Thesis Proposal: Identification of Software Failures in Complex Systems Using Low-Level Execution Data

Deborah Stephanie Surden Katz

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Claire Le Goues, Chair
Philip Koopman
Eric Schulte, Grammatech, Inc.
Dan Siewiorek

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: Software quality; Software testing; Autonomous systems; Robotics; Oracle problem

Abstract

Autonomous systems – systems that are designed to react independently and without human supervision to various stimuli in the environment – are big, complex, and difficult for humans to supervise and are an increasingly large portion of the systems being developed and in use today. In many of these systems, early knowledge of a fault would allow costly and safety-critical failures to be avoided.

My key insight is that I can look to typical program behavior as a basis for determining whether a program is operating within its normal parameters. To do this, I can record summaries of program behavior using low-level execution data to characterize each execution. By aggregating low-level execution data over many executions, I can create a picture of typical program behavior and suggest that a program behaving differently may be exhibiting unintended behavior. My techniques use the collected data as input to machine learning algorithms which build models of expected program behavior. I use these models to analyze individual program executions and make a prediction about whether the given execution represents typical behavior.

My core thesis is: Low-level execution signals recorded over multiple executions of a robotics program or portion thereof can be used to create machine learning models that, in turn, can be used to predict whether signals from previously-unseen executions represent usual or unusual behavior. The combination of low-level instrumentation and models can provide predictions with reasonable trade-offs between prediction accuracy, instrumentation intrusiveness, and calculation efficiency.

To support this thesis I propose the following work. The preliminary work, which is complete, uses dynamic binary analysis on small programs to construct supervised and unsupervised machine learning models to detect program executions that exhibit errors. Additional proposed work that is complete evaluates instrumented simulations of the ARDUPILOT autonomous vehicle software to construct machine learning models to identify executions with errors.

I further propose to evaluate novelty detection techniques on varied robotics programs. I then propose to refine my techniques for instrumenting program executions by evaluating trade offs in approaches to reduce overhead in instrumentation, a nontrivial problem because overhead can perturb execution paths in timing-sensitive programs. By reducing overhead but retaining meaningful information, I may be able to reach parts of programs previously unreachable by my instrumentation techniques.

I propose to evaluate the new work on several robotics and autonomous systems in simulation. I plan to evaluate accuracy in identifying program faults. I also plan to evaluate intrusiveness and efficiency of instrumentation and to evaluate the trade offs between these two sets of factors.

Contents

1 Introduction **1**
- 1.1 Thesis Statement . . . 5
- 1.2 Proposed Work . . . 6
  - 1.2.1 Preliminary Work . . . 6
- 1.3 Work to be Completed . . . 6

2 Review of Literature and Background **8**
- 2.1 Related Work . . . 8
- 2.2 Background . . . 12
  - 2.2.1 Dynamic Binary Instrumentation . . . 12
  - 2.2.2 Machine Learning Models . . . 13

3 Dynamic Binary Analysis to Detect Errors in Small Programs **16**
- 3.1 Motivating Example . . . 16
- 3.2 Approach . . . 17
  - 3.2.1 Dynamic execution signals . . . 18
  - 3.2.2 Model generation . . . 19
- 3.3 Experimental Design . . . 20
- 3.4 Results . . . 21
  - 3.4.1 Supervised Learning . . . 21
  - 3.4.2 Unsupervised Outlier Detection . . . 22
- 3.5 Limitations Suggestive of Future Directions . . . 23

4 Dynamic Binary Instrumentation to Detect Errors in Robotics Programs – ARDUPILOT **24**
- 4.1 The ARDUPILOT System . . . 24
- 4.2 ArduPilot Approach . . . 24
- 4.3 Experimental Setup . . . 25
  - 4.3.1 Supervised Machine Learning . . . 26
  - 4.3.2 Collecting Signals with Dynamic Binary Instrumentation . . . 26
- 4.4 Results . . . 27
  - 4.4.1 *RQ1:* Supervised Learning on a Single Defective Version of ARDUPILOT 27
  - 4.4.2 *RQ2:* Supervised Learning on a Defective Version of ARDUPILOT and Its Repaired Counterpart . . . 28

4.4.3 *RQ3:* Prediction Accuracy on Varied Amounts of Data 28

5 Novelty Detection on Varied Robotics Programs **31**
5.1 Method 31
5.2 Evaluation 32
5.3 Evaluation Subjects 32
5.4 Expected Contributions 32

6 Intrusiveness Reduction **33**
6.1 Analysis of Smaller Units In Robotics Software 33
6.1.1 Method 34
6.1.2 Expected Contributions 34
6.2 Signal Selection and Reduction 34
6.2.1 Method 35
6.2.2 Expected Contributions 35
6.3 Sampling 36
6.3.1 Method 36
6.3.2 Expected Contributions 36
6.4 Evaluation Metrics 36
6.5 Evaluation Subjects 37

7 Proposed Timeline **38**

8 Conclusion **40**

Bibliography **41**

1 Introduction

It is increasingly important to understand autonomous and robotics systems and guard against unintended behavior. Autonomous systems – systems that are designed to react independently and without human supervision to various stimuli in the environment – 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 [27]. One example is the ExoMars Mars Lander, which crashed on the surface of Mars. The consequences of the crash included $350 million in lost equipment and time. Due to the inaccessibility of the Martian environment, humans were unable to detect and repair the failure before the catastrophic consequences occurred. The problem was likely due to an implementation mistake that failed to account for timing inconsistencies between sensors [3, 38]. Although most autonomous systems do not operate on Mars, they generally: (1) are difficult to observe; (2) have properties that make it difficult to determine whether they are behaving correctly; and (3) can cause damage that is expensive or endangers human lives when they fail.

