PredSym: Estimating Software Testing Budget for a Bug-Free Release

Arnamoy Bhattacharyya
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
arnamoyb@ece.utoronto.ca

Timur Malgazhdarov
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
timur.malgazhdarov@mail.utoronto.ca

ABSTRACT

Symbolic execution tools are widely used during a software testing phase for finding hidden bugs and software vulnerabilities. Accurately predicting the time required by a symbolic execution tool to explore a chosen code coverage helps in planning the budget required in the testing phase. In this work, we present an automatic tool, PredSym, that uses static program features to predict the coverage explored by a symbolic execution tool – KLEE, for a given time budget and to predict the time required to explore a given coverage. PredSym uses LASSO regression to build a model that does not suffer from overfitting and can predict both the coverage and the time with a worst error of 10% on unseen datapoints. PredSym also gives code improvement suggestions based on a heuristic for improving the coverage generated by KLEE.

CCS Concepts

•**Computer systems organization → Reliability;**

Keywords

Software Testing; Bugs; Symbolic Execution

1. INTRODUCTION

Software testing is resource-hungry, time-consuming, labor intensive, and prone to human omission and error. Despite massive investments in quality assurance, serious code defects are routinely discovered after software has been released [16], and fixing them at so late a stage carries substantial cost [17]. It is therefore imperative to overcome the human-related limitations of software testing by developing automated software testing techniques.

Automated techniques like model checking and symbolic execution, are highly effective [1, 18], but their adoption in industrial generalpurpose software testing has been limited. We blame this gap between research and practice on three challenges faced by automated testing: scalability, applicability, and usability. Path explosion – the fact that the number of paths through a program is roughly exponential in program size – severely limits the extent to which large software can be thoroughly tested. One must be content either with low coverage for large programs, or apply automated tools only to small programs.

During the software testing phase, it is often useful to have a good estimate on the time required for testing. This helps in planning the time and resources to be invested, to test the software before a solid release. Different static program features can be used to determine the coverage provided by a symbolic execution engine. If the testing team has a given coverage to reach, they can get an estimate of the time necessary to reach that estimate. On the other hand, if there is a fixed time allotted for the testing purpose, based on the estimated coverage, the team can decide whether the software can be released with a given tolerance of security vulnerabilities.

In this research we use machine learning techniques to predict the coverage explored by a symbolic testing engine for a fixed time as well as the time required to explore a given coverage. We use a publicly available symbolic execution tool KLEE to test our approach on a variety of applications. We try to answer the following research questions:

1.1 Research Questions

- What are a set of static program features that determine the coverage explored by a symbolic engine?
- Can the coverage and time be predicted reasonably using the static program features?
- Can suggestions be given on slight modifications of the code that may result in a higher coverage?

2. RELATED WORK

Dynamic symbolic execution has been implemented by several tools from both academia and research labs (e.g., [4, 5, 1, 6, 7, 8, 9, 10, 11]). These tools support a variety of languages, including C/C++, Java and the x86 instruction set, implement several different memory models, target different types of applications, and make use of several different constraint solvers and theories. In this section we discuss briefly a few existing popular symbolic execution tools.

EXE [7] is a symbolic execution tool for C designed for comprehensively testing complex software, with an emphasis on systems code. To deal with the complexities of systems

A-TEST'16, November 18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4401-2/16/11...$15.00
http://dx.doi.org/10.1145/2994291.2994294

code, EXE models memory with bit-level accuracy. This is needed because systems code often treats memory as untyped bytes, and observes a single memory location in multiple ways: e.g., by casting signed variables to unsigned, or treating an array of bytes as a network packet, inode, or packet filter through pointer casting. As importantly, EXE provides the speed necessary to quickly solve the constraints generated by real code, through a combination of low-level optimizations implemented in its purposely designed constraint solver STP [7, 13], and a series of higher-level ones such as caching and irrelevant constraint elimination.

