

TRIPWIRE: Mediating Software Self-Awareness

James F. Bowring James M. Rehg Mary Jean Harrold

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
{bowring, rehg, harrold}@cc.gatech.edu

Abstract

We propose TRIPWIRE as a framework that provides for the mediation of software self-awareness by supporting real-time assessment and response capabilities. Our approach is inspired by the recent success of automatic speech recognition systems, which can assess the likelihood of a potentially unbounded set of possible utterances and select the most likely candidate in real-time, given an underlying model of the conversational domain. We see an analogy between estimating an utterance from an acoustic waveform and estimating the behavior of a program from dynamic-analysis data. In both cases, there is an inherently stochastic causal relationship between the quantity of interest and the measurement data. Our strategy is to leverage the successful tools and methods of speech recognition in the novel context of program behavior analysis. TRIPWIRE technology provides a systematic approach to behavior modeling and recognition with three main elements: strong domain knowledge and tools, learned statistical models, and real-time classification processes.

1 Introduction

The explosion of software into the fabric of our environment has outpaced the ability of humans to maintain and manage digital systems. The recent example of the malfunction of software aboard the Mars rover Spirit is evidence that software engineering and software management remain problematic even when vast human resources are available. One promising approach to this dilemma is to develop meta-strategies for self-awareness by software systems that enable self-diagnosis and self-repair. IBM's initiatives with autonomic computing address this idea for the management of systems on a large scale [16]. However, self-awareness begins at home, so to speak, and little has been done to provide for the self-awareness of individual

programs such as those driving robots and other scientific devices or even those in a desktop environment.

In current practice, outputs produced by the execution of terminating programs can be intercepted for data collection and analysis. For example, the Internet browsers of Microsoft and Netscape provide problem-reporting facilities invoked before the browsers terminate. Self-aware systems have been studied in robotics, control systems, software architecture, and fault-tolerant computing. The research approaches generally focus on the creation of high-level models to assess program state and behavior and then to provide mechanisms for re-configuration either for repair or for self-improvement (e.g., [7, 8, 12, 22]).

One metaphor proposed for autonomic systems involves training software to be responsive to its own pulse and other runtime health-markers, just as humans are aware of their own temperature, pulse, and breathing rates when they differ from the norm. Humans become more self-aware with the feedback from electronic monitors attached to their bodies. These monitors act as sentinels or *tripwires* to alert the user and possibly to activate remedial strategies. Consider software for a robot. If the software were aware of when it and the robot were deviating from a set of behavioral norms and could take action to avoid a catastrophe, as with a well-placed tripwire, the benefits would be substantial.

There is an emerging consensus that the development of self-awareness for software will enhance dependability and safety for a wide range of applications. There are two aspects to self-awareness: a set of baseline models that characterize the historical behavior of program executions, and a real-time analysis component that can assess deviations from these behaviors during execution. Thus self-awareness can be posed in terms of two basic questions:

- How can we efficiently and effectively extract behavior models from historical data?
- How can we enable running software to evaluate its own behavior in terms of these models?

Our goal is to develop a framework for software self-awareness, named TRIPWIRE, that supports real-time assessment of software behaviors and real-time responses to self-assessment. TRIPWIRE embodies a systematic approach to behavior modeling and recognition with three main elements: strong domain knowledge and tools, learned statistical models, and real-time classification processes. Our approach is inspired by the recent success of automatic speech recognition systems, which can assess the likelihood of a potentially unbounded set of possible utterances and select the most likely candidate in real-time, given an underlying model of the conversational domain. We see an analogy between estimating an utterance from an acoustic waveform and estimating the behavior of a program from dynamic-analysis data. In both cases, there is a causal relationship between the quantity of interest (utterance or behavior) and the measurement data. But this relationship is inherently stochastic, due to the richness and variability of the measurement set. Moreover, like the set of possible English sentences, the set of program behaviors is too large to model explicitly in all but the most trivial cases.

Our strategy is to leverage the successful tools and methods of speech recognition in the novel context of program behavior analysis. We expect the synergies that result from our interdisciplinary approach to yield new perspectives on program behavior and new insights into the capabilities of these statistical models. In particular, we will leverage the tools of mature software development processes, such as operational profiling [19], in conjunction with static and dynamic program analysis [2]. We will combine these analysis tools with machine learning techniques for characterizing and modeling program executions. A wide range of powerful modeling tools have been developed for temporally-sequenced data. Some examples include Markov chains, hidden Markov models, simplicial mixtures of Markov chains, and dynamic Bayesian networks. We have recently demonstrated the potential value of this combination of disciplines, through our application of active learning [5] techniques to build behavior classifiers from Markov models derived from program executions [3].

2 Background and Related Work

Other researchers have also begun to explore the use of statistical modeling and machine learning techniques to extract behavior summaries from execution data. For example, Podgurski and colleagues [9, 20] use clustering techniques to build statistical models from program executions and apply them to the tasks of fault detection and failure categorization. Dickinson, Leon, and Podgurski demonstrate the advantage of automated clustering of execution profiles over random selection for finding failures [9].

