Delft University of Technology
Software Engineering Research Group
Technical Report Series

Supporting Software Inspection with Static Profiling

Cathal Boogerd and Leon Moonen

Report TUD-SERG-2009-022

TUD-SERG-2009-022

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Supporting Software Inspection with Static Profiling *

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

Leon Moonen
Simula Research Laboratory
Norway
Leon.Moonen@computer.org

August 23, 2009

Abstract

Static software checking tools are useful as an additional automated software inspection step that can easily be integrated in the development cycle and assist in creating secure, reliable and high quality code. However, an often quoted disadvantage of these tools is that they generate an inordinate number of warnings, including many false positives due to the use of 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 a demand-driven computation of the likelihood that execution reaches the locations for which warnings are reported. This analysis is orthogonal to other prioritization techniques known from literature, such as severity levels and statistical filtering to reduce false positives. We evaluate the feasibility of our technique using a number of case studies and assess the quality of our static estimates by comparing them to actual values obtained by dynamic profiling.

1 Introduction

Software inspection [17] is widely recognized as an effective technique to assess and improve software quality and reduce the number of defects [35, 23, 45, 33, 47]. 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. One of the advantages of

* This work has been carried out in the Software Evolution Research Lab at Delft University of Technology 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 under the BSIK03021 program.

software inspection is that the software can be analyzed even *before* it is tested. Therefore, potential problems are identified and can be solved early, and the cost of repairing a defect is generally acknowledged to be much lower when that defect is found early in the development cycle [5, 25].

Until recently, software inspections have been a formal and predominantly manual process which, although effective at improving software quality, proved to be labor-intensive and costly. The strict requirements often backfired, resulting in code inspections that were not performed well or sometimes even not performed at all. Therefore, various researchers and companies have started to address these issues and have developed techniques and tools that aim at supporting the software inspection process. We can distinguish two approaches: (1) tools that *automate* the inspection *process*, making it easier to follow the guidelines and record the results; and (2) tools that perform *automatic code inspection*, relieving the programmers of the manual inspection burden and supporting continuous monitoring of the software quality.

In this paper, we deal with the second category: tools that perform automatic code inspection, usually built around static analysis of the code. In its simplest form, such automatic inspection consists of 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 code. A well-known example is the C analyzer LINT [27]; others are discussed in the related work section. 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. This conservative behavior can lead to *false positives* hidden within the results, incorrectly signaling a problem with the code. Kremenek and Engler [31] observed that program analysis tools typically have false positive rates ranging between 30–100%. The problem is magnified by the large amount of warnings that these tools produce, resulting from more scrutinous examination of the source code than can be achieved with a typical compiler. Solving all reported issues can be especially daunting if the tool is introduced later in the development process or during maintenance, when a significant code base already exists.

Such issues have also been identified at NXP (formerly Philips Semiconductors), our industrial partner in the TRADER project, where we investigate and develop methods and tools for ensuring reliability of consumer electronics devices. Originally, the functionality of these devices was mostly implemented in hardware, but nowadays their features are made easily extensible and adaptable by means of software. 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, NXP reported that its developers have experienced problems handling the information overload mentioned earlier.

To cope with the large number of warnings, users resort to 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 tool, especially in cases where the first defects reported by the tool turn out to be false positives. Although previous research has addressed the latter issue [31], and tools often report severity levels based on the *type* of defect found, none of these approaches consider the *location* of the defect in the source code. This information can be seen as an additional indication of the possible impact of a fault, and thereby provides another means of ranking or filtering the multitude of warning reports.

The goal of this paper is to help users of automated code inspection tools deal with the aforementioned information overload. Instead of focusing on improving a particular defect detection technique in order 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 the most relevant warnings. To this end, we present a technique to compute a static profile for the software under investigation, deriving an estimate for the execution likelihood of reported defects based on their location within the program. The rationale behind this approach is that the violating code needs to be executed in order to trigger the undesired behavior. As such, execution likelihood can be considered a contextual measure of severity, rather than the severity based on defect types that is usually reported by inspection tools.

This paper extends our earlier work [6] with an improved presentation and assessment of our approach, an industrial case study into its feasability, costs and benefits and a detailed study into the accuracy of the static branch prediction techniques. The remainder of this paper is organized as follows: in Section 2, we will give an overview of related work in static profiling and ranking software inspection results. Section 3 describes how we use static profiling to rank software inspection results, while the source code analysis that produces the profile is discussed in Section 4. We present a number of case studies in Section 5, their results in 6, and evaluate our findings in Section 7. Finally, we conclude by stating contributions and future work in section 8.