KLEE [1] is a redesign of EXE, built on top of the LLVM compiler infrastructure [15]. Like EXE, it performs mixed concrete/symbolic execution, models memory with bit-level accuracy, employs a variety of constraint solving optimizations, and uses search heuristics to get high code coverage. One of the key improvements of KLEE over EXE is its ability to store a much larger number of concurrent states, by exploiting sharing among states at the object-level, rather than at the page-level as in EXE. Another important improvement is its enhanced ability to handle interactions with the outside environment – e.g., with data read from the file system or over the network – by providing models designed to explore all possible legal interactions with the outside world. As a result of these features, EXE and KLEE have been successfully used to check a large number of different software systems, including network servers, file systems, device drivers and library code.

3. BACKGROUND

The term symbolic execution has different meanings in different settings. Informally, we understand symbolic execution as a way of interpreting programs that contain symbolic values. A symbolic value is defined by the symbol and the set of concrete values it can range over. For instance, we can define α to be a symbol that can range over any value from the set of all 32-bit integers (such a set can be viewed as the type of the symbol). To perform symbolic execution on C programs, we let variables store symbolic values (e.g., variable x stores symbol α rather than a concrete integer like 3).

To interpret a program with symbolic values, we have to extend the usual semantics of the program. For example, executing the statement $y = x + 3$ where $x = \alpha$ should yield $y = (\alpha + 3)$, a symbolic *expression*. The symbolic executor maintains the program state throughout the execution.

The state comprises two parts: *Var*, a mapping from variables to values which include symbolic expressions (e.g., after executing $y = x + 3$, *Var* becomes $x = \alpha, y = (\alpha + 3)$) and a set of constraints on symbolic values. For example, we can constrain symbols by ranges (e.g., $\alpha > 0, 1 \leqslant \beta < 10$), or constrain the relationship between symbols (e.g., $\alpha < \beta$). Constraints on symbols can be provided as part of the program specification, or can be induced from the execution.

The symbolic executor runs a program in very much the same way as how an ordinary interpreter does. However, things start becoming different when it comes to conditionals, where the execution has to branch according to the state. In C, conditionals correspond to if-statements. An if-statement consists of a condition, which is an expression, a true branch, which is executed if the condition is evaluated to true, and a false branch, which is executed otherwise. If the condition is a symbolic expression, it could be that the condition may evaluate to either true or false, hence both branches could be feasible. To completely explore all possibilities, the symbolic executor must conceptually fork the execution to examine both branches.

Constraint solvers are used to reason about symbolic expressions automatically. A constraint solver is a procedure that, given a set of constraints over variables, finds an assignment of the variables that satisfy the constraints. Today, there are many types of constraint solvers available, and they vary in the problem domains that they are designed for. The choice of constraint solvers depends on the language and the nature of the program being executed.

To summarize, symbolic execution, in its simplest form described above, explores all possible paths in a program that a normal run can execute. No abstraction on values is made, and therefore symbolic execution retains complete information of how values flow through the program.

The code coverage (fractions of paths explored as compared to the number of possible paths in a program) depends on a number of program features. To predict the coverage given a time or to predict the time required to explore a given code coverage, therefore, we have to extract the features first and then build the prediction model. In the next section, we describe the program features we use to train our machine learning model and provide description on how to actually utilize them to build the prediction model without over-fitting.

4. APPROACH

The performance of a symbolic execution tool is measured by the code coverage provided by the tool for a given time budget [1]. For detecting bugs, this is crucial because given an arbitrary code, more coverage indicates a higher probability of finding a random bug in the code. On the other hand, if for a release of a bug-free software, a certain coverage testing is sufficient, the testing time estimation helps plan the budget that will be used for the testing phase. Our approach for predicting the performance of a symbolic execution tool based on program behaviour is done in the following three steps:

4.1 Step 1: Extract Static Program Features

The code coverage provided by a symbolic execution tool depends a number of non-trivial program features. For example, a lower lines of code (LOC) does not necessarily mean that given the same time budget, the symbolic execution tool will explore more coverage as compared to a program with higher LOC. If the program with a lower LOC has more branches for example, the symbolic execution tool will take more time to generate a reasonably high coverage.

