Automatic Software Fault Localization using Generic Program Invariants*

Rui Abreu† Alberto González†‡ Peter Zoeteweij† Arjan J.C. van Gemund†

†Embedded Software Lab
Delft University of Technology
The Netherlands
{r.f.abreu, p.zoeteweij, a.j.c.vangemund}@tudelft.nl

‡ETS de Ingeniería Informática
Universidad de Valladolid
Spain
a.gonzalezsanchez@tudelft.nl

ABSTRACT

Despite extensive testing in the development phase, residual defects can be a great threat to dependability in the operational phase. This paper studies the utility of low-cost, generic invariants ("screeners") in their capacity of error detectors within a spectrum-based fault localization (SFL) approach aimed to diagnose program defects in the operational phase. The screeners considered are simple bit-mask and range invariants that screen every load/store and function argument/return program point. Their generic nature allows them to be automatically instrumented without any programmer-effort, while training is straightforward given the test cases available in the development phase. Experiments based on the Siemens program set demonstrate diagnostic performance that is similar to the traditional, development-time application of SFL based on the program pass/fail information known before-hand. This diagnostic performance is currently attained at an average 14% screener execution time overhead, but this overhead can be reduced at limited performance penalty.

Categories and Subject Descriptors

D.2.5 [**Software engineering**]: testing and debugging—*debugging aids, diagnostics*

Keywords

Program spectra, fault localization, black box diagnosis, error detection, program invariants.

1. INTRODUCTION

Spectrum-based software fault localization (SFL [1]) is a low-cost fault diagnosis approach that is used in several tools/techniques for automated diagnosis and debugging [4, 15, 23]. Essentially, SFL correlates execution activity of parts of a program with the program's pass/fail outcome. The analysis yields a list of suspect program parts, ranked in likelihood of containing the fault. Recent research activity has shown that SFL can pinpoint program faults up to an accuracy such that on average less than 20 percent of the program need be inspected [1, 2, 15].

Typically, SFL is used at the *development phase*, when a number of (regression) test cases are executed. However, application of fault diagnosis at the *operational phase* is gaining importance as it is becoming clear that no economic design process will deliver sufficiently dependable (embedded) software, given current-day's systems complexity [16, 20]. Despite extensive testing, residual defects may still cause failures, which also need to be detected, either as feedback to development-time debugging, or as input to a run-time diagnosis and recovery procedure (collectively known as FDIR - fault detection, isolation, and recovery).

The accuracy of fault diagnosis is critically dependent on the accuracy of the *error detection* (pass/fail information) input. Since no testing oracle is available in the operational phase, fault diagnosis is highly dependent on the error detectors that are integrated within the program. Error detection ranges from *application-specific* (e.g., a user-programmed invariant that checks if the argument to an `sqrt()` is non-negative) to *generic* (e.g., a compiler-generated range check). The latter category of detectors usually require *training* to adapt to the application-specific program profile (e.g., the actual range a variable is permitted to have). This training is automatically performed during testing at development-time.

This paper studies the utility of low-cost, generic error detectors ("screeners") as input to SFL in the operational phase. The motivation for this study is the following. In [2] it has been established that the diagnostic accuracy of SFL is not very sensitive to error detection quality, provided that the number of (test) runs (program execution profile information) is not too small. As this insensitivity especially applies to the false negatives (a weak point of low-cost screeners, as shown in the paper), low-cost screeners may already yield acceptable diagnostic accuracy. As low-cost and generic error detectors would (1) avoid costly programmer involvement, and (2) minimize run-time time/space overhead, the combination screening-SFL seems an appealing prospect in the context of dependable (embedded) software development.

An advantage of using screeners to flag a run as passed

*This work has been carried out 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.

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

or failed is that we don't rely on a reference output. While this is typically available for regression testing at development time, this is not the case in the operational phase (see Figure 1). In this paper we "replace" the reference program by screeners and investigate its influence on the diagnostic accuracy of SFL.