At the same time, detecting faults in complex systems that interact with the environment is complicated by several additional factors. Systems designed to behave autonomously present a particular challenge in determining whether they are behaving safely or as intended [28, 46]. These systems grow extremely complex because they are intended to react to all possible scenarios, including ones that the humans designing the systems could not have anticipated [45]. But while these systems should anticipate all possible scenarios, such anticipation is not always possible. Often, their operating parameters are poorly-defined at all but the most basic of levels. For example, one failure mode in an autonomous vehicle may be defined as a collision with a pedestrian; but the details of what constitutes a collision with a pedestrian in terms of the vehicle's internal parameters may not be well defined. In fact, if the vehicle fails to detect the pedestrian at all, there may be no internally-observable failure with respect to that failure mode [45].

In addition, because the environments in which these systems operate are complex and unpredictable, designers and testers may not have thought of all conditions that, if reached, would pose a safety risk. For example, some self-driving car companies have tried to create comprehensive testing environments by studying road configurations in different real locations and recreating on their test site and simulations each new configuration encountered. However, there may always exist combinations of configurations and situations that the companies have not encountered or fully encompassed in their testing [45, 56].

In many of these systems, early knowledge of a fault – along with tools that enable interven-

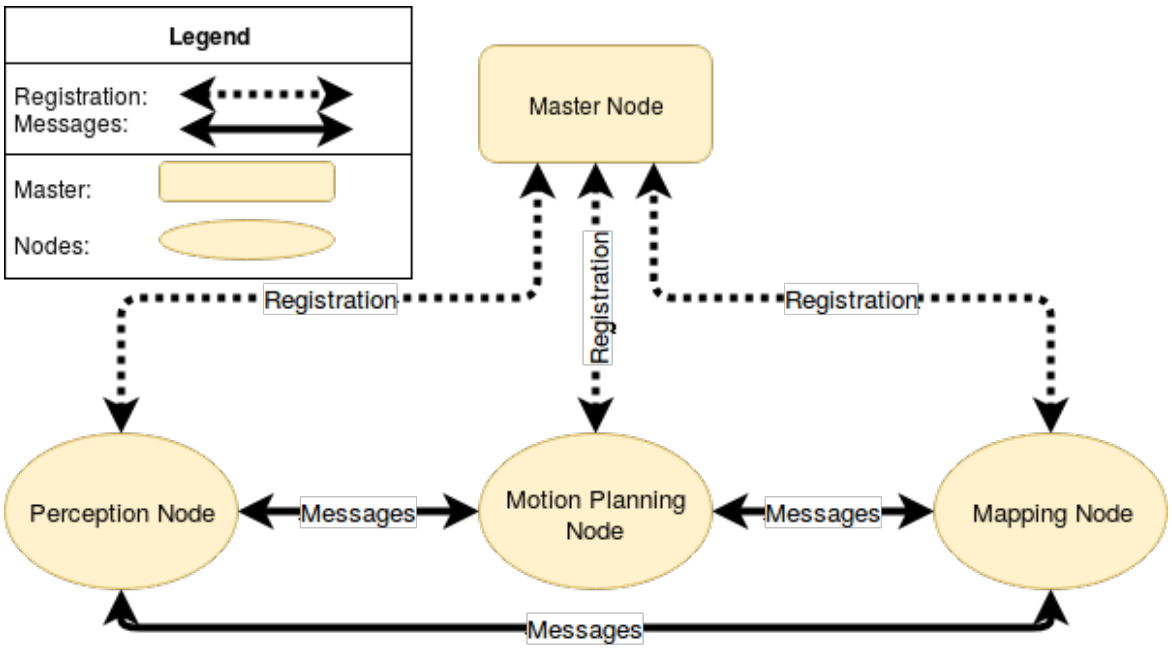

Figure 1.1: Simplified Robot Architecture

tion – would allow costly and safety-critical failures to be avoided [82]. It is, therefore, important to figure out if there is a failure in the software execution as soon as possible. It is possible for a software failure to occur long before the result causes a problem or can be observed by traditional means.

For example, consider the simplified program architecture outlined in Figure 1.1. The master node activates and registers various specialized nodes that handle various functions of the robot. These nodes communicate with each other by passing messages. In this example, the master node has activated three additional nodes: a perception node, a motion planning node, and a mapping node. In this example, the perception node interprets data from sensors such as cameras and calculates the locations of obstacles. The mapping node requests data from the perception node. On receiving this data, the mapping node converts the information about obstacles into a map of the robot's surroundings. The motion planning node requests this map, and the mapping node sends it. The motion planning node uses the map to calculate the intended path for the robot to take. The perception node updates with new information, and the other nodes periodically repeat their functions to keep their information current.

The following example is inspired by a real bug found in a robotics system that uses such an architecture. A program unit may do something undesirable, such as storing a value that is not a number ("NaN") instead of a value that the rest of the program can interpret, as in Figure 1.2. In this example sequence, a robot's perception node wrongly computes the Y-component of the location of an obstacle to be a NaN. At a later time, Time B, the mapping node requests the location of this obstacle. The perception node sends a message with the stored location values at Time C. The mapping node then tries to use the location values to compute a map at Time D and suffers a crash. Storage of this NaN occurred long before the program failure, and the program failure happened in a different program unit than the initial incorrect storage.