2 Related Work

2.1 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 [27] 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 [15, 16]. Various tools specialize in checking security vulnerabilities. The techniques used range from lightweight lexical analysis [40, 46, 21] to advanced and computationally expensive type analysis [26, 22], constraint checking [42] and model checking [11]. Some techniques deliberately trade formal soundness for a reduction in complexity in order to scale to the analysis of larger systems [14, 9, 20] whereas others focus on proving some specific properties based on more formal verification techniques [2, 10, 12].

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. [49, 14]. 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.

2.2 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 [31]. 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 probability 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 locations 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.

Another possibility to deal with false positives is the history-based warning prioritization approach by Kim and Ernst [29]. In this approach, the version history and issue tracking system for a project are mined to determine which warnings were actually removed during a bug-fix. By means of this distinction,

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

a *true* positive rate for a class of warnings can be computed, which can in turn be used to rank them when analyzing new revisions of the project. In earlier work, Kim and Ernst suggested to extract the warning fix time from the version repository [28]. In this case, types of warnings that are always fixed quickly can be given priority over those that are rarely fixed.

2.3 Static Profiling

Static profiling is used in a number of compiler optimizations or worst-case execution time (WCET) analyses. By analyzing program structure, an estimation 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 more expensive data flow based analyses such as constant propagation [30, 36, 3], symbolic range propagation [39, 4, 32], or even symbolic evaluation [18]. Although there have been many studies on branch prediction, there are only a few approaches that take this a step further and actually compute a complete static profile [48, 43]. For branch prediction, these use heuristics similar to the ones we employ. Contrary to our estimation of the execution *likelihood*, the existing techniques compute execution *frequencies*, requiring additional loop iteration prediction mechanisms, typically implemented using expensive fix-point computations. Moreover, in contrast with these approaches, we perform analysis in a *demand-driven* manner: i.e., only for the locations associated with the warning reports we are trying to rank.

2.4 Testability

Voas et al. [41] 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 analyzes 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 information whereas we derive estimates purely statically. Finally, infection- and propagation probability apply to locations that contain faulty code, whereas the

inspection results we are dealing with may also indicate security vulnerabilities and coding standard violations that do not suit these two concepts.

3 Approach

This section describes how we apply static profiling to prioritize code inspection results. Although typically, a static profile for a given program gives execution frequencies for all program locations, we report the *execution likelihood* instead. The reason for this is that scalability of the approach is an important criterion, and the computation of frequencies, especially with regard to loops, requires more sophisticated analysis. This being the case, henceforth we will refer to our static profiling as Execution Likelihood ANalysis (ELAN), of which a detailed description is given in the next section.

We employ *static* program analysis instead of dynamic analysis because we want to be able to use our prioritization technique with automated software inspection tools early in the development process, i.e. before testing and integration. In addition, performing a dynamic analysis is less feasible in our application domain (embedded systems), both because it affects timing constraints in the system (observer effect), and because a complete executable system is not always available earlier during the development (e.g. due to hardware or test data dependences).

3.1 Process overview

An overview of the approach is depicted in figure 1. The process consists of the following steps (starting at the top-left node):

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 basic block, and its start-vertex in the graph.

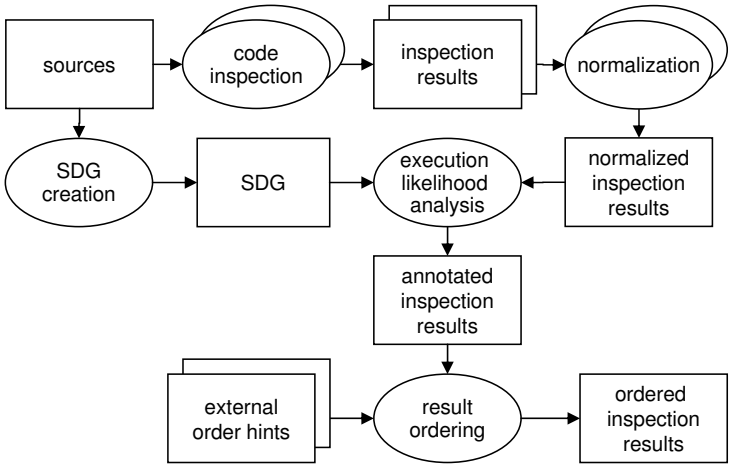

Figure 1: ELAN prioritization approach

(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.