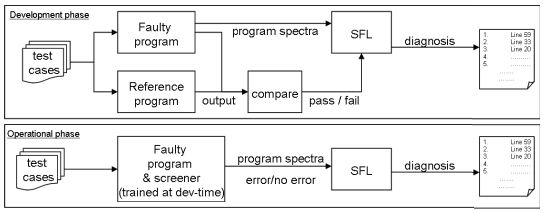

Figure 1: Screener-SFL vs. reference-based SFL

Screeners have also been used for direct fault localization instead of just error detection, albeit with limited success [21]. Also from this perspective, feeding their output to a specific (spectrum-based) fault localization algorithm increases diagnostic quality. To the best of our knowledge, we are the first to study the combination of low-cost, generic screeners with spectrum-based localization to achieve automatic fault localization in the operational phase. In particular, the paper makes the following contributions:

- We evaluate the error detection performance of two screeners, viz. the bitmask screener [11] and the range screener [22] for the Siemens benchmark set of programs [13] while varying the amount of initial training.
- We evaluate the diagnostic performance of SFL for the Siemens set benchmarks using the error information as provided by the screeners, and we compare this performance with the SFL performance based on the intrinsic pass/fail information provided by the Siemens benchmark tests itself.
- We argue that the screener-SFL combination gives more educated guesses to find the faulty locations than the stand-alone screeners, such as in [21].

Our main findings show that although the error detection quality of screeners is quite limited, the diagnostic quality of SFL using range screeners in the operational phase can match the quality of SFL based on test cases in the development phase, provide sufficient test cases are available for training (hundreds of runs).

The paper is organized as follows. In the next section we present our approach to software error detection and fault diagnosis. In Section 3 we describe our experimental setup. In Section 4 the results of our experiments are presented. A comparison to related work appears in Section 5. Section 6 concludes the paper.

2. BACKGROUND

As defined in [3], we use the following terminology. A *failure* is an event that occurs when delivered service deviates from correct service. An *error* is the part of the total state of the system that may cause a failure. A *fault* (bug) is the cause of an error in the system.

To illustrate these concepts, consider the C function in Figure 2, which is meant to return `true` if all elements in the array are greater than, or equal to zero, and `false` otherwise. There is a fault (bug) in the condition in line 6: the logic operator $\leq$ was mistakenly used instead of just $<$.

When `lst[i]` equals zero an error occurs after the code inside the conditional statement is executed. Such errors can be temporary, and need not to induce a failure: if we apply `nonNegative` to the sequence $\langle 0, -1, 2 \rangle$, an error occurs in line 7 as $lst[0]$ equals 0. However, the error occurring from `lst[0]` is "obscured" by the evaluation of the second element of the array, which is -1, and the function correctly returns `false`. Faults do not automatically lead to errors either: no error will ever occur if all elements in the array are different from zero.

```
1. bool nonNegative(int *lst, int n) {
2.   int i;
3.   int res = true;

5.   for ( i = 0; i < n; i++ ) {
6.         if ( lst[i] <= 0 ) { /* correct: < */
7.            res = false; }}
8.   return res; }
```

Figure 2: A defective C function

The purpose of *diagnosis* is to locate the *faults* (defects) that are the root cause of errors. As such, error detection is a prerequisite for diagnosis. As a rudimentary form of error detection, failure detection can be used (deciding whether a program run failed or not by inspecting the actual program output), but in software more powerful mechanisms are available, such as checking program invariants, pointers array bounds, deadlock, etc. In this paper, we will consider invariant-based error detectors.

2.1 Invariant-based Error Detection

Program invariants are conditions that have to be met by the state of the program for it to be correct. Although many kinds of program invariants have been proposed in the past [8, 9, 22], we will focus on bitmask invariants [11, 22] and dynamic range invariants [22], as they require minimal overhead (therefore, they lend themselves well for application within resource-constrained environments, such as embedded systems).

