Understanding Intended Behavior using Models of Low-Level Signals

Deborah S. Katz
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, United States of America
dskatz@cs.cmu.edu

ABSTRACT

As software systems increase in complexity and operate with less human supervision, it becomes more difficult to use traditional techniques to detect when software is not behaving as intended. Furthermore, many systems operating today are nondeterministic and operate in unpredictable environments, making it difficult to even define what constitutes correct behavior. I propose a family of novel techniques to model the behavior of executing programs using low-level signals collected during executions. The models provide a basis for predicting whether an execution of the program or program unit under test represents intended behavior. I have demonstrated success with these techniques for detecting faulty and unexpected behavior on small programs. I propose to extend the work to smaller units of large, complex programs.

CCS CONCEPTS

• **Software and its engineering → Software reliability**; **Software testing and debugging**;

KEYWORDS

Software quality, Software testing, Oracle problem

ACM Reference format:
Deborah S. Katz. 2017. Understanding Intended Behavior using Models of Low-Level Signals. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17-DOC),* 4 pages.
https://doi.org/10.1145/3092703.3098237

1 INTRODUCTION AND MOTIVATION

Autonomous systems are big, complex, and difficult for humans to supervise, and are an increasingly large portion of the systems being developed and in use today. These systems are used in situations that can make it difficult for humans to observe them and deliver commands to them – such as systems in space – and in safety-critical situations, such as large, semi-autonomous vehicles operating in the presence of pedestrians. In these cases, it is important to figure out if there is a failure in the software execution as soon as possible. Software failures, for example the storage of a NaN instead of a value, may occur long before their results are easily observed by traditional means. By detecting the failure early, it may be possible to take corrective action before the possibly catastrophic result of that failure. For example, an arithmetic calculation may cause a program unit to crash, which may cause problems in other program units; or an autonomous vehicle may fail to detect a pedestrian in its way.

It is difficult to figure out if programs are failing, especially when the programs in question are big and complicated programs that people do not supervise directly or if there are portions of the program into which people do not see directly. Systems designed to behave autonomously, reacting independently and without human supervision to various stimuli in the environment, present a particular challenge in determining whether they are behaving as intended. These systems grow extremely complex because they are designed to react to all possible scenarios, including ones that the humans designing the systems could not have anticipated. In many circumstances, there is no precise definition of correct behavior, beyond the broad guidelines that the designers intended to incorporate. For example, an autonomous robot may be designed to adhere to broad safety principles, such as that it must not collide with an object, but the idea of what constitutes a correct path for the robot to take is much looser.

My key insight is that low-level information about the behavior of an executing program gives a picture of the characteristics of that execution. Aggregating the data over many executions of the same code, allows a more complex and nuanced picture.

My primary hypothesis is:

- Low-level execution signals on multiple executions of a program or portion thereof can be used to create a model to predict whether signals from previously-unseen executions represent usual or unusual behavior.

This work builds on earlier work in software testing, which addresses the problem of trying to figure out whether a program is behaving as intended, the *oracle problem.* This problem has been studied in a variety of contexts [1, 6, 11, 12, 14, 16]. Many of these approaches attempt to determine what a program should do by inferring invariants, generating a semantic notion of what the code should do. Also, many of these approaches require source code, which is often not available in complex or autonomous systems. Even in the cases when the user does have source code available, it is rare to have a user who understands the semantic intent and behavior of every portion of the source code. While these approaches

ISSTA'17-DOC, July 2017, Santa Barbara, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3098237

may be useful in certain contexts, many future-generation systems require a broadly-applicable approach that does not require source code or a semantic understanding of what a program should do.

Others have also proposed creating models of execution correctness [2, 4, 7, 8, 10, 15], but their techniques all differ from mine in important respects. My technique uses a broader range of underlying data and is more generalizable because the data I collect can be collected on a broad range of programs. In addition, some focus on building Markov models, which are powerful, but limited in scalability. Additional work looks at anomaly and intrusion detection in a security context, using related techniques [3, 5]. The anomaly and intrusion detection work looks at different execution properties than those I measure, some of which requires a more detailed control- and data-flow analysis of the program under test.

2 KEY INSIGHT AND APPROACH

Low-level information about the behavior of an executing program gives a picture of the characteristics of that execution. Low-level information about many executions of the same program can provide the basis for a more complex model of the program's possible behaviors. I propose to collect relevant low-level information about many executions, create a model based on the low-level data, and use that model to predict whether a new execution fits into the range of expected program behaviors or whether it represents an unexpected or unintended behavior. While not all rare behaviors are unintended, unintended behaviors are more likely to be rare.