3.2 Graph definition and traversal preliminaries

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. We can then estimate the execution likelihood of the given location by traversing the paths found and making predictions for the outcomes of all conditions (or branches) found along the way. To enumerate the paths we use the program's *System Dependence Graph* (SDG) [24], 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 PDG. In addition, there is a special vertex called *entry vertex*, modeling the start point of control for a function. In the remainder, when we use the term location this refers to a vertex in the PDG of a program. 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 dependences 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*.

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.

As mentioned before, the paths obtained this way are usually conditional, and we need an additional mechanism to produce static likelihood estimates for conditions. In the next section we elaborate on the graph traversal as well as discuss various static branch predictors, from simple assumptions to type-based heuristics.

4 Execution Likelihood Analysis

This section introduces the algorithm for calculating the execution likelihood of a single program point, defined as *the probability that the associated program statement is executed at least once in an arbitrary program run.* 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.

4.1 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* [34]. The chop of a program P with respect to a source element s and a target element t gives us all 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. Obviously, this approach can only work 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 of 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.

4.2 Refined branch prediction heuristics

To gain more insight into the speed/accuracy trade-off, we extend the algorithm with branch prediction heuristics introduced by Ball and Larus [1], applied in the manner discussed by Wu and Larus [48]. The latter 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 [37], 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 1. The table lists the probability that a condition which satisfies the heuristic will evaluate to *true*. The refined heuristics are:

Heuristic	Probability	Heuristic	Probability
Loop branch (LBB)	0.88	Pointer (PH)	0.40
Value check (OH)	0.16	Loop exit (LEH)	0.20
Return (RH)	0.28		

Table 1: Heuristics and associated probabilities

Project Name	ncKLoC	# nodes in SDG	# non-global nodes in SDG	# CPoints in SDG
Uni2Ascii	3	10368	5022	138
Chktex	8	30422	10149	769
Link	17	88766	33647	3009
Antiword	24	119391	35371	2787
Lame	53	93812	39937	3673
TV	67	1926980	119010	10079

Table 2: Case study programs and their metrics.

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.

We should remark that the numbers in Table 1 are based on empirical research on different programs [1] than used in our experiments. Deitrich et al. [13] provide more insight into their effectiveness and applicability to other systems (and discuss some refinements specific to compilers). We will see the influence of the heuristic probabilities used in our case in the experiments discussed in the next section.

5 Experimental setup

Over the course of the previous sections, various aspects of the approach have been discussed that are worth investigating. In particular, these are the accuracy of the branch prediction mechanism and the static profile, and the analysis time of the profiler. We will investigate these as follows:

IV1 The heuristics mentioned in Section 4 are calibrated for a testbed other than our own. We therefore repeat part of the experiment in [1], using dynamic profile data gathered from our testbed. This will give us prediction values for the non-loop heuristics that are fitted to our testbed.

IV2 Including the calibrated heuristics obtained in IV1, we have three different means of branch prediction: uniform (UP), heuristic (HP) and calibrated heuristic (CHP). In this investigation we will compare the static profiles computed using the different techniques with dynamic execution profiles for the testbed.

IV3 Finally, we measure the analysis time involved in computing the static profiles in IV2 and relate it to the size of the corresponding SDG.

The investigations have been performed using an implementation of ELAN as a plugin for Codesurfer,[6] a program analysis tool that can construct dependence graphs for C and C++ programs.

5.1 Selected cases

In our investigations we have used a small selection of programs which is shown in Table 2. The table provides source code properties for the different cases, 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, and the total number of control points in the program. The apparent discrepancy between the size in LoC and the size of the SDG in vertices can be largely attributed to the modeling of global variables in the SDG. Global variables are added as parameter to every function, which adds a number of vertices, linear in the amount of global variables, to each function's PDG. For a better intuition of the size of the program, global variables have been filtered out in the "non-global" column, which shows a better correspondence with the size in LoC. Note that vertices representing function parameters are not included in the graph traversal as defined in Section 4, and therefore do not impact analysis speed.

The programs were selected 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 100 different test runs were recorded, and profile data was saved. In case of the TV control software, we extracted the dynamic information using the embedded profiling software developed by our colleagues [50], with which they collected profile information on a basic block level. However, this level was not sufficiently detailed to extract exact branching information, and therefore the TV software has not been included in IV1. This also means that the calibrated heuristics resulting from IV1 have not been used on the TV software in IV2, since it makes little sense to investigate the influence of calibration on a case that has not been included in the calibration process.