We identify the following program features that determine the coverage generated by a symbolic execution tool:

4.1.1 Total Number of Instructions

Intuitively, a code with more number of instructions takes more time than a code with fewer instructions. Therefore it is an important metric that indirectly determines the time needed by a symbolic execution tool.

4.1.2 Total Number of Branch Instructions

This is an important code feature that determines the coverage of a symbolic execution. Each branch instruction in the program adds a pair of new paths in a program. When the branches are nested, there is an exponential increase in the number of paths the program has. Therefore, we have used the total number of branch instructions in a program as a feature.

4.1.3 Average Depth of Branch Instructions

The total number of branch instructions alone does not give a total picture of the number of distinct paths taken during the program run. For the same number of branches, with a higher branching depth, the number of dynamic paths increases combinatorially. We therefore consider the average branch depth for estimating the total number of paths explored by the symbolic execution tool. This metric is calculated by the following formula:

$$\frac{\sum_{i=1}^{n} depth_i}{n} \tag{1}$$

Where $depth_i$ is the depth of a starting branch (if statement) in a block of nested branches and n is the number of starting branches in the program.

4.1.4 Average number of exit points from programs

The number of paths in a program will have a direct relation with the number of exit points from the program, where each exit point adds to at least one path in the program. To get the program termination points, we look at the calls to the *exit* from all the functions and return statements from *main*. To have a normalized value that can be comparable across benchmarks, we divide the total number of exit points by the total number of functions in the program.

4.1.5 Number of program inputs

Symbolic execution considers symbolic values to the inputs for a program instead of concrete values. In this way it can generate a lot more input testcases by constraint solving. Therefore, the total number of inputs for a program will have an effect on the coverage generated by the symbolic execution tool. The higher the number of inputs, more conditions needs to be checked with a possible increase in input dependent path exploration.

4.1.6 Average Loop Depth

This feature is necessary due to its importance in determining which paths the symbolic execution tool will explore to maximize coverage. As loops with high depth takes a long time to execute and the symbolic execution tool may wait infinitely waiting to finish exploring the path containing the loop, most tools try to avoid exploring paths containing deep loop nests. But if there is no other path to explore, a program with a large number of deeply nested loop will have low coverage given a specific time budget. This metric is calculated by the following formula:

$$\frac{\sum_{i=1}^{m} depth\_loop_i}{m} \tag{2}$$

Where $depth\_loop_i$ is the depth of the outermost loop in a nested loop block and m is the number of outermost loops in the program.

4.1.7 Total number of recursive calls

Similar to the case of deep loop nests, recursions also cause more time to execute programs symbolically. Therefore we add the total number of recursive function calls to the feature set as well.

We use static analysis to extract the above mentioned program behaviour metrics except the number of program inputs. We can not determine automatically the number of program inputs because it is not easy to determine the number of inputs of an arbitrary program.

4.2 Step 2: Profiling

After extracting the program behaviour metrics for a number of programs, we run them using the *same* time and symbolic inputs (the number of inputs that are symbolic) to get the *coverage* profile. The time and coverage information along with the static program feature information create a *training* file that is fed to a machine learning algorithm described in the next section.

4.3 Step 3: Building Performance Model

After collecting all the coverage and time information for a number of programs along with their statically identified features, we use machine learning to generate a performance model that can predict the time budget for a preferred coverage of any arbitrary program. This is not an easy task given the arbitrary interaction between the non-trivial program features. Also we have to be careful so that the model does not overfit on the training dataset and it gives reasonable prediction on unseen test data.

4.3.1 Enriched Feature Space Using LASSO

For building a model that can reasonably predict the coverage or the time, a simple linear model is not enough. Without prior knowledge about the interaction of the parameters (our static program features) and how they together determine the coverage and time is a non-trivial difficult problem. We can make guesses about the interaction of different parameters but in that case, finding out the correct subset from all the guesses is difficult.