I propose to use this technique to build models of intended and unintended behavior – using supervised machine learning based on known examples of intended and unintended behavior – and models of usual and unusual behavior – using unsupervised machine learning for anomaly detection. When given similar execution data for a previously-unseen execution, these models can predict whether that execution corresponds to unintended or unusual behavior. This knowledge can alert a human or an automated damage-control tool that intervention may be required.

To gather the data to build these models, I collect many low-level signals reflecting characteristics of program behavior for each execution of a program or portion of a program.

3 PRELIMINARY WORK

To validate these insights, I conducted preliminary work on small programs, to see if models built from low-level execution data could predict whether executions passed or failed. I recorded runtime data on various executions of each program, using several test inputs, including at least one that corresponded to failing or unintended behavior and at least one that corresponded to passing or intended behavior. I included executions of multiple versions of each program under test. I used the collected signals to build models of behavior.

I use Pin[1], a dynamic binary instrumentation toolkit distributed by Intel, to create tools to collect the low-level signals on running programs. I collected data on several versions of each program, each running with multiple test inputs. For each execution, I collected 165 low-level signals, later using feature extraction to choose 15 that were the most broadly relevant. I used the data for both supervised and unsupervised machine learning.

[1]https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

Table 1: Benchmark programs, from the Siemens SIR data set. Lines of code is the number of lines of code in a correct variant; number of variants is the number of unique versions of each program; and number of test cases is the number of separate test cases we use for each.

Program	Lines of Code	Number of Variants	Test Cases
SIR artifacts			
printtokens	475	8	4130
printtokens2	401	10	4115
replace	514	33	5542
schedule	292	10	2650
schedule2	297	11	2710
tcas	135	42	1608
totinfo	346	24	1052

Examples of the 15 most relevant signals include: the number of unique instructions executed, the mean of all addresses read, the address of the most frequent stack write, and the mean distance between a read an the next write.

For supervised machine learning, I gave a portion of the data points, along with labels indicating if each data point represented a correct or incorrect execution, to a supervised learning algorithm, which built a predictive model.[2] I then tested the predictive model on the remaining held-out data, by giving the model unlabeled data points and collecting the predictions of whether each data point corresponded to a correct or incorrect execution. I repeated the procedure using a standard 10-fold cross-validation technique.

For unsupervised machine learning, I built models without using any labels. These models use simple outlier-detection techniques to predict whether data points represent executions outside the realm of what a program usually does.

I used several suites of small benchmark programs common in testing research to validate the insights in this work. Here I present results from the Software-artifact Infrastructure Repository (SIR) Siemens objects.[3] Table 1 lists the lines of code, numbers of test cases, and number of variants for each program.

All experiments used a four core virtual machine running Ubuntu 14.04, without address space layout randomization. The experiments use Pin 2.13 and the Scikit Learn 0.15.2[4] package for model construction and evaluation [9, 13]. I compiled the programs under test using gcc version 4.8.2 for target x86_64-linux-gnu with settings as defined by the programs' makefiles.

Table 2 shows results for supervised learning on the data collected from executions of these programs, using standard machine learning assessment metrics.

[2]I conducted experiments both with the raw data set – which contained many more passing executions than failing executions – and an artificially balanced data set – built by leaving out data points in the set of passing executions. While I achieved comparable results for both techniques, I report the balanced results here.
[3]http://sir.unl.edu/portal/index.php
[4]http://scikit-learn.org/

Table 2: Left side: results of decision tree model using 165-feature set. Center: results of decision tree model using 15-feature set. Right side: results of SVM model using the 15-feature set.

	Full feature set (DT)				Core feature set (DT)				Core feature set (SVM)			
program	*Acc*	*Prec*	*Rec*	*FM*	*Acc*	*Prec*	*Rec*	*FM*	*Acc*	*Prec*	*Rec*	*FM*
printtokens	0.79	1.00	0.79	0.88	0.76	0.99	0.77	0.87	0.80	1.00	0.81	0.89
printtokens2	0.83	0.99	0.83	0.90	0.80	0.99	0.81	0.89	0.80	0.99	0.80	0.88
replace	0.85	1.00	0.85	0.92	0.85	1.00	0.85	0.92	0.67	0.99	0.67	0.80
schedule	0.76	0.99	0.76	0.86	0.74	0.99	0.74	0.85	0.56	0.99	0.55	0.71
schedule2	0.86	1.00	0.86	0.92	0.81	1.00	0.81	0.90	0.57	1.00	0.57	0.72
tcas	0.83	1.00	0.83	0.90	0.80	1.00	0.80	0.89	0.66	0.99	0.66	0.79
totinfo	0.90	0.99	0.90	0.94	0.91	0.99	0.91	0.95	0.77	0.98	0.77	0.86