5.2 Experimental process

In both IV1 and IV2 we need measurements from static as well as dynamic analysis. In IV1, we measure the average true/false ratio (dynamic) for conditions

[6] `www.grammatech.com`

where heuristics apply (static). In IV2 we compare two rankings, one consisting of program locations ranked by their execution likelihood estimate (static), and one ranked by the measured likelihood (dynamic). The process of gathering this information consists of 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 estimating every basic block).

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. In addition, save the branch direction percentages for use in IV1.

5. For IV2, we create two sets of program locations, the first sorted by static estimates, the second by measured dynamic usage, and compare them using Wall's unweighted matching method [44]. This will give us a *matching score* for different sections of the two rankings.

To illustrate this matching 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 matching 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%.

For IV3 we record the analysis time involved in computing the profiles as in IV2. The testbed used consists of programs of different size (cf. Table 2), which helps to understand the scalability of the approach. Also, to understand the effect of the caching mechanism introduced in section 4, we perform the analysis for different subsets of the complete set of program points involved. Specifically, we take subsets of differing sizes (e.g. 10%, 20% of the original size) by sampling the original set uniformly, and measure the analysis running time for these subsets. All measurements were performed on a laptop with an Intel Pentium Mobile 1.6Ghz processor and 512Mb of memory, running MS Windows XP Pro.

Project	LEH		OH		PH		RH	
Antiword	0.29	1	0.38	15	0.29	4	0.23	3
Chktex	0.60	1	0.44	4	n.a.	0	n.a.	0
Lame	0.28	1	0.35	19	0.19	1	n.a.	0
Link	0.29	3	0.30	12	0.23	7	0.34	1
Uni2ascii	n.a.	0	0.00	5	1.00	2	n.a.	0
total	0.31	2	0.34	13	0.24	5	0.29	1

Table 3: Case study programs and branch statistics

Segment	Antiword			Chktex			Lame		
	UP	HP	CHP	UP	HP	CHP	UP	HP	CHP
1	0.44	0.44	0.44	0.67	0.67	0.67	0.49	0.49	0.51
2	0.47	0.45	0.43	0.42	0.42	0.42	0.29	0.37	0.37
5	0.57	0.58	0.53	0.34	0.28	0.31	0.29	0.31	0.39
10	0.52	0.51	0.50	0.36	0.28	0.31	0.31	0.34	0.39
20	0.43	0.45	0.46	0.52	0.54	0.54	0.32	0.34	0.38
40	0.48	0.46	0.47	0.39	0.39	0.39	0.46	0.46	0.46
80	0.77	0.78	0.78	0.78	0.78	0.78	0.84	0.84	0.84
Segment	Link			Uni2ascii			TV		
	UP	HP	CHP	UP	HP	CHP	UP	HP	CHP
1	0.11	0.00	0.11	1.00	1.00	1.00	0.13	0.13	n.a.
2	0.33	0.32	0.33	1.00	1.00	1.00	0.11	0.10	n.a.
5	0.25	0.25	0.25	0.50	0.50	0.50	0.15	0.15	n.a.
10	0.59	0.53	0.52	0.25	0.25	0.25	0.17	0.17	n.a.
20	0.60	0.59	0.58	0.65	0.65	0.71	0.35	0.35	n.a.
40	0.65	0.66	0.65	0.53	0.53	0.65	0.63	0.64	n.a.
80	0.83	0.83	0.83	0.79	0.79	0.79	0.96	0.96	n.a.

Table 4: Matching scores for different segments of the rankings obtained using uniform (UP), heuristic (HP) and calibrated heuristic prediction (CHP)

6 Results

6.1 IV1: Branch heuristic measurements

The results of the first investigation can be found in Table 3. For each project and heuristic, two values are displayed: the first is the ratio of taken branches out of all the branches where the heuristic was applicable; the second is the percentage of branches in the program where the heuristic could be applied. We can observe that only for the Return heuristic, the value used in [48] corresponds to our measured total rate. Although the other values differ significantly (up to 18 percent point), at least the rationale is confirmed. For example, in the Pointer Heuristic we assume that a comparison between pointers is likely to fail, and this is in line with the measured average taken ratio of 0.24. In any event, the measured values are significantly different from those used by uniform prediction (0.5); we will see in the next investigation (IV2) whether such differences actually impact the accuracy of the profiler.