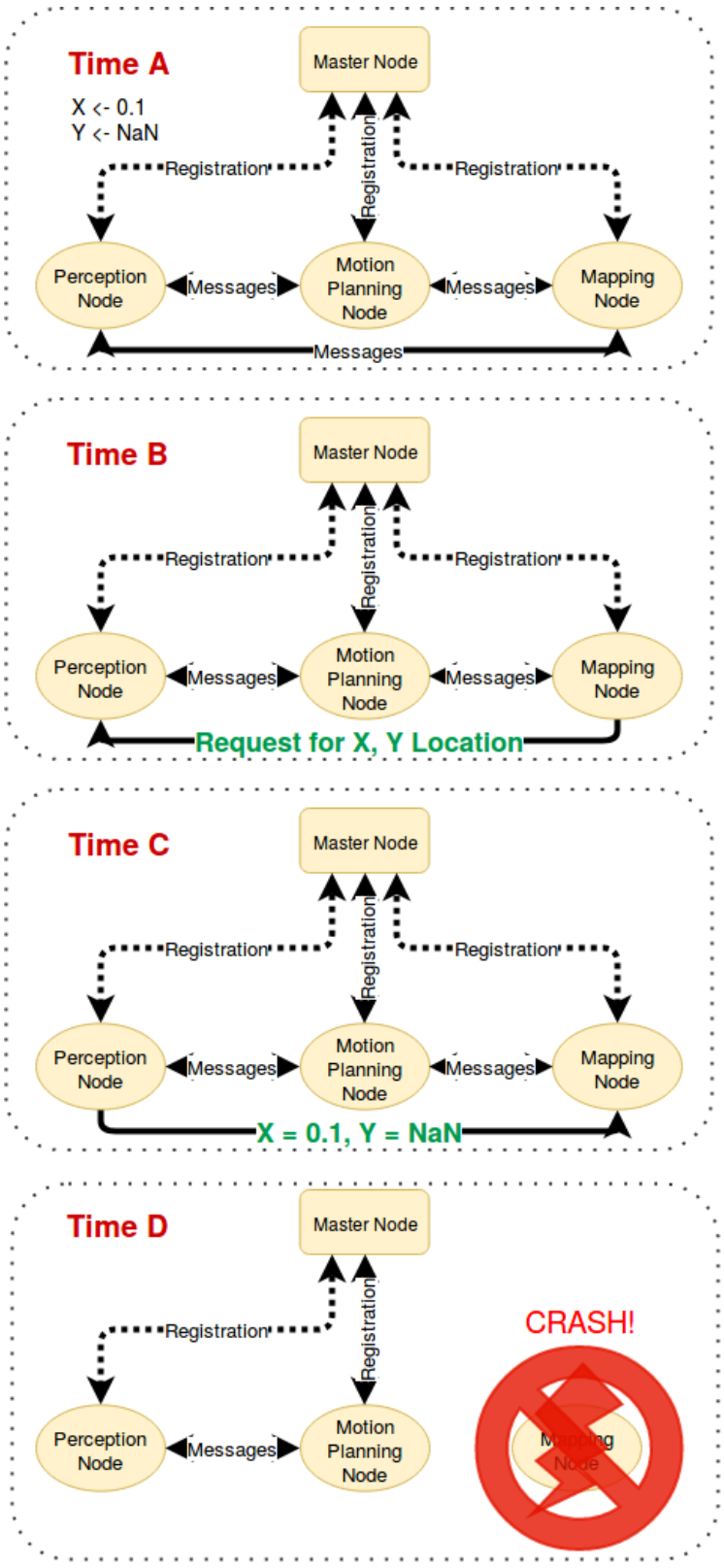

Figure 1.2: A Simplified Crash Sequence

In a real world situation, the resulting program failure might be benign or it may result in a chain of failures that causes catastrophic consequences, such as a vehicle crash or a collision with a pedestrian. If we can detect the unusual and undesired storage of the NaN before it causes a problem in other program units, it may be possible to take corrective action before the possibly-catastrophic result of the failure. While it may be easy to monitor specifically for NaNs in many circumstances, the example extends by analogy to other behaviors that are unusual in context.

Existing techniques for identifying bugs in software have limitations. I describe these techniques and their limitations in more detail in Section 2.1. These techniques include using a suite of test cases; using invariants, either manually-written or inferred; using statistical methods; program verification; and building formal models.

We do not always have source code for these systems or some of their sub-components [27]. Software that operates on embedded devices, such as much of autonomous and robotics software, is often distributed without source code, both because it contains proprietary information and because it is sometimes distributed on the resource-limited environments of the robots, systems, or subcomponents [27]. For tools to be effective on such systems, we need techniques that do not require access to source code. Although many of the systems on which I test my techniques do have source code available, I design the techniques to be applicable independent of source code or language.

My key insight is that I can look to typical program behavior as a basis for determining whether a program is operating within its normal parameters. To do this, I can record summaries of program behavior using low-level execution data to characterize each execution. By aggregating low-level execution data over many executions, I can create a picture of typical program behavior and suggest that a program behaving differently may be exhibiting unintended behavior. My techniques use the collected data as input to machine learning algorithms which build models of expected program behavior. I use these models to analyze individual program executions and make a prediction about whether the given execution represents typical behavior.

For an intuitive understanding of this approach, recall the example presented in Figure 1.2 and the accompanying text above. By monitoring low-level signals in a given node, such as the values stored to memory in the perception node, we have the opportunity to detect that the NaN stored represents an unusual behavior.

I propose a family of techniques to address some of the major and important challenges hindering effective testing autonomous and robotics systems. In summary, the techniques address the following challenges:

- It is difficult to analyze correct behavior on systems without source code.
- It is difficult to analyze correct behavior when expected system behavior is complex – possibly undefined or poorly-defined – and not fully-captured by test cases.

To address these challenges, I propose techniques that build a picture of expected execution behavior from observation of trends in execution behavior. I do this at a low level, which means observing behavior at the level of machine instructions and memory operations. While it may be possible to make useful observations at many levels of program behavior, such as interactions between program units or externally-observable output or behavior, low-level data provides several advantages. Notably, by monitoring various low-level signals, there is no need to choose which semantically-meaningful behaviors to observe. This property is useful because, in com-

plex robotics and autonomous software, one cannot assume knowledge of which behaviors are semantically meaningful. In addition, observing low-level behavior offers the opportunity to detect unusual behavior before it manifests in externally-observable consequences.

I combine data from several low-level signals as input to the machine learning models. This approach reflects the observation that no one signal captures every unusual behavior. In fact, some behavioral deviations may only be observable when more than one signal deviates together.