Least Angle Shrinkage and Selection Operator (LASSO) is a variation of linear regression that can select the best parameters that describe the model from a pool of parameters. In this way, LASSO makes it possible to select the most significant parameters for predicting the success rate of commits of the transactions from an initial pool of guessed parameters. In this section, we describe briefly how LASSO works and also describe how we build our initial set of parameters with an educated guess.

4.3.2 Background on LASSO

Shrinkage is a method in statistics where a penalty is applied to the coefficients of the regression predictor model and thus the coefficients values are shrunk to bring them close to zero. Shrinkage techniques are useful in cases where there is high correlation between the model parameters or in cases where some parameters cause the model to overfit. By selecting the parameter that best describe the model, these shrinkage methods often result in better prediction accuracy. Ridge regression technique [19] adds a L_2 penalty to the coefficients and thus shrink the coefficients of models *close* to '0' but does not shrink any coefficient to '0' and therefore cannot eliminate any parameter. Therefore, although the ridge regression technique builds model with better prediction accuracy, the generated models are not different from the simple linear regression in terms of user interpretability.

By adding a L_1-penalty to the regression coefficients, the *Least absolute shrinkage and selection operator* (LASSO)

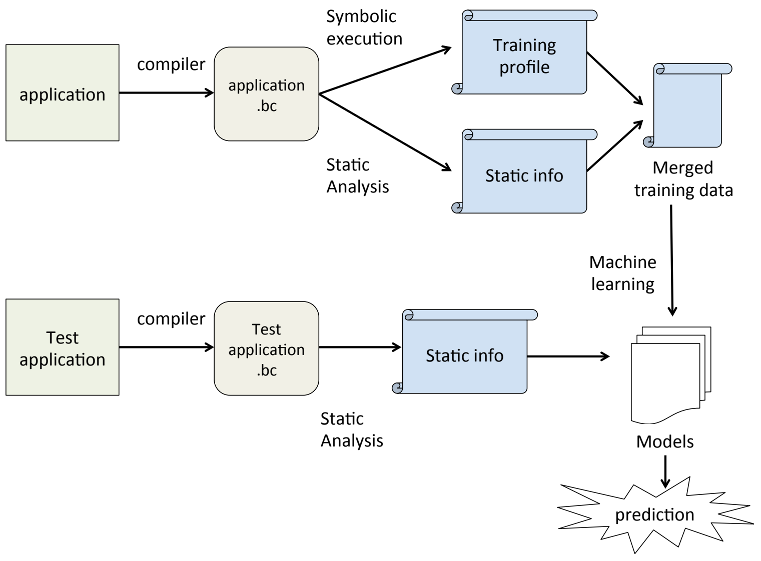

Figure 1: Workflow of PredSym.

technique [20] bridges the gap between user interpretability and prediction accuracy of the generated models.

Here we describe the regular (offline) LASSO briefly: LASSO assigns an L_1 penalty to the coefficients of input parameters (predictors) which leads to sparse solutions and thus achieves results that are easier to interpret. LASSO accepts n training examples or observations $(y_i, x_{ij}) \in \mathbb{R} \times \mathbb{R}^m, i = (1, 2, \ldots, n)$ and $j = (1, 2, \ldots, m)$. We assume we have m input parameters for the model.

Here y is the performance metric we want to model and x is the set of input parameters. We wish to fit a linear model to predict the success ratio of commits y_i as a function of x_{ij} and a feature vector $\theta \in \mathbb{R}^m, y_i = x_i^T\theta + v_i$, where v_i represents the noise in the observation. The LASSO optimization problem is given by

$$\min_{\theta} \frac{1}{2} \sum_{i=1}^{n} (x_i^T\theta - y_i)^2 + \mu_n \|\theta\|_1 \tag{3}$$