Segment	Antiword			Chktex			Lame		
	UP	HP	CHP	UP	HP	CHP	UP	HP	CHP
1	0.72	0.72	0.76	1.00	1.00	1.00	0.66	0.66	0.83
2	0.69	0.67	0.65	1.00	1.00	1.00	0.33	0.41	0.41
5	0.63	0.67	0.68	0.78	0.78	0.78	0.29	0.31	0.39
10	0.59	0.64	0.64	0.67	0.66	0.69	0.31	0.34	0.40
20	0.44	0.46	0.47	0.53	0.54	0.55	0.31	0.32	0.35
40	0.35	0.34	0.33	0.40	0.41	0.41	0.23	0.23	0.23
80	0.25	0.24	0.24	0.44	0.44	0.45	0.22	0.22	0.22
Segment	Link			Uni2ascii			TV		
	UP	HP	CHP	UP	HP	CHP	UP	HP	CHP
1	0.79	0.86	0.93	1.00	1.00	1.00	0.30	0.33	n.a.
2	0.84	0.86	0.87	1.00	1.00	1.00	0.23	0.23	n.a.
5	0.81	0.81	0.81	1.00	1.00	1.00	0.16	0.16	n.a.
10	0.81	0.81	0.81	1.00	1.00	1.00	0.14	0.14	n.a.
20	0.63	0.63	0.62	0.76	0.76	0.82	0.08	0.08	n.a.
40	0.47	0.46	0.46	0.53	0.53	0.65	0.06	0.06	n.a.
80	0.33	0.32	0.32	0.43	0.43	0.43	0.09	0.09	n.a.

Table 5: Average measured execution likelihood for different segments of the rankings obtained using uniform (UP), heuristic (HP) and calibrated heuristic prediction (CHP)

6.2 IV2: Relating static and dynamic profiles

Table 4 displays the matching scores (as explained in Section 5) for each project. On each row, a different part of the ranking has been matched, where segment n refers to the n% topmost locations in the ranking. The three values present for each project denote the scores obtained using the different variants of branch prediction; uniform, heuristic and calibrated heuristic, respectively. Since the profiling results are used as prioritization, we are first and foremost interested in the matching scores for the top-most segments. Typically, a randomized ranking would produce scores of 0.01, 0.02 and 0.05 for these top segments, which is easily outperformed by the ELAN technique. However, there is a problem in using these scores for evaluating the accuracy of the ranking. Consider Link, for example: in our test series, 11% of the locations involved were always executed, which means that the four top-most segments 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 the estimates were perfect, locations in our static segment 1 could be anywhere in segment 2, 3 or 4 of the dynamic ranking. This explains the strikingly low matching scores in the upper segment for Link. In practice this will be less of a problem since these values are used in conjunction with other priorities, e.g., the reported severity. Still, it shows that to understand accuracy completely, we also need to look at the distribution of the measured likelihood in the static ranking.

In Table 5, the average execution likelihood for all the locations in a certain segment are shown; again, the values represent the results without heuristics, with heuristics, and with calibrated heuristics, respectively. We took the ranking based on our *static* likelihood estimate and calculated the average *measured*

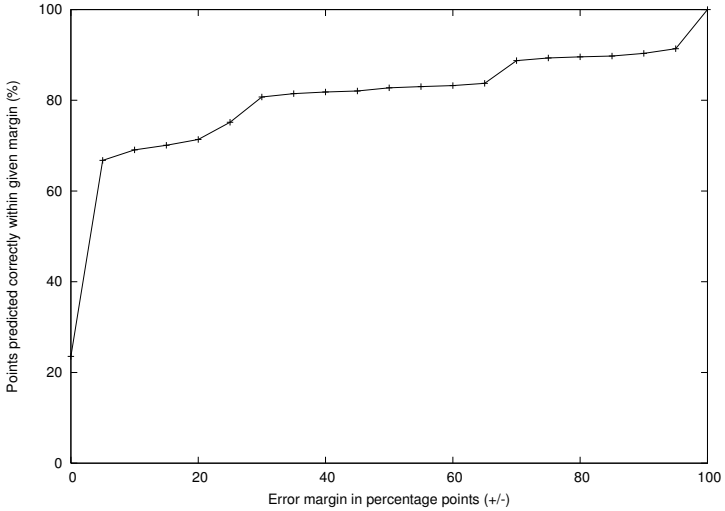

Figure 2: ELAN overall accuracy

execution likelihood for each segment. The values shown in Table 5 cannot be compared amongst programs, as they depend on the runtime behavior of the individual program. For instance, 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 relation can be observed here.