A *bitmask invariant* is composed of two fields: the first observed value (fst) and a bitmask (msk) representing the activated bits (initially all bits are set to 1). Every time the invariant is used the bits (new) are checked according to:

$$violation = (new \oplus fst) \wedge msk \quad (1)$$

where $\oplus$ and $\wedge$ are the bitwise *xor* and *and* operators respectively. If the *violation* is non-zero, an invariant violation is reported. In error detection mode (operational phase) an error is flagged. During training mode (development phase) the invariant is updated according to:

$$msk = \neg(new \oplus fst) \wedge msk \quad (2)$$

Although bitmask invariants were used with success by Hangal and Lam [11], they have limitations. First of all, their support for representing negative and floating point numbers is limited. Finally, the upper bound representation of an observed number is far from tight.

To overcome these problems, we also consider *range invariants*, e.g., used by Racunas *et al.* in their hardware perturbation screener [22]. In the context of our paper, range invariants are used to represent the (integer or real) bounds of a program point. Every time a new value is observed, it is checked against the currently valid bounds. If the value is outside the bounds, an error is flagged in error detection mode (operational phase), while in training mode (development phase) the range is extended to include the new value.

On the one hand, range invariants can handle those cases bitmask invariants cannot. On the other hand, they learn slower in the training phase, especially in the case of counters. Furthermore, range invariants are unable to represent some conditions that bitmasks can, such as even/odd properties. Despite the above shortcomings, both screeners have been proven to be useful error detectors while their simplicity allows them to be implemented in hardware, thus minimizing run-time overhead.

2.2 Spectrum-based Fault Localization

In SFL program runs are captured in terms of a spectrum. A program spectrum [12] can be seen as a projection of the execution trace that shows which parts (e.g., blocks, statements, or even paths) of the program were active during its execution. Diagnosis consists of analyzing which of the parts' activation patterns show the greatest correlation with the error pattern of different executions.

Program spectra usually allocate a boolean for each program part that signifies whether that part was executed (a so-called "hit spectrum"). Every program run produces a spectrum that constitutes a row in a binary matrix of M rows (one per run) and N columns (one for each part). Next to the spectra, a binary vector of size M is constructed using the (pass/fail) error detection information of each of the M runs (either obtained from our experimental screeners or by comparing the program output with the output of a correct reference version). For each column j of the matrix, its similarity s_j to the error vector is evaluated. The part whose column has the highest similarity is considered most likely to contain the fault. The degree of similarity between each matrix column and the error vector, is calculated using *similarity coefficients* taken from data clustering techniques [6, 14].

A similarity coefficient is a function using four counters: a_{te}, where t is either touched (1) or not (0) and e is either failed (1) or passed (0). For example, $a_{11}(j)$ counts the number of times part j was exercised in failed runs. Many similarity coefficients exist. Known to be amongst the best for SFL [1, 2], in this paper we consider the following three similarity coefficients: Ochiai, Tarantula, and Jaccard. As an example, the latter is defined as $a_{11}(j)/(a_{11}(j)+a_{01}(j)+a_{10}(j))$ (the other coefficients are not defined due to space limitations - for more information see [1]).

In this paper we consider a program part to be a statement. Consequently, the output of the fault diagnosis technique is a ranked list of statements in order of likelihood to be at fault. Given the simplicity of the algorithm the execution overhead of SFL is small (6% at the statement-level instrumentation, derived from [2]).

3. EXPERIMENTAL SETUP

In this section we describe the benchmark set used in our experiments and the workflow of the experiments.

Program	Faulty Versions	LOC	Test Cases	Description
`print_tokens`	7	539	4130	Lexical Analyzer
`print_tokens2`	10	489	4115	Lexical Analyzer
`replace`	32	507	5542	Pattern Recognition
`schedule`	9	397	2650	Priority Scheduler
`schedule2`	10	299	2710	Priority Scheduler
`tcas`	41	174	1608	Altitude Separation
`tot_info`	23	398	1052	Information Measure

Table 1: The Siemens benchmark Set

3.1 Benchmark Set

In our study, we used a set of test programs known as the *Siemens set* [13]. The Siemens set is composed of seven programs. Every single program has a correct version and a set of faulty versions of the same program. Each faulty version contains exactly one fault. Each program also has a set of inputs that ensures full code coverage. Table 1 provides more information about the programs in the package (for more detailed information refer to [13]). Although the Siemens set was not assembled with the purpose of testing fault diagnosis techniques, it is typically used by the research community as the set of programs to test their techniques.