where μ_n is a regularization parameter. The solution of Equation 3 is typically sparse, i.e. the solution θ has few entries that are non-zero, and therefore identifies which dimensions in x_i are useful to predict the response y_i. The original LASSO proposal [20] did not have the additional $\frac{1}{2}$ multiplied with the first term of Equation (3). It was later shown that the LASSO problem is equivalent to the Basis Pursuit Denoising (BPDN) optimization problem which needs the multiplier $\frac{1}{2}$. Also BPDN representation of LASSO makes it algorithmically easier to solve. LASSO is particularly useful in our case because LASSO is used in cases where the number of observations is less than the number of predictor variables and it generates a model that is more interpretable than Ordinary Least Square Regression or Ridge Regression [19].

4.3.3 Forming an Initial Search Space

Though LASSO can select the most relevant parameters from a pool of initial parameters, forming the initial parameter space is non-trivial, as there can be infinite number of transformations (e.g. squares, cubes, logarithms, exponentials etc.) that can be applied to each hyperparameter and then each of these transformed hyperparameters can react in infinite number of ways to build a model, giving a combinatorial explosion in the number of hyperparameters to build a model from.

Bhattacharyya et al. [14] provides a solution that uses an educated guess about how most parameters behave in regression models and define a normal form called Extended Performance Model Normal Form (EPMNF) that can be used to define an initial search space for parameters. The EPMNF has the following form:

$$p_{hybrid} = \{\iota_i^k \log^l \iota_i^k \cup C_w(\iota_i^k \log^l \iota_i^k)\},\ \{\iota_i, \iota_2, \ldots, \iota_r\} \in I \tag{4}$$

Here C_w represents the interaction terms constructed from hyperparameters generated after applying transformations using the EPMNF definition, taken w at a time. If from EPMNF we have κ transformed *hyperparameters* initially, we construct new *hyperparameters* by taking all different combinations of the transformed hyperparameters in groups of size w where $2 \leq w \leq \kappa$. For example, if EPMNF gives us three *hyperparameters* $\iota_1^2, \log \iota_2, \frac{1}{\iota_3}$, we will construct the following new *hyperparameters* in groups of size 2 and 3: $\iota_1^2 \cdot \log \iota_2, \frac{\log \iota_2}{\iota_3}, \frac{\iota_1^2}{\iota_3}, \frac{\iota_1^2 \cdot \log \iota_2}{\iota_3}$. Bhattacharyya et. al mentions that the predictive quality of the models using the values $w = 2, 3, 4$ greatly improves precision of models and a value of $w > 4$ marginally improves the model quality. We have experienced a good trade-off between quality and overhead with a value of $w = 2$.

After the *initial* search space of *hyperparameters* is formed, LASSO regression models are generated. LASSO removes insignificant *hyperparameters* and generates easily interpretable model for predicting the success ratio of commits.

Figure 1 shows the workflow of our tool.

5. GOING BEYOND: PROVIDING SUGGESTIONS FOR IMPROVEMENT

Once we have built a model on which we can test applications for predicting coverage for a fixed time or for predicting time for a given coverage, we also provide suggestions about improving the code that may result in an improved coverage for a less time. We apply a brute-force search for optimization by slightly modifying (both increasing and decreasing) a single feature (e.g. number of branches) by a small fraction of the original value, keeping all other feature values the same. This gives us a new predicted value for the metric of intention (either coverage or time). We repeat this procedure multiple times for each feature and observe if there is a gradual decrease in case of time or a gradual increase in case of coverage. If there is a gradual improvement (either decrease or increase depending on the metric of interest) for modifications of a parameter, we suggest changing *that* parameter value to the user. If more than one parameter have gradual decreases, we suggest the one with the highest decrease from the original predicted value. One example improvement is the removal of redundant checks (branches) in the code.

Figure 2 shows our algorithm for this search.