- **True Positives** *(TP)* correct predictions of errors
- **True Negatives** *(TN)* correct predictions of no errors
- **False Positives** *(FP)* incorrect predictions of errors
- **False Negatives** *(FN)* incorrect predictions of no errors
- **Accuracy** *(Acc)* The portion of samples predicted correctly
- **Precision** *(Prec)* The ratio of returned labels that are correct: $TP/(TP + FP)$.
- **Recall** *(Rec)* The ratio of true labels that are returned: $TP/(TP + FN)$.
- **F-Measure** *(FM)* The harmonic mean of precision and recall.

The results are fairly comparable between the 15 feature set and the full 165 feature set, with the full feature set performing slightly better for many programs. The right side of Table 2 shows the outcomes on the fifteen reduced signals with a support vector machine classifier instead of a decision tree. The classifier is Scikit Learn's SVC classifier with cache size 1000 and all other parameters set to their defaults. As one can see by comparing the right side of Table 2 to the center, the decision tree classifier outperforms the support vector machine classifier for most programs. It is possible that the SVM classifier could be made to perform better with proper tuning but I leave that to future work.

4 PROPOSED FUTURE WORK

My preliminary results validate that the technique of building models of low-level execution characteristics has predictive power for distinguishing correct from incorrect executions. The preliminary work also brings up challenges in how to develop these techniques into practical analyses.

I am currently working to extend these techniques in a variety of ways. The major thrust is an extension to small units of large, real systems. In addition, I am working to make the predictions more useful to humans and automated systems that might use them; to get a better picture of what low-level information is useful in different circumstances; and to gain more information about the nature of program behavior, building on the same base data.

4.1 Extending to Large, Real Systems

The techniques I propose require several modifications to be useful in the larger programs that are commonly in use today and that are in most need of assistance in detecting when unintended behavior has occurred.

For example, in a large, complex program, the variety of control flow possibilities means that measurements of low-level data over the whole program in different executions will be vastly different, depending on the control flow that the program follows. The diversity of data values for reasons unrelated to correctness reduces the predictive power of any model built from that data.

Therefore, instead of instrumenting a whole program, I propose to isolate portions of programs for instrumentation and modeling. One way to do this is to leverage Pin's ability to identify routines and limit instrumentation to specific routines. By recording information separately for every execution of a particular routine, each execution becomes its own data point, and I can develop a model of intended behavior based on these data points.

Similarly, I can choose units of code to analyze, based on an address range in the binary. I can record data whenever those instructions are executed. Each execution of those instructions (or a portion thereof) becomes a data point on which to build a model.

The approach of instrumenting smaller units has the benefit of scalability. It can gather meaningful data on complex programs and has less overhead than instrumenting entire programs. There are also challenges. To meaningfully restrict instrumentation, you need to know which portion of the binary is relevant to possible unintended behavior. I have been working with portions of programs that are known to have problems, for proof-of-concept work. While it is much easier to identify the relevant binary if you have access to source code and can compile the programs with debug symbols, it is possible to use other techniques that do not require such visibility. I am also facing and addressing issues that arise because many large, complex programs have timing assumptions and may exhibit different behaviors when portions of the program run more slowly under instrumentation. I am working on solutions to mitigate these challenges in a generalizable and scalable way.

4.2 Other Directions for Extending the Work

4.2.1 Categorizing Program Behavior. My initial experiments focus on identifying unintended or anomalous behavior. However, programs exhibit many different behavior patterns. The data for many executions of the same program or program section show clustering or other patterns. I would like to investigate whether these clusters or patterns represent any particular types of behaviors. For example, can we use clusters to identify different types of errors in executions? (e.g., memory leak, infinite loop, calculation outside of usual range) Can we identify when a program unit is exercising different types of intended behaviors?

4.2.2 *Usability of Output.*

In general, my techniques predict whether a given instrumented execution fits the pattern of earlier instrumented executions. I have been using anomalous behavior as a proxy for unintended behavior. However, not all anomalous executions are actually incorrect – they may simply execute an uncommon but correct behavior. I would like to investigate how to improve the usefulness of the information presented to the developer or automated tool when my models predict unintended behavior. This might include confidence that the behavior is unintended, the possible type of error indicated, the region of the program that manifested the error, and other relevant data.