In total the Siemens set provides 132 programs. However, as no failures are observed in two of these programs, namely version 9 of `schedule2` and version 32 of `replace`, are discarded. Besides, we also discard versions 4 and 6 of `print_tokens` because the faults in this versions are in global variables and the profiling tool used in our experiments does not log the execution of these statements. In summary, we discarded 4 versions out of 132 provided by the suite, using 128 versions in our experiments.

3.2 Workflow of Experiments

Our approach to study automatic fault diagnosis in the operational phase comprises three stages. First, the target program is instrumented to generate the statement spectra and execute the invariants (see Figure 3). To prevent faulty programs to corrupt the logged information, the program invariants and spectra themselves are located in an external component ("Screener"). The instrumentation process is implemented as an optimization pass for the LLVM tool [17] in *C++* (for details on the instrumentation process see [10]). The program points screened are all memory loads/stores, and function argument and return values.

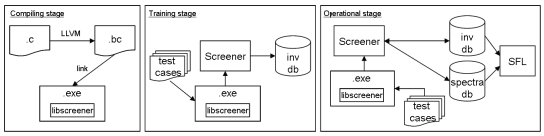

Figure 3: Workflow of experiments

Second, the program is run for those test cases for which the program passes (its output equals that of the reference version), in which the invariants are operated in training mode. The set of test cases used to train the program invariants is of great importance to the performance of the error detectors at the operational (detection) phase. In the experiments the number of these so-called correct test cases is varied between 10% and 100% of all correct cases (267 and 2666 cases on average, respectively) in order to evaluate the effect of training.

Finally, we execute the program over all test cases (total number of test cases per program are in Table 1), in which the invariants are executed in detection mode. Both errors

and spectra are collected in data bases which are input to the SFL component.

4. EXPERIMENTAL RESULTS

Before presenting our results, we first define our performance metrics for error detection and fault diagnosis, respectively.

4.1 Evaluation Metrics

Error detection Error detection techniques may either fail to recognize a genuine error (*false negative*), or may flag an error where there is none (*false positive*). Error detection quality is measured in terms of the false negative rate f_n and the false positive rate f_p, which measure the number of false negatives divided by the checking set size and the number of false positives divided by the training set size, respectively. The checking set includes all test cases in the set of test cases that failed to produce a correct output. In turn, the training set is composed of those that passed, i.e., that behave correctly (at this stage, failed/passed information is obtained by comparing the output of the faulty program with the reference program).

Fault diagnosis As spectrum-based fault localization creates a ranking of statements in order of likelihood to be at fault, we can retrieve how many statements a software developer would have to inspect until he hits the faulty one. If there are two or more statements ranking with the same coefficient, we use the average ranking position for all of them.

Let $d \in \{1, \ldots, N\}$ be the index of the statement that we know to contain the fault. For all $j \in \{1, \ldots, N\}$, let s_j denote the similarity coefficient calculated for statement j. Then the ranking position is given by

$$\tau = \frac{|\{j|s_j > s_d\}| + |\{j|s_j \geq s_d\}| - 1}{2} \quad (3)$$

We define accuracy, or quality of the diagnosis as the effectiveness to pinpoint the faulty location. It is defined as

$$q_d = (1 - \frac{\tau}{N-1}) \cdot 100\% \quad (4)$$

This metric represents the percentage of statements that need not be considered when searching for the fault by traversing the ranking.