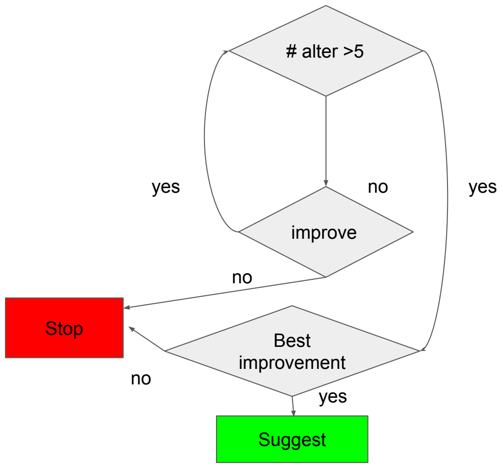

Figure 2: Suggestion Algorithm by PredSym for code improvement.

6. IMPLEMENTATION

We have implemented our tool using the latest stable release of KLEE [1] to gather profile about symbolic execution. We have written several LLVM compiler passes to extract the static features from the program. The LASSO machine learning tool is written in R and we have a web based interface written in python and HTML5 and CSS that enables users to upload their files in the webserver. After performing the static analysis on the uploaded code and prediction based on the existing model, PredSym presents the results in text format that the user can download. Deploying our tool in a web-based interface allows us to run heavy machine learning algorithms in a powerful cloud environment.

7. EVALUATION

In this section, we first try to argue whether our choice of the program static feature makes sense. Then we provide prediction results on both seen and unseen data from a wide range of applications from GNU Coreutils and Bugbench [3]. In the end, we rank the program features based on their effect on improving the code coverage provided by the symbolic execution engine KLEE [1].

7.1 Importance of Chosen Features

In the first set of experiments, we show how the chosen static program features affect the coverage with increased time budget. We ran KLEE on 93 applications the Coreutils version 6.11, that had bugs reported [1], with the following symbolic input budget for all applications.

klee –sym-args 0 1 10 –sym-args 0 2 2 –sym-files 1 8 –sym-stdout

The option combination tells KLEE to use *zero* to *three* symbolic arguments, the first being of size 10 and the last 2 being of size two for the program. It also tells KLEE to use *one* symbolic file of maximum size 8 bytes as input and treat *stdout* as symbolic. The options given to KLEE has to come from the deep understanding of the program behaviour and we chose the options suggested by the authors of KLEE for the Coreutils package. As our goal is to use KLEE in a testing phase of the development life cycle, we believe that the tester will have good understanding of the code and its inputs so that he can supply the appropriate symbolic input budgets to KLEE.

We performed experiments with two different time budgets – 30 and 120 seconds for the 93 Coreutils apps and 5 applications from the Bugbench.

We observe that the coverage increases with increasing time budget (from 60% to 70% maximum). For each of the metrics, the coverage follows the *same* statistical distribution regardless of the time budget.

7.2 Prediction Results

For testing the quality of the prediction tool, we first gather coverage and time information and also the static program features for 85 coreutils applications and 3 Bugbench applications using the arguments mentioned in the previous section. We use five different randomly selected times (between 1 minute to 30 minutes) for gathering the coverage information. This gives us the training data for the machine learning model. Then we perform predictions on both seen and unseen (not used while training) benchmarks – 8 from Coreutils and 2 from Bugbench. Performing predictions on both seen and unseen data reveals whether there is overfitting on the training data by our machine learning framework.

Figures 3 and 4 show our prediction results on the both seen and unseen benchmarks from both Coreutils and Bugbench, in the ascending order of the predicted and original values. Figure 3 shows the predicted and actual times the code will take to reach 60% coverage. We have an error of 10% for the prediction. Figure 4 shows the actual and predicted coverages given a time budget of 5 minutes for the 6 applications. The predictions are within 3% of the actual values. The error in the prediction is reasonable for us, confirming no chance of overfitting. As seen in the figures, trend is nicely captured by our prediction.

