or inputs. Fisher and Freudenberger observed that, in general, varying program input tends to influence which parts of the system will be executed, rather than influencing the behavior of individual branches [16]. This suggests that, typically, there are a number of highly data-dependent branches early on in the program, while the rest of the control structure is rather independent. For example, a command-line tool may have a default operation and some other modi of operation that are triggered by specifying certain command-line arguments. At some point in this program, there will be a switch-like control structure that calls the different operations depending on the command-line arguments specified. This control structure is important as it has a major impact on the rest of the program, and it is also the hardest to predict due to its external data dependence. However, this information (in terms of our example: which operation modi are most likely to be executed) is exactly the type of information possessed by domain experts such as the developers of the program. Therefore, the simple extension of our approach with a means to specify these additional (input) probabilities can further improve applicability to such situations.

7. Concluding Remarks

Contributions We present a method for the prioritization of software inspection results based on statically computing the likelihood that program execution reaches locations for which issues are reported, i.e., we prioritize code inspection results using static profiling. We discuss a novel *demand-driven* algorithm for computing execution likelihood based on the system dependence graph and we present and evaluate a number of optimizations that further increase accuracy and performance.

We investigate the feasibility of the described approach using a number of case studies in which a prototype tool was developed and applied to several open source software systems and software embedded in a Philips television. We investigate the correlation between our static predictions and actual execution data found by dynamic profiling and we report on the performance of our approach. This empirical validation shows that the approach is capable of correctly prioritizing program locations. In addition, the approach scales well to larger systems.

Future Work Together with Philips Semiconductors, we are preparing a larger case study, in which our technique will be evaluated during development and inspection of software to be embedded in a new digital TV.

Moreover, we want to experiment with a number of ideas to further improve our approach by incorporating more advanced program analysis techniques, such as range propagation [26], that are basically aimed at enabling better estimations of the outcomes of conditions. Also, they can be used to compute execution frequencies, which will benefit the ranking by better distinguishing locations at the top of the ranking. However, since such analyses typically come with additional computational costs, we want to investigate if the improved accuracy actually warrants the expenses involved.

References

[1] T. Ball and J. R. Larus. Branch prediction for free. In *ACM Conf. on Programming Language Design and Implementation (PLDI'93)*, pages 300–313, June 1993.

[2] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The blast query language for software verification.. In *11th Intnl. Static Analysis Symposium (SAS 2004)*, volume 3148 of *LNCS*, pages 2–18. Springer, 2004.

[3] D. Binkley. Interprocedural constant propagation using dependence graphs and a data-flow model. In *5th Intnl. Conf. on Compiler Construction (CC'94)*, volume 786 of *LNCS*, pages 374–388. Springer, 1994.

[4] W. Blume and R. Eigenmann. Demand-driven, symbolic range propagation. In *8th Intnl. Workshop on Languages and Compilers for Parallel Computing, (LCPC'95)*, volume 1033 of *LNCS*, pages 141–160. Springer, 1995.

[5] C. Boogerd and L. Moonen. Prioritizing software inspection results using static profiling. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2006.

[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exp.*, 30(7):775–802, 2000.

[7] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Softw. Eng.*, 30(6):388–402, 2004.

[8] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *9th ACM Conf. on Computer and Communications Security (CCS'02)*, pages 235–244. ACM, 2002.

[9] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *ACM Conf. on Programming Language Design and Implementation (PLDI'02)*, pages 57–68. ACM, 2002.

[10] B. L. Deitrich, B-C. Cheng, and W. W. Hwu. Improving static branch prediction in a compiler. In *18th Annual Intnl. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 214–221. IEEE, 1998.

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *4th Symposium on Operating Systems Design and Implementation*, pages 1–16, October 2000.

[12] D. Evans, J. Guttag, J. Horning, and Y.M. Tan. LCLint: A tool for using specifications to check code. In *2nd ACM Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[13] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, 2002.

[14] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. Jour.*, 15(3):182–211, 1976.

[15] T. Fahringer and B. Scholz. A unified symbolic evaluation framework for parallelizing compilers. *IEEE Trans. Parallel Distrib. Syst.*, 11(11):1105–1125, 2000.

[16] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *5th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, 1992.

[17] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *ACM Conference on Programming language design and implementation (PLDI'02)*, pages 234–245. ACM, 2002.

[18] J. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, Univ. of California, Berkeley, Dec. 2002.

[19] Shafer G. *A Mathematical Theory of Evidence*. Princeton Univ. Press, 1976.

[20] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[21] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[22] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *13th Usenix Security Symposium*, pages 119–134, Aug. 2004.

[23] S. C. Johnson. Lint, a C program checker. In *Unix Programmer's Manual*, volume 2A, chapter 15, pages 292–303. Bell Laboratories, 1978.

[24] J. Knoop and O. Rüthing. Constant propagation on the value graph: Simple constants and beyond. In *9th Intnl. Conf. on Compiler Construction (CC 2000)*, volume 1781 of *LNCS*, pages 94–109. Springer, 2000.

[25] T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *10th Intnl. Symposium on Static Analysis (SAS 2003)*, volume 2694 of *LNCS*, pages 295–315. Springer, 2003.

[26] J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *ACM Conf. on Programming Language Design and Implementation (PLDI'95)*, pages 67–78, 1995.

[27] A. A. Porter, H. Siy, A. Mockus, and L.G. Votta. Understanding the sources of variation in software inspections. *ACM Trans. Softw. Eng. Meth.*, 7(1):41–79, 1998.

[28] G. W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Softw.*, 8(1):25–31, 1991.

[29] S. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *6th Intnl. Conf. on Theory and Practice of Software Development (TAPSOFT'95)*, volume 915 of *LNCS*, pages 651–665. Springer, 1995.

[30] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 1–14, Jan. 1997.

[31] Secure Software. Rats: Rough auditing tool for security. `http://www.securesoftware.com/resources/tools.html`.

[32] C. Verbrugge, P. Co, and L.J. Hendren. Generalized constant propagation: A study in C. In *6th Intnl. Conf. on Compiler Construction (CC'96)*, volume 1060 of *LNCS*, pages 74–90. Springer, 1996.

[33] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conf. (ACSAC 2000)*. IEEE, Dec. 2000.

[34] J.M. Voas and K.W. Miller. Software testability: The new verification. *IEEE Softw.*, pages 17–28, 1995.

[35] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, Feb. 2000.

[36] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In *ACM Conf. on Programming Language Design and Implementation (PLDI'94)*, pages 85–96, 1994.

[37] D. W. Wall. Predicting program behavior using real or estimated profiles. In *ACM Conf. on Programming Language Design and Implementation (PLDI'91)*, pages 59–70, June 1991.

[38] D. Wheeler. Flawfinder. `http://www.dwheeler.com/flawfinder/`.

[39] D. A. Wheeler, B. Brykczynski, and Jr. R. N. Meeson, editors. *Software Inspection : An Industry Best Practice*. IEEE, 1996.

[40] C. Wohlin, A. Aurum, H. Petersson, F. Shull, and M. Ciolkowski. Software inspection benchmarking - a qualitative and quantitative comparative opportunity. In *8th Intnl. Software Metrics Symposium (METRICS 2002)*. IEEE, 2002.

[41] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *27th Annual Intnl. Symposium on Microarchitecture*, pages 1–11. ACM/IEEE, 1994.

[42] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *10th ACM Conf. on Computer and Communications Security*, 2003.