Finally, we illustrate overall accuracy by means of the relation between error margins and accurate estimates, using a diagram similar to the ones in [32], adapted to indicate profile accuracy instead of branch prediction accuracy. The figure displays the number of locations for which the execution likelihood was estimated correctly within the given error margin. For instance, point (5,65) in the graph signifies that 65% of all the locations in the experiment was estimated correctely within a 5 percent point error margin. Only one line has been plotted, because the differences between the three different variants were too small to be noticeable in this figure.

6.3 IV3: Analysis time measurements

The analysis time investigation consists of two parts. First is the relation between the size of the program and the profiler analysis time. Recall our algorithm, which computes a slice, traverses the subgraph obtained, and derives estimates for conditions. This indicates the importance of the size of the SDG, rather than the number of KLoC. This relationship is illustrated in figure 3, where the size of the SDG in vertices has been plotted against the analysis time

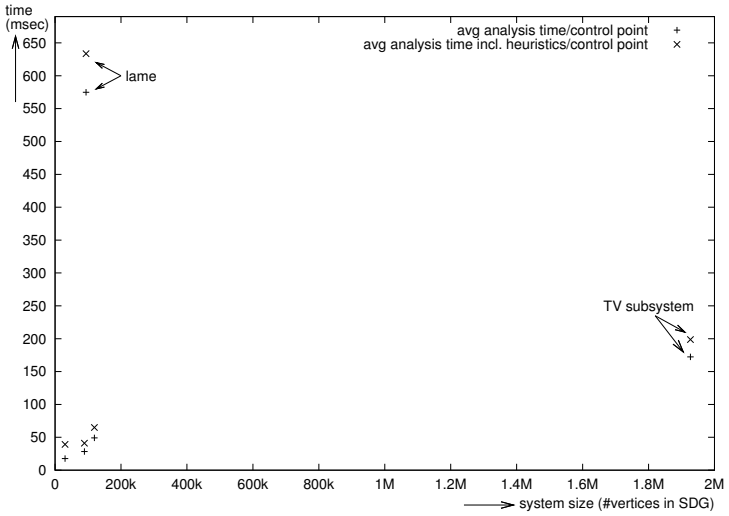

Figure 3: ELAN analysis time for various case studies

per location involved in the investigation (i.e., IV2).

The second part aims at characterizing the time behavior for a single program when a smaller or larger part of that program is analyzed. We illustrate this by means of one of the programs, antiword, where we record analysis time for different subsets of locations in that program. The resulting measurements are displayed in figure 4. Here, the point (20,120) in the graph means that the analysis of a random selection of 20% of all locations in antiword takes 120 seconds. 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.

7 Evaluation

7.1 IV1-2: Profiler accuracy

When looking at the data of the accuracy experiments, perhaps most striking are the differences between tables 4 and 5. Even though locations with a higher execution likelihood in general seem to be ranked higher, the matching 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 locations that were always executed ranged from 4% to 34%. This typically means that, even though we may propagate many of the important locations towards the top of the list, we cannot distinguish between those in the topmost segments of that list. This explains the seeming discrepancy between matching score and average execution likelihood.

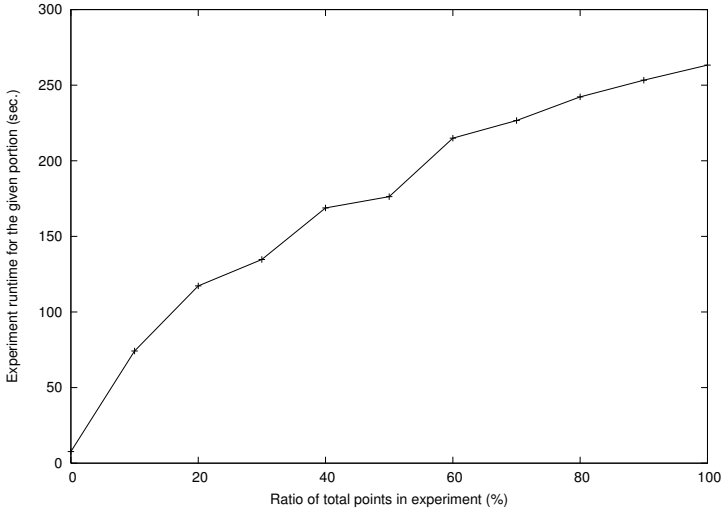

Figure 4: ELAN caching behavior illustrated

Another observation is that both accuracy tables, and especially table 5, show a drop in ranking accuracy for the NXP TV software. We believe this can be partly 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).