1.1 Thesis Statement

Thesis Statement: Low-level execution signals recorded over multiple executions of a robotics program or portion thereof can be used to create machine learning models that, in turn, can be used to predict whether signals from previously-unseen executions represent usual or unusual behavior. The combination of low-level instrumentation and models can provide predictions with reasonable trade-offs between prediction accuracy, instrumentation intrusiveness, and calculation efficiency.

By "low-level execution signals," I mean data about program execution as recorded by dynamic binary instrumentation. Such signals include information collected after individual machine instructions and do not require source code to collect. I elaborate on these signals and the methods by which I collect them in Section 2.2.1.

By, "useful," and, "reasonable," I mean that the accuracy, intrusiveness, and efficiency – as measured by the means defined below – fall within acceptable boundaries as suggested by the literature.

As an overall metric, "accuracy," measures whether something is correct. I construct machine learning models and procedures that predict whether particular executions correspond to either unusual (anomalous) or usual (nominal) behavior. Applied to this situation, "accuracy," refers to the extent to which the models and algorithms correctly determine whether particular executions correspond to either unusual or usual behavior. There are several accepted ways to measure accuracy. I propose to evaluate these experiments by the accuracy metrics defined in Section 2.2.2, with particular reference to precision, recall, and F-measure. *Precision* is an appropriate metric because it measures the extent to which the executions that the techniques flag as unusual actually exhibit abnormal behavior. Put another way, a precise algorithm does not identify many false positives, relative to the total number of executions it identifies as unusual. False positives would reduce the usefulness of the algorithm for any human or automated repair technique that consumes its output. *Recall* is an appropriate metric because it measures the extent to which the technique identifies all abnormal behavior and does not miss any. In other words, an algorithm with high recall does not fail to flag many instances of unusual behavior, relative to the total number of instances in the data set. Without high recall, the usefulness of the algorithm would be reduced because the algorithm could not be counted on to detect most of the behavior it is supposed to detect. *F-measure* balances precision and recall, so that neither metric is trivially maximized at the expense of the other.

By, "intrusiveness," I mean the degree to which the instrumentation that I use to collect execution perturbs program execution. For, "efficiency," I intend to measure the degree to which

instrumentation adds additional time to program executions. The less time added, the higher the efficiency. To measure these factors, I propose to evaluate the experiments with respect to the effect that instrumentation has on program execution. To study intrusiveness, I will look at program points reached at different levels of instrumentation over the course of many executions with the same inputs and environment. If the program regularly reaches different program points with different levels of instrumentation, I can conclude that the instrumentation is intrusive, in that it affects the program's execution direction. This metric is appropriate because the technique is most useful when it does not heavily influence the program's control flow behavior by adding overhead. To measure efficiency, I will also measure timing between equivalent program points at different levels of instrumentation, controlling for flow deviation from intrusiveness. This metric is appropriate because a technique with low overhead will be more likely to be adopted.

1.2 Proposed Work

1.2.1 Preliminary Work

I have completed work that demonstrates that my approach is reasonable to address the challenges I identified above.

I have used dynamic binary analysis in combination with machine learning to detect errors in small programs, as discussed in Section 3. I extended this work to detect errors in various versions of and scenarios in the popular ArduPilot autonomous vehicle software in Section 4.

1.3 Work to be Completed

In my preliminary work, I observed that the techniques I propose are broadly able to detect various program behaviors that deviate from expected or usual behavior. These techniques apply both to small programs and to the ArduPilot autonomous vehicle software operating in certain controlled circumstances. However, the preliminary work suggested certain limitations in the techniques as applied.

- One of the limitations is the limited set of programs on which I tested the techniques. In Section 5, I propose to address this limitation by expanding testing to additional robotics programs.
- One limitation is that the supervised learning techniques limit application to situations in which we have labeled data: data for which we know whether the executions were typical or anomalous. In Section 5, I propose to address this limitation by expanding use of novelty-detection and outlier-detection techniques, expanding them to additional circumstances.
- One of the limitations is that the instrumentation used in my techniques can perturb program execution. In Section 6, I propose to address this limitation by analyzing whether I can reduce overhead while maintaining meaningful results with signal selection and sampling and also by analyzing smaller program units within robotics software.

- An additional limitation is that larger trends in program behavior can obscure smaller fluctuations. In Section 6.1, I propose to address this limitation with the analysis, mentioned above, of smaller program units within robotics software.

In summary, I propose the following work:

- Establishing using dynamic binary analysis in conjunction with machine learning models as a technique to detect errors in small programs in Section 3.
- Applying dynamic binary instrumentation and machine learning to detect errors in the ARDUPILOT robotics program in Section 4.
- Using novelty and outlier detection methods to detect errors in additional robotics programs in Section 5.
- Evaluating several points in the design space of possible techniques to reduce intrusiveness of instrumentation tools to be used with the techniques established here in Section 6.

2 Review of Literature and Background

2.1 Related Work

The following sections give an overview of related work and background concepts that inform the proposed work.

Dynamic Binary Analysis There are several reasons for developing analysis techniques that do not depend on access to source code or debugging information. Techniques that do not require access to source code are more widely applicable. They can be used with proprietary implementations for which source code is not available, implementations written in different languages, and machine code for different architectures [10]. Robotics and autonomous systems often incorporate components from multiple suppliers, and source code is not always available for the components [27]. When source code or debugging information is not available, many analysis and debugging techniques, such as those that use abstract syntax trees, are not easily available [68].

Several other dynamic techniques do not require source code. For example, Clearview extends the invariant inference and violation work described below to Windows x86 binaries, without the need for source code or debugging information [63].

Eisenberg et al. [22] introduce using dynamic analysis to trace program functionality to its location in binary or source code. However, as with many dynamic analysis tools, the implementation is limited to Java.