Podgurski and colleagues use many kinds of feature pro-

files as the basis for cluster formation. However, they do not isolate or evaluate the effects of individual features on the construction of the models and the recognition of behaviors. Our approach differs from theirs in that we will explore the utility of using a restricted subset of all possible features of program executions instead of a large set of features. Our goal is to understand in detail the power of each feature as a predictor of certain behaviors.

We focus on the class of features that describe event-transitions in program executions. An *event-transition* is a transition from one program entity to another. Types of *1st-order* event-transitions include branches (source statement to sink statement), method calls (caller to callee), and definition-use pairs (definition to use). We examine in particular *event-transition profiles*, which record the frequency of occurrences of event-transitions during an execution. Podgurski and colleagues build classifiers based on logistic regression, whereas we leverage the stochastic nature of profiles of event transitions. We posit that, in general, event-transition features exhibit the *Markov property*—that the probability distribution of future states of a process depends only upon the current state. We show the utility of Markov models based on these profiles as predictors of program behavior.

The use of Markov models to describe the stochastic dynamic behavior of program executions is not new. Whitaker and Poore use Markov chains to model software usage from specifications prior to implementation [24]. Cook and Wolf confirm the power of Markov models as encoders of individual executions in their study of automated process discovery from execution traces [6]. In contrast, we will use Markov models to describe the statistical distribution of transitions measured from executing programs and thus to directly classify program behaviors.

Another category of related work uses a wide range of alternative statistical learning methods to analyze program executions. Although these works differ substantially from ours in detail, we share a common goal of developing useful characterizations of aggregate program behaviors. Harder, Mellen, and Ernst automatically classify software behavior using an operational differencing technique [15]. Their method extracts formal operational abstractions from statistical summaries of program executions and uses them to automate the augmentation of test suites. In comparison, our modeling of program behavior is based exclusively on the Markov statistics of events. Brun and Ernst use dynamic invariant detection to extract program properties relevant to revealing faults and then apply batch learning techniques to rank and select these properties [4]. However, the properties they select are themselves formed from a large number of disparate features and the authors' focus is only on fault-localization and not on program behavior in general. In contrast, we seek to isolate the features critical to de-

scribing various behaviors. Additionally, we apply active learning to the construction of the classifiers in contrast to the batch learning used by the authors.

Sekar and colleagues have proposed a mechanism to implement model-carrying code [21] focused on high-level models of security-relevant behavior. They propose techniques to automatically extract models from system call histories external to the program itself using finite state automata. They propose that the policies derived from their models be enforced at the kernel level. In contrast, we use events internal to the program to extract behavior models and propose that they be used at the application level.

Researchers who have used machine learning techniques for recognizing behaviors or other related tasks, uniformly employ batch-learning techniques (e.g., [1, 4, 9, 17, 14, 18, 20]). In *batch-learning*, a fixed quantity of manually-labeled training data is collected at the start of the learning process. In contrast, our research uses an active-learning paradigm [5] for behavior classification. In *active learning*, the classifier is trained incrementally on a series of labeled data elements. We have demonstrated the advantages of applying active-learning techniques to the automation of classifier construction from models of individual executions.

3 Discussion

Statistical model-building is an art, and success is highly dependent on domain knowledge and experimentation. In particular, it is critical for the model-builder to choose meaningful data representations to study. There is a wide variety of data that can be collected from executing programs. For example, the probes can monitor the execution environment, network traffic, or actual events that occur inside the running program. We concentrate on the latter, as our initial focus is on the behavior of individual programs. We are specifically interested in the class of features that describe event-transitions in program executions. These include branches as well as other transitions such as definition-to-use of variables. We will explore combinations of these features to determine whether feature compositions can improve the models. We will investigate the use of more complex data features, such as event-transition paths of length two or more. We use profiling mechanisms to collect frequency data for these transitions between events for use in our models. Therefore, an important aspect of this research is to examine the trade-offs between the costs of collecting higher-order event-transition profiles and the benefits provided by their predictive abilities.

Markov models are a popular and effective tool for modeling temporal data. Our previous work suggests that Markov models built from event transitions are reliable predictors of program behavior. We will continue to explore the use of Markov models. In addition, we will explore the

use of hidden Markov Models [11] and simplicial mixtures of Markov models [13]. In a *hidden Markov model*, the state variable in the Markov chain is assumed to be unobservable (i.e. cannot be measured directly). However, at each time instant the hidden state emits an observation variable which can be measured. The HMM specifies a statistical relationship between the hidden state and the observation, in addition to the usual transition probabilities for the hidden state. Once the parameters of the model have been learned from data, powerful inference techniques, such as the Viterbi algorithm, can be used to estimate the most likely sequence of hidden states given a sequence of observations. Learning in the HMM case uses the Expectation-Maximization technique citeduda2001, which is more complex than standard parameter estimation for a Markov chain. However, as a consequence of their widespread use in automatic speech recognition, there is an established body of practice for training HMMs which can be leveraged for our application.