4.2.3 *Identifying Important Low-level Information.*

My initial experiments have found that different signals are important to determining whether a program is behaving as intended, depending on the program analyzed. I would like to examine which signals are most important for different programs and program portions, to try to draw conclusions about which signals are most important to collect in different circumstances. Because data collection at the binary level involves significant overhead, limiting data collection to only the most useful signals may improve efficiency.

5 EVALUATION STRATEGIES

My goal is to make useful predictions of incorrect behavior on complex systems before the incorrect behavior causes problems. I am currently expanding the work to test open-source robotics projects, usually based on the ROS platform, for example the ROS-based Husky project[5]. These projects present many interesting complexities, such as time-sensitivity, reliance on message passing and distributed state, and nondeterminism. I am beginning by validating the technique on units known to contain errors, exhibited in some but not all executions. When my technique can detect with reasonable accuracy whether an execution exhibited incorrect behavior on the units known to contain errors, I will expand to analyzing units not known to contain errors, to see if the technique can find errors or anomalies where none were known before.

To evaluate success on units with known errors, I compare whether my model predicts that the executions where the error occurred were erroneous. More broadly, I collect true positives, true negatives, false positives, false negatives, precision, recall, and F-measure, as explained in Section 3.

For situations where it is unknown whether a particular portion of a program contains an error and under which circumstances it may manifest, I will subjectively assess the executions that my model determines to be unusual behavior, to see if they represent unintended behavior. I will also present these assessments to developers familiar with the project under test to see if they can determine if the execution was correct. I will use any input from these assessments to refine my model building technique, so that the output is more useful.

In addition, I will use the machine learning techniques of feature selection, clustering, and dimensionality-reduction to gain more knowledge of the behavior of the programs under test, such as which signals are most indicative of different kinds of errors, which signals are most important to collect in different circumstances, and whether program behavior can be classified into categories, based on the clustering it exhibits.

[5]http://wiki.ros.org/Robots/Husky

6 CONCLUSIONS

In conclusion, I propose a novel technique for detecting runtime errors in difficult to understand systems. This approach has benefits particularly applicable to the types of complex and autonomous software systems that are becoming more prevalent. These benefits include early detection of potential issues not immediately obvious to humans in complex systems, so that corrective action is possible.

ACKNOWLEDGMENTS

This work was funded in part by the U.S. Air Force under contract #FA8750-15-2-0075. Any opinions, findings, or recommendations expressed are those of the author.

REFERENCES

[1] Earl T. Barr, Mark Harman, Phil Mcminn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing : A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[2] James F. Bowring, James M. Rehg, and Mary Jean Harrold. 2004. Active Learning for Automatic Classification of Software Behavior. In *International Symposium on Software Testing and Analysis (ISSTA '04)*. 195–205.

[3] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. 2007. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *USENIX Security Symposium (SS'07)*. 15:1–15:16.

[4] Yuriy Brun and Michael D. Ernst. 2004. Finding Latent Code Errors via Machine Learning over Program Executions. In *International Conference on Software Engineering (ICSE '04)*. 480–490.

[5] Dorothy E. Denning. 1987. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering* 13, 2 (1987), 222.

[6] Gordon Fraser and Andreas Zeller. 2012. Mutation-Driven Generation of Unit Tests and Oracles. *Transactions on Software Engineering* 38, 2 (2012), 278–292.

[7] Murali Haran, Alan Karr, Michael Last, Alessandro Orso, Adam A. Porter, Ashish Sanil, and Sandro Fouche. 2007. Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks. *IEEE Transactions on Software Engineering* 33, 5 (2007), 287–304.

[8] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach. In *Knowledge Discovery and Data Mining (KDD '09)*. 557–566.

[9] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation (PLDI '05)*. 190–200.

[10] Chengying Mao and Yansheng Lu. 2005. Extracting the Representative Failure Executions via Clustering Analysis Based on Markov Profile Model. In *Advanced Data Mining and Applications (ADMA '05)*. 217–224.

[11] Carlos Pacheco and Michael D. Ernst. 2005. Eclat: Automatic Generation and Classification of Test Inputs. In *European Conference on Object-Oriented Programming (ECOOP '05)*. 504–527.

[12] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE '07)*. Minneapolis, MN, USA, 75–84.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.

[14] Mauro Pezze and Cheng Zhang. 2014. Automated Test Oracles: A Survey. *Advances in Computers* 95 (2014), 1–48.

[15] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. 2003. Automated Support for Classifying Software Failure Reports. In *International Conference on Software Engineering (ICSE '03)*. 465–475.

[16] Tao Xie. 2006. Augmenting Automatically Generated Unit-test Suites with Regression Oracle Checking. In *European Conference on Object-Oriented Programming (ECOOP'06)*. 380–403.