4.2 Results

Before evaluating the performance of the screener-SFL combination, we first evaluate the error detection performance of the screeners by comparing their output to the pass/fail outcome over the entire benchmark set (all test cases, a pass/fail is determined by comparing the output of a program version to the output of the correct reference version).

Figure 4 plots the percentage of f_p and f_n for both bitmask and range invariants for different percentage of (correct) test cases used to train the invariants, when instrumenting all program points in the program under analysis. The plots represent the average over all programs, which has negligible variance (between $0 - 0.1\%$ and $3 - 4\%$, for f_p and f_n respectively). It can be seen that f_p rapidly approaches zero, which means that the screeners are unlikely to generate false alarms. On the other hand, f_n rapidly increases, meaning that even for minimal training many errors are already tolerated. This is due to:

- Limited detection capabilities: only single upper/lower bounds are screened, i.e., two simple, unary operations, in contrast to the host of invariants conceivable, based on complex relationships between multiple variables (typically found in application-specific invariants)
- Limited training accuracy: although the plots indicate that the *quantity* of training data is sufficient, the *quality* of the data is inherently limited. As explained earlier, an error need not cause a program failure. In a number of cases a (faulty) program error did not result in a failure (i.e., a different output than the correct reference program). Consequently, the screener is trained to accept the error, thus limiting its detection sensitivity.

Furthermore, the results also show that the range screener outperforms the bitmask screener.

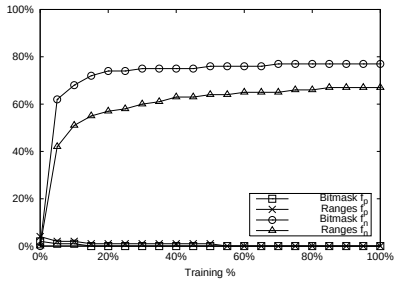

Figure 4: False positives and negatives on average

We now proceed to evaluate the diagnostic performance of the screener-SFL chain. Figure 5 plots q_d for the three similarity coefficients presented in Section 2.2 versus the training percentage as used in Figure 4 (for 10%-intervals). The first point plotted is for 10% training, since if the invariants are not trained at all, all the runs will be flagged as failed, which will render all statements equally likely to be at fault (i.e., no information). As expected, given the results in Figure 4, the range screener outperforms the bitmask screener. As for higher training effort f_p equals zero, it may be concluded that the difference in q_d is due to the lower f_n of the range screener. Figure 5 also shows that q_d is relatively insensitive to the amount of training as the f_n and f_p trends tend to cancel out.

The performance for the bitmask screener varies between 60% and 70%, for the range screener it varies 76% and 81% (meaning that 30-40% and 19-24% of the code has to be inspected on average to find the fault, respectively). Furthermore, the q_d given by the similarity coefficients are very similar, so preferring one over another yields marginal improvements. Comparing the screener-SFL performance with SFL at development-time (84% on average [1]), we conclude that the use of screeners in an operational context yields comparable diagnostic accuracy to using pass/fail information available in the development-time phase. This result is mainly due to the fact that the quantity of error information compensates the limited quality (in particular, the false negative rate). Even for 10% training the results are comparable, which corresponds to an average input to SFL of

36 true positives (correctly detected errors), 2656 true negatives, 45 false positives, and 179 false negatives. From [2] we know that in the original (development-time) scenario SFL only requires some 6 true failures to approach optimal performance, being insensitive to false negatives once 6 true failures are captured. Furthermore, as 36 exceeds this threshold, the damage of f_p (45) is limited.

While SFL execution time overhead is limited (6%), the large screening density (an average of 494 screeners for all ld/st and arg/ret points) introduce a slowdown factor ranging from 1.7 (for `tcas`) to 33.1 (for `replace`). However, these numbers are due to our choice of implementation. As mentioned in Section 3, for experimentation/isolation purposes, the target program and the screener are two components running in different processes, and the communication between them is rather time consuming. Additional measurements indicate that executing the screener code within the target program would only result in an execution time overhead of 14% on average (5% variance) for the Siemens set (calculated by inserting dummy statements representing the number of required statements to run the screener).