7.3 Important Program Features for Determining Coverage and Time

We also conduct experiments for ranking the program features that affect the performance of symbolic execution in the tested benchmarks. As seen in Section 5, we develop an algorithm for suggesting changing code behaviour that will produce better coverage given a time budget and a less time given a coverage budget. Figures 5 and 6 show the percentage of the applications from the Coreutils and Bugbench suite in each category. It is interesting to note that both for improving time and coverage, a major percentage of applications need to change the number of branch instructions and the branch depth in their code, giving an indication that there may be presence of unnecessary corner case checks that can be removed. The next major percentage of applications can benefit from reducing the average loop depth. Aggressive compiler optimizations on loops may be applied on these codes to provide a better coverage or a less time. Our tool provides these suggestions and relies on the application developer to find opportunities for improvement.

8. CONCLUSION

In this work, we demonstrate our tool PredSym, that can automatically predict the coverage explored by a symbolic execution tool for a given time budget and the time necessary to test a given portion of the code. We use program features that indirectly determine the total number of program paths. We make the use of LASSO regression to predict the performance and also provide an algorithm to suggest code modifications for improvement. Results agree with our choice of program features and also the prediction results are reasonably good on unseen data. We believe that using this tool, the software testers will be greatly benefited for estimating a budget for the testing phase of the software, and thus release the highest quality products that meet the budget.

9. REFERENCES

[1] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08). USENIX Association, Berkeley, CA, USA, 209-224.

[2] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In Proceedings of the sixth conference on Computer systems (EuroSys '11). ACM, New York, NY, USA, 183-198.

[3] Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. 2009. BegBunch: benchmarking for C bug detection tools. In Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009) (DEFECTS '09), Ben Liblit, Nachiappan Nagappan, and Thomas Zimmermann (Eds.). ACM, New York, NY, USA, 16-20

[4] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In TACAS'07, 2007.

[5] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In ASE'08, Sept. 2008.

[6] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself (invited paper). In SPIN'05, Aug 2005.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In CCS'06, OctâĂŞNov 2006. An extended version appeared in ACM TISSEC 12:2, 2008.

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In PLDIâĂŹ05, June 2005.

[9] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In NDSS'08, Feb. 2008.

[10] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In ESEC/FSEâĂŹ05, Sep 2005.

[11] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In TAP'08, Apr 2008.

[12] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In CAV'06, 2006.

[13] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In CAV'07, July 2007.

[14] Arnamoy Bhattacharyya and Torsten Hoefler. 2014. PEMOGEN: automatic adaptive performance modeling during program runtime. In Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT '14). ACM, New York, NY, USA, 393-404.

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '04). IEEE Computer Society, Washington, DC, USA, 75-.

[16] RedHat. RedHat security. http://www.redhat.com/security/updates/classification, 2005.

[17] Steve McConnell. Code Complete. Microsoft Press, 2004.

[18] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the Swarm tool. In Intl. SPIN Workshop, 2008.

[19] Hoerl, A. E. and Kennard, R. W. [1970a]. Ridge regression: biased estimation for non-orthogonal problems. Technometrics 12, 55-67.

[20] Tibshirani, R. (1996): Regression Shrinkage and Selection Via the Lasso, J. Royal Stat. Soc. (B), 58, 267–288.

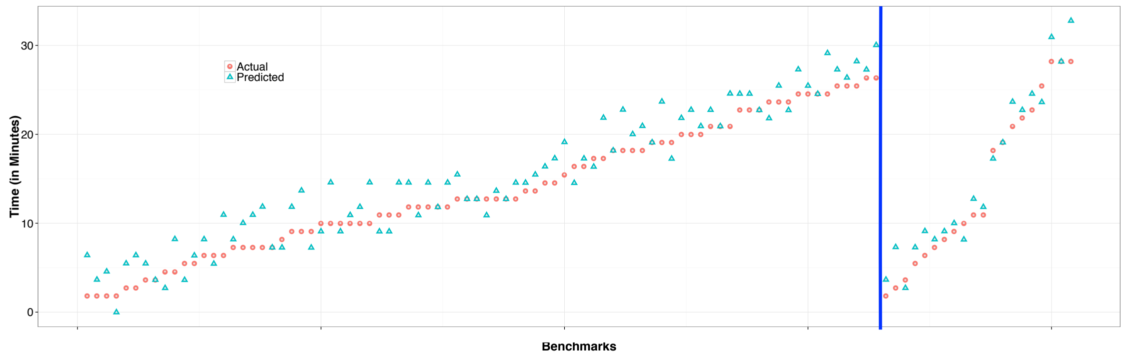

Figure 3: Predicted and actual values of time (in minutes) required to generate 60% coverage on seen (left of blue line) and unseen data (right of blue line).