Observation-based testing is a name given to testing techniques that involve "taking an existing set of program inputs (possibly quite large); executing an instrumented version of the software under test on those inputs to produce execution profiles characterizing the executions; analyzing the resulting profiles, with automated help; and finally selecting and evaluating a subset of the original executions [20]." This approach makes use of dynamic analysis through instrumentation [49].

There exist general-purpose frameworks for writing and using instrumentation tools for dynamic binary analysis. These tools are discussed in Section 2.2.1.

Anomaly, Novelty, and Outlier Detection Anomaly, novelty, and outlier detection refer to a collection of techniques that identify data points that are in some way unusual or appear to deviate markedly from or be inconsistent with the patterns created by the rest of the data [8, 37, 65].

Approaches to anomaly detection can be separated into three basic groups. In one group, anomalies are detected with no prior knowledge of the data. The second group of techniques

models both the nominal data and the abnormal data, in an approach similar to supervised classification. The third group of techniques creates a model from normal (nominal) data and identifies whether new data fits within that model [37].

This third group of approaches is common and is analogous to a one-class classification problem in the context of machine learning. In a one-class classification problem, one set of data is treated as the *normal* or *positive* data. The task is to distinguish the normal data from all other data, i.e., the data that does not fit into the class (which are the anomalies). One-class algorithms usually assume that the normal data is better represented in the data than any abnormal data [65].

Some work builds models of unusual software executions by inferring invariants and identifying executions that violate these invariants. While authors sometimes use the term anomaly detection for that approach, I will address that work separately below [26].

Clustering Algorithms Many anomaly, novelty, and outlier detection approaches, as described above, make use of clustering algorithms, which arrange similar data points into groups [4].

Clustering approaches include (1) an algorithm for organizing data into clusters and (2) a metric for determining whether new data fits into the existing clusters or is an anomaly.

Work has applied clustering to identification of software failures [21, 35, 54, 57]. Haran et al. set out a general framework for approaching using execution data as a basis for classifying whether program outcomes are successful [34, 35], including the use of clustering. Dickinson et al. focus on using clustering for filtering executions worthy of further study [20, 21]. Mao and Lu [57] propose using more complex clustering based in Markov models to identify which failure executions typify failure modes. Note that in this approach, clustering is used to find executions that are typical, not those that are anomalies or outliers.

Intrusion Detection Work on intrusion detection, especially host-based intrusion detection, which looks at techniques for determining whether an adversarial attacker has gained access to the host system [19, 76] is a subset of dynamic analysis and is strongly related to anomaly detection. At a high level, the intrusion detection models monitor the events when applications interact with the operating system, particularly in system calls. These intrusion detection techniques can make use of anomaly detection. Initial IDSs were based in the idea of creating a database of usual patterns in system call traces and identifying any other patterns as anomalies. Advances on these approaches have included formalizing the system models, reducing overhead, and incorporating timing as a factor in patterns [54].

Intrusion detection systems have evolved towards modeling normal system behavior, but attackers have gotten good at mimicry attacks, in which they mask their attacks as events within the bounds of normal system behavior [76]. This adversarial behavior can lead to something of a stalemate. To this end, companies have begun applying machine learning and artificial intelligence techniques in their threat-detection approaches [39, 40].

Statistical Fault Identification [51, 52, 78, 79]

Liblit et al. propose a collection of techniques that identify program faults using statistical techniques [51, 52, 78, 79]. The techniques require source code and look at isolating deterministic and nondeterministic bugs by using statistical techniques to correlate faults with program

predicates – true/false assessments about variable values that are assessed at many instrumentation points throughout the program. These techniques include approaches for leveraging a large number of user executions of programs, distributing the overhead burden of instrumentation among users [51]. Subsequent refinements to the techniques included uniting approaches for deterministic and nondeterministic bugs [78]; increasing the usefulness of the program predicates identified [79]; and separately identifying the effects of different bugs [52].

Dynamic Invariant Detection A body of work focuses on automatically detecting invariants – properties that hold true over all correct executions of a program – and identifying bugs by identifying executions for which those invariants are violated. Automated invariant detection relies on automatically generating inferences about a program's semantics [24].

One of the pioneers in this area is Daikon [25, 26]. It detects invariants across a range of languages and types of invariants. However, it is limited in application to programs with source code, it is limited by the program points at which it can make inferences, and it has limited scalability.

Other approaches such as DIDUCE [32, 33] are limited to particular languages, require source code, fail to scale, or have other limitations which reduces their usefulness in complex autonomous systems.

One invariant detection approach that does not require source code is Clearview [63]. However, Clearview is primarily a technique for repairing errors once they are detected, and it is limited to the particular types of attacks it is designed to work with. To detect any other type of error or attack, a separate technique would have to be used. In this way, Clearview could be used in conjunction with techniques such as those proposed in this work.

The Oracle Problem The proposed work builds on earlier work in software testing, which addresses the problem of trying to figure out whether a program is behaving as intended, which is known as the *oracle problem*. Several works provide a useful summary of oracle problem research [7, 64].

The oracle problem is a barrier to automated software testing because, even when a test encounters a software defect, an imperfect oracle may not detect that a defect was encountered. The oracle problem has been studied in a variety of contexts [16, 29, 60, 61, 64, 69, 71, 77]. The proposed work can be thought of as using dynamic binary instrumentation in combination with machine learning to provide a pseudo-oracle. This pseudo-oracle provides a basis to believe whether a program behaves as intended or exhibits a defect. It can also be used in conjunction with other oracle-estimation techniques. As such, it enhances the usefulness of automated testing techniques by providing another tool for understanding whether any automatically- generated tests reveal defects.

Kanewala and Bieman [42] survey existing program testing techniques that attempt to substitute for an oracle. They highlight several approaches in the domain of computer graphics that make use of machine learning [15, 30]. While one might expect these approaches to be similar to those proposed in this work, this category of techniques focuses on using machine learning to validate the *output* of a program, rather than ensuring its correct operation at all times. The distinction is key when applied to situations where properties such as safety must be maintained

at all times.