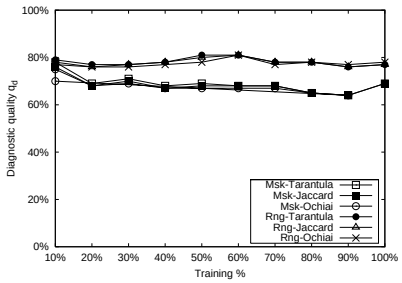

Figure 5: Diagnostic quality q_d on average

A way to reduce the overhead is to carefully select which program points to instrument (e.g., currently we also store invariants for constants, but they do not give any relevant info - hence, they could be discarded). To obtain an indication of the potential improvement we have also varied the number of range screeners by considering ld/st and arg/ret points separately (as they outperform bitmasks we only consider ranges).

Figure 6 shows q_d based on screening either ld/st points or arg/ret points. For many training situations, the (average 393) ld/st screeners approach achieves similar performance to total screening, whereas the (average 101) arg/ret screeners entail a drop in diagnostic performance. Due to space limitations we do not include the f_n/f_p plots.

Table 2 summarizes the trade-off between diagnostic performance and overhead, choosing the training percentage that delivers the best results.

	arg/ret	ld/st	arg/ret and ld/st
# Program Points	101 ± 72.3	393 ± 172	494 ± 203.6
Overhead [%]	4.8 ± 1.9	11.6 ± 4.6	14.3 ± 5.5
q_d (training)	75% (10%)	81% (50%)	81% (50%)

Table 2: Range screener density vs. Performance

In previous work [11, 21], screeners were used to directly pinpoint the faulty location. However, we observed that program invariants violations can occur in other locations than the faulty one, leading the developer to inspect code that is neither the faulty line itself nor related to it (this situation led to the conclusion that program invariants may not be useful for debugging in [21]). The disadvantages of stand-alone screeners over our screener-SFL approach are therefore twofold: (1) the set of unrelated candidate statements is much larger than for SFL, (2) the set is not ranked, which further increases the probability of inspecting unnecessary code.

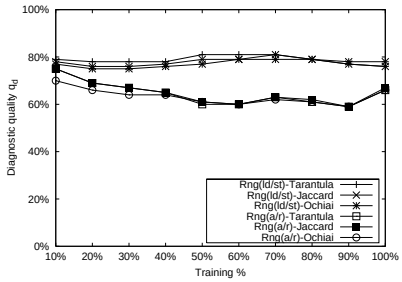

Figure 6: Diagnostic quality q_d for only either function arguments/returns (a/r) or loads/stores (ld/st)

5. RELATED WORK

Dynamic program invariants have been subject of study by many researchers [8, 9, 22]. However, conversely to the work presented in this paper, they have never been used as the error detection input for fault localization techniques but focused in finding the root cause of the fault themselves.

Daikon [9] is a dynamic invariant detector for C, C++, Java, Perl, and IOA. It stores program invariants for several program points, such as call parameters, return values, and relationship between variables. Carrot [21] is a lightweight version of Daikon, as it uses a smaller set of invariants. Carrot tries to use program invariants to pinpoint the faulty locations directly. Similarly to our experiments, the Siemens set is also used to test Carrot. Due to the negative results reported, it has been hypothesized that potential program invariants may not be suitable for debugging. DIDUCE [11], on which the bitmask study in our paper is inspired, is an dynamic invariant detector for Java programs. Essentially, it stores program invariants for the same program points as in this paper. It was tested on four real world applications with successful results. However, the error used in the experiments was caused by a variable whose value was constant throughout the training mode and that changed in the operational phase (hence, easy to detect using the bitmask screener). In [22] several screeners are evaluated to detect hardware faults. Evaluated screeners include dynamic ranges, bitmasks, TLB misses, Bloom filter based screener. The authors concluded that bitmask invariants perform slightly better than range invariants. However, the (hardware) errors used to test the screeners constitute random bit errors which, although ideal for bitmask screeners, hardly occur in program variables.