We tried alleviating that problem by making another pass over the source code, replacing the function pointers by direct function invocations. For this particular case this is feasible solution because all the function pointers are explicitly stored in a reference table. Although the transformation does not result in an executable system, it is perfectly analyzable using the standard techniques we employ in our approach.

However, this did not result in the increase in accuracy that we expected. There is another factor in play here: only a very small part of the software is exercised at all. This can be observed by looking at the difference between the TV software and the other programs in the 80 percent segment in Table 5. The part of the TV software under test is only the infrastructure of the complete TV software, and has very little 'real' functionality of its own. The support role of this subsystem is likely only fully exercised when run as part of a complete TV software stack, which was not the case here. With many reachable parts of the software not exercised, the measured accuracy suffers. This problem is less pronounced in the other projects we studied.

The differences in accuracy between the three variants investigated are typically small, which could be due to the small number of branches that proved to be compatible with the heuristics used. Also, it seems to indicate the rela-

tively large importance of the control structure itself, instead of the behavior of individual branches. This would explain why, even when using rather simple mechanics, the results diplayed in Table 2 and Figure 2 are promising. Notably, almost two thirds of all the locations involved in the experiment were estimated correctly with an error margin of 5 percent point. Moreover, although not displayed here, during our investigation we found that precision seems to be highest for locations with likelihood close to 0 or 1. Considering our application, this makes it relatively safe to ignore the former locations, while the latter constitute an interesting starting point for manual inspection.

7.2 IV3: Analysis time

There are a number of observations that we can make regarding analysis time behavior: 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 suggests that including more sophisticated analyses can still result in a feasible solution.

Second, there is one program that does not conform to this observation: analysis time for Lame is significantly higher than for 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. Such expressions lead to large, hightly-connected graphs, slowing down the traversal. Because this occurs early on in the program, it affects many of the locations we are testing. Specifically, it 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, in which we excluded the suspect function, we found that the analysis time for the rest of the program was according to expectations (30 ms/location). This also illustrates the fact that our analysis speed investigation is actually somewhat negatively biased, since the uniform sampling results in a relatively large number of locations from this 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.

We would like to stress that the reported timings are actually worst-case approximations. Because in our experiments the complete program is covered, adding more warning locations would not lead to more analysis time. The analysis is only run once for every basic block, so with the addition of more warnings only the source locations need to be determined. In this manner we provide the benefits of a demand-driven approach while ensuring that sorting large number of warnings remains feasible.

Finally, we remark that our approach is orthogonal to other prioritization and filtering techniques discussed in the related work. Nevertheless, 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.

7.3 Threats to validity

Internal validity As our approach is based on the SDG, the level of detail with which this graph is constructed directly affects its outcome, especially in terms of accuracy. It should be noted, therefore, that the graphs can have both 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 (aliasing), think for example of modeling control dependences when using function pointers. To improve this accuracy, a flow insensitive and context insensitive points-to analysis [38] has been employed in the implementation to derive safe information for every pointer in the program.

External validity In our test set, we use mostly 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 [19]. Recall that we observed similar behavior in our particular setup of the TV software, where large parts are simply never executed at all. 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.

8 Concluding Remarks

Prioritizing inspection results remains an important last step in the inspection process, helping developers focus on important issues. To this end, many approaches have been proposed, but none so far used the location of the reported issue. Therefore, we have presented a method for prioritization based on statically computing the likelihood that program execution reaches locations for which issues are reported. In other words, we prioritize code inspection results using static profiling. This profiler consists of a novel *demand-driven* algorithm for computing execution likelihood based on the system dependence graph.

We have investigated the feasibility of the described approach by implementing a prototype tool and applying it to several open source software systems as well as software embedded in the NXP television platform. We show the relation between our static estimates and actual execution data found by dynamic profiling and we report on the speed of our approach. This empirical validation shows that the approach is capable of correctly prioritizing 65 percent of all program locations in the test set within a 5 percent point error margin. In addition, the approach scales well to larger systems.

In concurrent work [8], we experiment with a number of ideas to further improve our approach by incorporating more advanced program analysis techniques, such as range propagation [32], 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, it is of great interest to investigate if the improved accuracy actually warrants the expenses involved.

In another track [7] we describe how to use software history information to relate inspection rules to actual bugs. In the near future, we intend to use these techniques to *select* which rules to use in the inspection process ex-ante rather than the ex-post prioritization of inspection output. Also, we want to further expand the number of industrial and open source cases where we test the feasibility of this approach.