Verification and Formal Methods Much work has gone into formal verification of cyber-physical systems in order to avoid software faults. However, as Zheng et al. point out in their survey of literature and interviews with practitioners on verification and validation for cyber-physical systems, there are many gaps between the verification work and practical application to entire real systems. Some of the challenges identified are that developers do not understand software verification and validation; existing formal techniques do not meet developers' needs, such as needing to model physics in addition to computation; and developers revert to an approach based in trial-and-error because of the inapplicability of formal tools [80, 81].

Additional challenges include that formal models of systems often include assumptions that do not necessarily hold true in the real world. Any proofs done by humans or by human-designed proof assistants are susceptible to human error. Continuous elements of the systems may not be adequately modeled [67].

One of the most significant drawbacks of formal verification is the time and effort required to achieve anything nearing a complete formal model of a real system. An example of the extraordinary costs is the verification of the kernel of a secure CPS application which was proved, using a state of the art theorem prover but required 20-person-years' worth of effort [43, 44, 80].

Testing Autonomous Vehicles and Robotics Testing autonomous vehicles and robotics systems presents problems unique to those related domains. Beschastnikh et al. outline some of the challenges and drawbacks to existing approaches to debugging distributed systems, such as robotics and autonomous vehicles [9]. Several approaches have addressed aspects of the problems in testing these systems. Sotiropoulos et al. motivate testing robotics in simulation and demonstrate the approach's effectiveness in some domains [70]. Tuncali et al. define a *robustness function*for determining how far a system is from violating its parameters [75]. Notably, this approach relies on well-defined system requirements, which are absent in many systems. Timperley et al. attempts to categorize real bugs reported in the ArduPilot autonomous vehicle software as to whether they can be reproduced and/or detected in simulation [74]. Various companies attempting to build self-driving cars or their constituent algorithms have created extensive simulation environments, in an attempt to capture all relevant real-world scenarios [56].

Fraade-Blanar et al. establish that there is no uniform definition of *safety* as it applies to autonomous vehicles and, therefore, there is no established way to measure that such vehicles are behaving safely. The authors propose ways to more systematically evaluate whether such systems are behaving safely and, by having such a measure against which to compare improve the safety of those vehicles [28].

Koopman and Wagner highlight the significant challenges involved in creating an end-to-end process that ensures that autonomous vehicles are designed and deployed in such a manner as to take account of all of the myriad concerns that contribute to the vehicles' ultimate safety [45]. They also advocate consciously disentangling the testing of different components of the systems and testing for different goals to ensure that testing results are well-understood and can be used effectively [46]. Hutchison et al. outline a framework for robustness testing of robotics and autonomous systems, highlighting the differences from traditional software [38]. Forrest

and Weimer further highlight challenges in detecting and repairing faults in certain classes of autonomous systems, such as the potential inaccessibility of the system, limited computing and power resources, and use of off-the-shelf components [27].

Theissler presents work using anomaly detection on data gathered in an automotive context [73]. While Theissler's work focuses on detecting faults injected in analog vehicle signals, it looks at anomaly detection on portions of a distributed system with many unpredictable environmental factors. The approaches can inform simulation testing of autonomous vehicles.

Fault Classes and Categorization While, in general, I have used terms such as, "bug," "fault," "error," "failure," and, "off-nominal," to refer to any software behavior that is unusual or unintended, other work has broken down the nature of unintended software behavior into a more precise taxonomy and dealt with the classification of these types of behaviors [5, 6, 17, 23, 31, 36, 50, 53, 66, 72]. Notably, Avizienis et al. set out distinctions among a *service failure*, which occurs when a service deviates from its functional specification, either because the service fails to comply or because the functional specification is inadequate; an *error*, an observed state of the system that differs from the correct state; a *fault*, which is the actual or posited cause of an error; and a *vulnerability*, which is an internal fault that enables harm to the system [6]. They further establish eight fault dimensions based on features such as objective and persistence.

2.2 Background

In this section, I will introduce several tools and concepts that I use extensively and to which I refer elsewhere in this document.

2.2.1 Dynamic Binary Instrumentation

Dynamic instrumentation works by analyzing a subject program while it executes. The tool performing the analysis inserts code that analyzes the subject program, to be run while the subject program runs. The act of inserting the code is known as instrumentation. Dynamic binary instrumentation operates on the level of object code (pre-linking) or executable code (post-linking). Dynamic binary instrumentation does its work at runtime, allowing it to encompass any code called by the subject program, whether it be within the original program, in a library, or elsewhere [55, 58, 59].

PIN My initial experiments use PIN version 2.13, to instrument the subject programs and collect the data for model construction. PIN is a dynamic binary instrumentation tool [55] distributed by Intel and available online.[1] PIN constructs a virtual machine in which the program under examination is run. PIN allows automated data collection at the machine instruction level, and can observe a program's low-level behavior while being transparent to the instrumented program.

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

PIN has a robust API that enables the creation of tools that measure many things about program execution.[2] However, instrumentation can be heavyweight and add significant overhead.

VALGRIND As a dynamic binary instrumentation framework, VALGRIND[3] allows tools based on the platform to record data about low-level events that take place during a program's execution. While tools based on VALGRIND can have significant overhead, there are optimizations that enable reduced overhead.

PIN and VALGRIND both have the capabilities to measure many of the low-level of signals of interest. However, their APIs and operations mean that different signals are easier to measure on each framework and that different operations take different amounts of time and add different amounts of overhead.

2.2.2 Machine Learning Models

These experiments often use the publicly available algorithms included in the Scikit-Learn[4] package for many functions related to machine learning, including data processing, model generation, and evaluation [62]. In some cases, the experiments use custom algorithms or customized versions of existing algorithms.