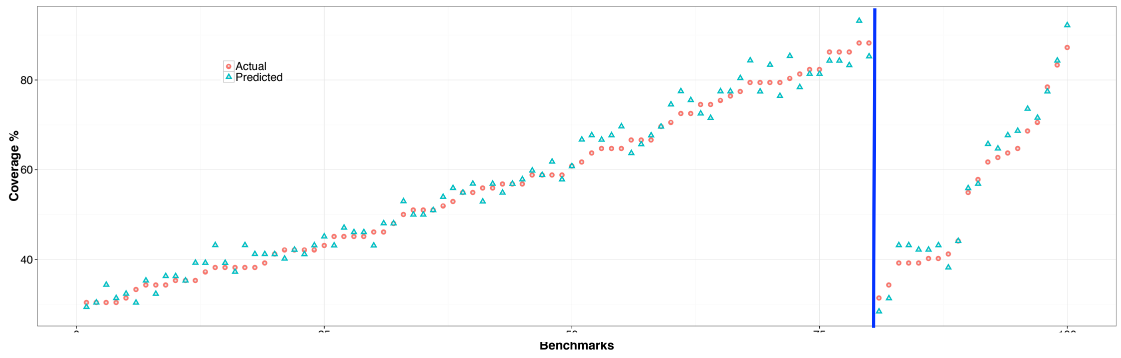

Figure 4: Predicted and actual values of coverage generated in 5 minutes on seen (left of blue line) and unseen data (right of blue line).

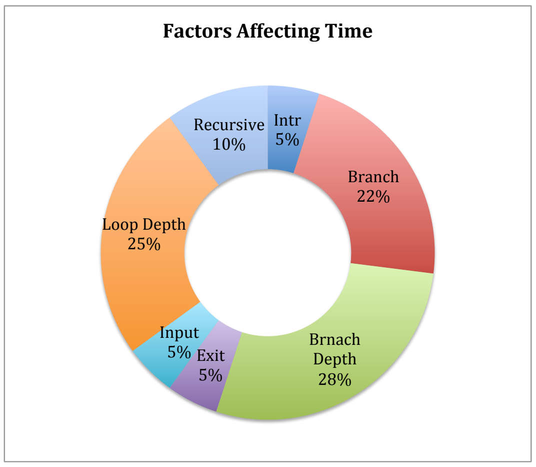

Figure 5: Percentage of applications benefited from changing different program features for test *time* improvement.

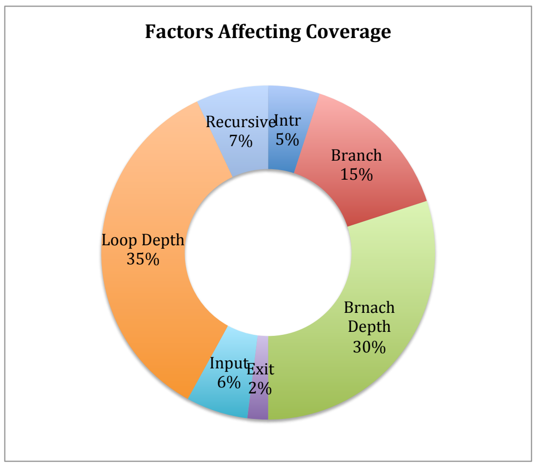

Figure 6: Percentage of applications benefited from changing different program features for test *coverage* improvement.