Our paper differs from the above work in that the screeners are not directly used to (help) pinpoint the root cause of a failure, but just to label an execution sequence as passed or failed as input to subsequent, more accurate fault localization.

Many fault localization tools exist, for example [5, 7, 15, 18, 23, 24]. In this paper we have used spectrum-based fault localization because it is known to be among the best techniques [15, 18]. However, none of the above work has been based on automatic error detection.

6. CONCLUSIONS & FUTURE WORK

In this paper we have studied the utility of low-cost, generic invariants ("screeners") in their capacity of error detectors within an SFL approach aimed at the operational phase, rather than just the development phase. Experiments carried out using the Siemens set show that, despite the simplicity (and the resulting high error rates) of the screeners, the diagnostic performance (measure in terms of q_d) of SFL is similar to the development-time situation. This implies that fault diagnosis with an accuracy comparable to SFL is indeed feasible at the operational phase with (1) no (human) overhead during the development phase, and (2) reasonably limited overhead during the operational phase (14% on average for the Siemens set). Furthermore, our results show that the overhead can be further reduced at a reasonable diagnostic performance penalty.

Future work includes the following. Although other screeners are more time consuming, they might lead to better diagnostic performance, and therefore worth investigating. We plan to study alternative screeners, such as Bloom filters [22], also allowing more analysis on the impact of false positives f_p and negatives f_n on q_d. Inspired by the fact that only a limited number of (so-called collar) variables are primarily responsible for program behavior [19], we also plan to study the impact of (smartly) reducing the amount of screened program points (overhead) on f_p and f_n, and consequently on q_d. Although earlier work has shown that the current q_d metric is comparable to the more common T-score [15, 23], we also plan to redo our study in terms of the T-score, also allowing a direct comparison between the use of stand-alone screeners and our screener-SFL approach.

7. ACKNOWLEDGMENTS

We gratefully acknowledge the fruitful discussions with our TRADER project partners.

8. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proc. PRDC '06*, pages 39–46, 2006.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proc. TAIC PART'07*, Sep 2007.

[3] A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. of DSN'02*, pages 595–604. IEEE CS, 2002.

[5] H. Cleve and A. Zeller. Locating causes of program failures. In *Proc. ICSE'05*, Missouri, USA, 2005.

[6] A. da Silva Meyer, A. A. Franco Farcia, and A. Pereira de Souza. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (*Zea mays* L). *Genetics and Molecular Biology*, 27(1):83–91, 2004.

[7] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In A. P. Black, editor, *Proc. ECOOP 2005*, volume 3586 of *LNCS*, pages 528–550. Springer-Verlag, 2005.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. ICSE'99*, pages 213–224, 1999.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, , and C. Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, 2007.

[10] A. González. Automatic error detection techniques based on dynamic invariants, Aug. 2007. Master's thesis.

[11] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. ICSE'02*, May 2002.

[12] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. *ACM SIGPLAN Notices*, 33(7):83–90, 1998.

[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. ICSE'94*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE CS Press.

[14] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[15] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. ASE'05*, pages 273–282, NY, USA, 2005.

[16] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[17] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. CGO'04*, Palo Alto, California, Mar 2004.

[18] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10):831–848, 2006.

[19] T. Menzies, D. Owen, and J. Richardson. The strangest thing about software. *Computer*, 40(1):54–60, January 2007.

[20] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, U.C. Berkeley, March 2002.

[21] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. Reiss. Automated fault localization using potential invariants. In *Proc. AADEBUG'03*, 2003.

[22] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee. Perturbation-based fault screening. In *Proc. HPCA'2007*, pages 169–180, Feb. 2007.

[23] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proc. ASE'03*, Montreal, Canada, October 2003. IEEE CS.

[24] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *Proc. AADEBUG'05*, pages 33–42, Monterey, California, USA, 2005. ACM Press.