Supervised Models

Supervised learning takes a set of training examples with given labels (e.g., "true" or "false"), and produces a predictive model. The model then generates predictions of labels for new data. Support vector machines, stochastic gradient descent, decision trees, perceptrons, artificial neural networks, and statistical learning algorithms are examples of supervised machine learning algorithms [47]. I use support vector machines and decision trees especially in this work.

A support vector machine maps training data as points in multidimensional space by kernel functions. It then attempts to construct a hyperplane that divides the data points between labels, such that the gap between the two sets of points is as wide as possible. The predictive model uses this hyperplane to assign predicted labels. Each new point is mapped into the multidimensional space; the predicted label is dictated by which side of the hyperplane it is mapped to [18].

For the supervised learning experiments that use a decision tree classifier, I use the one available out-of-the-box from Scikit Learn. This decision tree classifier provides results similar to or better than the results using other classifiers available from Scikit Learn across a wide range of different programs and collected data. In addition, the decision tree algorithm provides the ability to generate explanations of the decision processes in a human-understandable form, to a much greater extent than many other algorithms [1].

I assess supervised learning models using the accuracy metrics explained later in this section.

[2] https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/group__API__REF.html

[3] http://valgrind.org/

[4] http://scikit-learn.org/

Unsupervised and Clustering-based Models

Unsupervised and clustering-based algorithms take a somewhat different approach to classification than the supervised algorithms do.

Some of the clustering-based models can be used in the same way as supervised modes – trained on a set of training data with known labels and tested on unknown data – while other uses involve simultaneously creating a model and making predictions on the data points used to create the model.

Several existing algorithms include: One-class SVM [5], LOF [6], and LDCOF [4].

For some of the early work, I make use of a customized algorithm for novelty detection that makes use of domain knowledge. This customized algorithm is explained in Section 3.2.2.

Similarly, for the work on varied robotics programs, we use a more sophisticated distance metric in conjuction with a one-class clustering algorithm. This metric is inspired by the LDCOF algorithm [4]. This algorithm is described in Section 5.1.

Data Pre-Processing

In machine learning contexts, it is common to pre-process data into forms that are more amenable as input to the algorithms. Some common forms of data pre-processing are:

- **Scaling data:** Transforming the data for each feature such that each feature is in the same range (e.g., ranging from -1 to 1). This transformation can prevent features whose values are large from overwhelming features whose values are small, which is a pitfall with some algorithms.
- **Balancing data:** Transforming training data such that there are equal numbers of data points of each class (e.g., equal numbers of data points representing passing data and failing data). This transformation is useful if we do not want a supervised algorithm to create a model that predicts the class with more training data points more frequently than the class with fewer training data points. The transformation is accomplished either by duplicating data points from the class or classes with fewer data points or removing data points from the class or classes with more data points.
- **Signal reduction:** Evaluating and choosing which signals are most predictive and restricting algorithms to running on those. This approach can help to reduce overfitting.
- **Dimensionality Reduction:** Projecting feature vectors (data points) into a lower dimensional space. A feature vector can frequently consist of many features, making it difficult for humans to visualize. Dimensionality reduction creates a representation of each data point within the data set that can be visualized and analyzed at lower dimensions.

[5] http://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html

[6] http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html

Assessing Machine Learning Models

I use the following measurements to assess the accuracy of machine learning models. These measurements apply when we have models that make predictions that I can compare against known ground truth.

- **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.

3 Dynamic Binary Analysis to Detect Errors in Small Programs

This section describes proposed work that I have completed. To validate initial insights that low-level information about the behavior of an executing program gives a picture of the characteristics of that execution, I collected low-level information about many executions of several individual small programs. I created models of program behavior from the data collected from the executions of each program. I used those models to predict whether each new execution fit into the range of expected program behaviors or whether it represented an unexpected or unintended behavior. While not all rare behaviors are unintended, unintended behaviors are more likely to be rare [24]. To build the models, I made use of both supervised machine learning techniques and unsupervised anomaly detection techniques.

3.1 Motivating Example

The following example casts light on the insight behind my proposed techniques. Although these techniques operate at the level of binary machine code, I present a small source code example, which is easier for humans to follow, (Figure 3.1[1]) to illustrate the underlying insight. On December 31, 2008, all Microsoft Zune players of a particular model froze [2]. The cause was a software error that resulted in an infinite loop in the date calculation function on the last day of a leap year.

Consider the (correct) program behavior when called with a value that does *not* correspond to the last day of a leap year, such as `days = 1828` (January 1, 1985[2]). With this input, `(days > 365)` is `true`, entering the `while` loop. The condition in `if (isLeapYear(year))` is `false`, causing the program to enter the `else` clause. `days` is adjusted to `1463`; `year` is incremented. This loop and condition check repeats three more times, at which point `year=1984` and `days =368`. At this point, `isLeapYear(year)` and `days > 366` both evaluate to `true`, and `days` becomes `2` while `year` becomes `1985`. The `while` condition is now `false`, and the program correctly prints the year, `1985`.

Consider instead what happens if `days = 366`, corresponding to the last day of 1980, a leap year. Again the program enters the `while` loop, and `if (isLeapYear(year))` is `true`. However, `days > 366` is `false`, and `days` remains unchanged (indefinitely) on return to the

[1]Originally downloaded from `http://pastie.org/349916`

[2]Date computation begins with the first day of 1980.

```
1  void zunebug(int days) {
2    int year = 1980;
3    while (days > 365) {
4      if (isLeapYear(year)){
5        if (days > 366) {
6          days -= 366;
7          year += 1;
8        }
9      } else {
10       days -= 365;
11       year += 1;
12     }
13   }
14   printf("current year is %d\n", year);
15 }
```

Figure 3.1: Code listing for the Microsoft Zune date bug.

`while` condition.