HMMs are interesting from two perspectives. The first is that they can describe more complex probability distributions than simple Markov chains. If branch transitions are taken as the states in a Markov chain, then their statistical distribution would follow standard first- or second-order dependencies. However, if these transitions are taken as the observations in an HMM with a first- or second-order hidden-state dependency, marginalization of the hidden state induces much more complex distributions over the branches. In particular, the branch distribution does not have to obey the Markov property. In this context, the hidden states could correspond to higher-level categories of transitions.

The second potential benefit of HMMs is their ability to couple multiple aspects of program execution together in one statistical model. For example, advanced HMM models such as factorial HMMs, input-output HMMs, and conditional random fields [10] support more complex coupling between execution events and associated data such as program inputs and outcomes. These advanced tools may enable more powerful models of program behaviors.

Simplicial mixtures of Markov models are another recent generalization of basic Markov chains which have demonstrated an ability to capture common behavioral patterns in event streams. As data from program executions are also event streams, there is a real possibility of adding this technique to our family of statistical methods.

While we have demonstrated in our previous work the potential of Markov models as effective predictors of passing and failing behaviors, we are ultimately seeking much more granular definitions of behavior such as might be depicted in software requirements. In the limit, every unique execution may be said to represent a unique behavior, however, this produces an intractably large population of behavior models. Thus, the application of machine learning and

automated classification techniques to this population of models is intuitively attractive. In order to assess the quality of this applicability, however, we will need to demonstrate the existence of mappings from statistical models of behaviors to human-recognizable representations of the same behaviors.

A key issue in achieving software self-awareness through learned models is to devise efficient mechanisms for *provisioning* software with these models. We see an analogy to the deployment of speech recognition technology on hand-held devices, which faces similar trade-offs between computation, power, and accuracy for classification methods [23]. These tradeoffs can be addressed in two ways. The first is by tuning the algorithm. The second is to partition the computation between the client and a server. We will investigate these alternatives in the context of our TRIPWIRE system. Another aspect of provisioning for self-awareness is scalability. Here, potential solutions involve reducing the size of the models either by abstraction or by concentration on particular behaviors, depending on need.

References

- [1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proceedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 4–16, January 2002.
- [2] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.
- [3] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, July 2004. (To appear).
- [4] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, pages 2–9, May 2004.
- [5] D. A. Cohn, L. Atlas, and R. E. Ladner. Improving generalization with active learning. *Machine Learning*, 15(2):201–221, 1994.
- [6] J. E. Cook and A. L. Wolf. Automating process discovery through event-data analysis. In *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*, pages 73–82, January 1999.
- [7] C. Dabrowski and K. Mills. Understanding self-healing in service-discovery systems. In *Proceedings of the first workshop on Self-healing systems*, pages 15–20. ACM Press, 2002.
- [8] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 21–26. ACM Press, 2002.
- [9] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 339–348, May 2001.
- [10] T. G. Dietterich. Machine learning for sequential data: A review. In T. Caelli, editor, *Structural, Syntactic, and Statistical Pattern Recognition*, volume 2396 of *Lecture Notes in Computer Science*, pages 15–30. Springer-Verlag, 2002.
- [11] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley and Sons, Inc., New York, 2001.
- [12] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 27–32. ACM Press, 2002.
- [13] M. Girolami and A. Kaban. Simplicial mixtures of markov chains: Distributed modelling of dynamic user profiles. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.
- [14] K. C. Gross, S. McMaster, A. Porter, A. Urmanov, and L. Votta. Proactive system maintenance using software telemetry. In *Proceedings of the 1st International Conference on Remote Analysis and Measurement of Software Systems (RAMSS'03)*, pages 24–26, May 2003.
- [15] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25rd International Conference on Software Engineering (ICSE'03)*, pages 60–71, May 2003.
- [16] IBM Research. Autonomic computing, 2004. <http://www.research.ibm.com/autonomic/>.
- [17] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *Software Engineering*, 21(3):181–199, 1995.
- [18] S. Jha, K. Tan, and R. A. Maxion. Markov chains, classifiers, and intrusion detection. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 206–219, June 2001.
- [19] J. Musa. *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. McGraw-Hill, New York, 1999.
- [20] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25rd International Conference on Software Engineering (ICSE'03)*, pages 465–474, May 2003.
- [21] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 15–28. ACM Press, 2003.
- [22] M. Shaw. “self-healing”: softening precision to avoid brittleness: position paper for woss '02: workshop on self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 111–114. ACM Press, 2002.
- [23] J. H. U. Kremer and J. M. Rehg. Compiler-directed remote task execution for power management. In *Proc. Workshop on Compilers and Operating Systems for Low Power*, Philadelphia, PA, October 2000.
- [24] J. A. Whittaker and J. H. Poore. Markov analysis of software specifications. *ACM Transactions on Software Engineering and Methodology*, 2(1):93–106, January 1996.