References

[1] T. Ball and J. R. Larus. Branch prediction for free. *ACM SIGPLAN Notices*, 28(6):300–313, 1993.

[2] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The Blast Query Language for software verification. *Lecture Notes in Computer Science*, 3148:2–18, 2004.

[3] D. Binkley. Interprocedural constant propagation using dependence graphs and a data-flow model. *Lecture Notes in Computer Science*, 786:374–388, 1994.

[4] W. Blume and R. Eigenmann. Demand-driven, symbolic range propagation. *Lecture Notes in Computer Science*, 1033:141–160, 1995.

[5] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1981.

[6] C. Boogerd and L. Moonen. Prioritizing software inspection results using static profiling. In *Proceedings of the Sixth International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 149–158. IEEE Computer Society Press, 2006.

[7] C. Boogerd and L. Moonen. Assessing the value of coding standards: An empirical study. In *Proceedings of the 24th International Conference on Software Maintenance (ICSM)*, pages 277–286. IEEE Computer Society Press, 2008.

[8] C. Boogerd and L. Moonen. On the use of data flow analysis in static profiling. In *Proceedings of the Eighth International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 79–88. IEEE Computer Society Press, 2008.

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

[10] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[11] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the Ninth ACM Conference on Computer and Communications Security*, pages 235–244. ACM Press, 2002.

[12] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 57–68. ACM Press, 2002.

[13] B. L. Deitrich, B-C. Cheng, and W. W. Hwu. Improving static branch prediction in a compiler. In *Proceedings of the 18th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 214–221. IEEE Computer Society Press, 1998.

[14] 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 (OSDI)*, pages 1–16. USENIX, 2000.

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

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

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

[18] T. Fahringer and B. Scholz. A unified symbolic evaluation framework for parallelizing compilers. *IEEE Transactions on Parallel Distributed Systems*, 11(11):1105–1125, 2000.

[19] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–95. ACM Press, 1992.

[20] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM Press, 2002.

[21] Fortify. Rats: Rough auditing tool for security. Available at: `http://www.fortify.com/`, 2009.

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

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

[24] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[25] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

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

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

[28] S. Kim and M. D. Ernst. Prioritizing warning categories by analyzing software history. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR)*, pages 27–30. IEEE Computer Society Press, 2007.

[29] S. Kim and M. D. Ernst. Which warnings should i fix first? In *Proceedings of the Sixth Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 45–54. ACM Press, 2007.

[30] J. Knoop and O. Rüthing. Constant propagation on the value graph: Simple constants and beyond. In *9th International Conference on Compiler Construction (CC)*, volume 1781 of *Lecture Notes in Computer Science*, pages 94–109. Springer, 2000.

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

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

[33] A. A. Porter, H. Siy, A. Mockus, and L. G. Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering Methodology*, 7(1):41–79, 1998.

[34] T. Reps and G. Rosay. Precise interprocedural chopping. In *3rd ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 41–52. ACM Press, 1995.

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

[36] S. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *6th International Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 915 of *Lecture Notes in Ccomputer Science*, pages 651–665. Springer, 1995.

[37] G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

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

[39] C. Verbrugge, P. Co, and L. J. Hendren. Generalized constant propagation: A study in C. In *6th International Conference on Compiler Construction (CC)*, volume 1060 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1996.

[40] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *16th Ann. Computer Security Applications Conference (ACSAC)*, pages 257–267. IEEE Computer Society Press, 2000.

[41] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, 1995.

[42] 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 (NDSS)*, pages 3–17. The Internet Society, 2000.

[43] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96. ACM Press, 1994.

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

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

[46] D.A. Wheeler. Flawfinder. Available at: `http://flawfinder.sourceforge.net/`, 2009.

[47] C. Wohlin, A. Aurum, H. Petersson, F. Shull, and M. Ciolkowski. Software inspection benchmarking - a qualitative and quantitative comparative opportunity. In *8th International Software Metrics Symposium (METRICS)*, pages 118–130. IEEE Computer Society Press, 2002.

[48] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *27th Ann. International Symposium on Microarchitecture (MICRO)*, pages 1–11. ACM/IEEE, 1994.

[49] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *10th ACM Conference on Computer and Communications Security (CCS)*, pages 321–334. ACM Press, 2003.

[50] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. C. van Gemund. Diagnosis of embedded software using program spectra. In *Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, pages 213–220. IEEE Computer Society Press, 2007.

TUD-SERG-2009-022
ISSN 1872-5392