This example is simple in the sense that the defective behavior can be described by high-level desired program properties (such as "the function should return within 5 minutes."), and infinite loops are specifically targeted by other techniques [12, 14]. However, it still illustrates the insight that runtime program characteristics can detect aberrant program behavior, because there are numerous, intermediate behaviors that provide clues distinguishing between correct and incorrect behavior. For example most simply, in the aberrant case, the number of source code instructions executed is abnormally high. Similarly, the program never reaches portions of the source code corresponding to normal exit behavior (measured by program counter).

While no single runtime signal perfectly captures incorrect behavior, we hypothesize that combining signals can be used to characterize and distinguish between correct and incorrect behavior for programs.

3.2 Approach

This section outlines our approach for predicting whether a program passes or fails on new test inputs. In broad summary, we collected dynamic execution signals from programs executing on test inputs. Each program on each test case contributes a set of signals that comprise a single data point. We used off-the-shelf machine learning classifiers and labeled training data (in the supervised case) to construct models that predict whether observed program behavior on new inputs corresponds to correct or incorrect behavior.

3.2.1 Dynamic execution signals

My goal is to construct models that automatically distinguish correct from incorrect program behavior. I use machine learning to construct these predictive models. Recall that machine learning receives data in the form of *feature sets* that describe individual data points. In this work, I compose feature vectors of dynamic binary runtime signals.

I collected all runtime data using PIN, a dynamic binary instrumentation tool discussed in Section 2.2.1. PIN provides an interface that allows for the straightforward collection of all signals described below.

I measure and include the following dynamic signals in my model:

- **Machine Instructions Executed.** This is simply a count of all the instructions executed in the program. A program exhibiting unintended behavior may have an abnormally high or low number of instructions executed. For example, in the infinite loop example in Section 3.1, the unintended behavior corresponded to an abnormally high number of instructions executed.
- **Maximum Program Counter Value.** The program counter, also known as the instruction pointer, tracks which machine instructions a program executes within that program's machine code. The maximum value of the program counter therefore indicates how far into its code a program has progressed, which may indicate that the program is executing unexpected code (or failing to execute expected code) on some input. We limited the collection of program counter values to those that occur within the program code itself, excluding program counter values corresponding to libraries and other external code.
- **Minimum Program Counter Value.** The minimum value of the program counter indicates the earliest location within the program code that the program executes. It therefore may capture unexpected behavior similar to that captured by the maximum program counter value.
- **Number of Memory Reads** This signal corresponds to the number of times a program transfers data from memory to a register; we hypothesize that memory read or write behavior may follow patterns that vary between normal and abnormal executions. PIN instruments all instructions that have memory reads to collect this signal, and if a single instruction has more than one such read, that instruction is instrumented a corresponding number of times.
- **Number of Memory Writes** This signal corresponds to the number of times a program transfers data from a register to memory; we collect it for the same reason and in much the same way as memory reads.
- **Minimum Stack Pointer Value** The stack pointer indicates the boundary between available memory and program data stored in the program's stack. In x86 machines, the stack grows downwards, such that the higher the value of the stack pointer, the less memory used, and vice versa. The stack pointer is also a rough proxy for the depth of recursion, as the stack pointer is usually adjusted downwards each time a routine is called [11]. The minimum stack pointer value can provide a rough idea of the maximum depth of recursion exhibited in the program.

- **Maximum Stack Pointer Value** The maximum value of the stack pointer usually corresponds to the value of the stack pointer at the program's initiation. However, a deviation from this behavior could indicate wildly incorrect behavior.
- **Call Taken Count** At every call instruction the program encounters, the call may be taken or not taken. Whether calls are taken or not taken is a rough proxy for the shape of the program's execution.
- **Call Not Taken Count** Similarly to the call taken count, this signal measures when call instructions are not taken. The along with the call taken signal, call not taken signal is a proxy for the shape of a program's call graph.
- **Branch Taken Count** When a program reaches a branch instruction the program may jump to non-consecutive code, in which case we say that the branch is taken. The number of branches taken serves as a proxy for the dynamic program control flow. The number of branches taken captures behavior within a routine, while the number of calls taken captures the behavior across routines.
- **Branch Not Taken Count** Similar to the branch taken count, the branch not taken count measures when a branch instruction is reached, but the code falls through to consecutive code.
- **Count of Routines Entered** This signal measures the number of routines within the program actually executes. This can be a proxy for control flow behavior across functions within the program, whereas call taken count includes calls to library functions.

I chose these signals for both their predictive power and the ability to collect them without ruinous overhead. I have tried and rejected some other signals, such as keeping histograms of various values, because of their higher overhead of calculation and memory usage at runtime. I use these signals to construct predictive models, as described next.

3.2.2 Model generation

This section describes my approach for generating a predictive correctness model from the features described in the previous subsection (Section 3.2.1).

I built both supervised and unsupervised models to investigate the trade-offs inherent in the two types of techniques. As discussed in 2.2.2, supervised learning requires labeled training data. As a result, our model construction technique is applicable to situations in which a program already has a set of test inputs and known outputs (a longstanding set of regression tests, for example) that might be augmented by a test generation technique. On the other hand, our approach to unsupervised learning requires domain-specific assumptions. In our case, we used unsupervised outlier detection to group data points into two sets: typical behavior and outliers. However, to give meaning to the groupings, we assumed that the typical behavior corresponded to the program's intended behavior. This assumption was rooted in experimental observation, but the requirement of sufficient knowledge of the data to make this kind of assumption is a limitation of unsupervised learning.

Given a set of benchmark programs and test cases with known outputs, we gathered the signals described in Section 3.2.1 for each test execution into a single data point. For the training of

the supervised machine learning model, we associated each data point with a label corresponding to whether the test case passed or failed. We also investigated a labeling in which execution data points were labeled with "working" or "broken", depending on whether the associated program failed *any* of its test cases (this label is an approximation, since testing is known to be unsound). We used two sets of labels because we wanted to study how to best characterize the intended and unintended program behavior.

We also used this labeled data to evaluate the accuracy of the unsupervised learning approach (though not as training data, by definition). Our outlier detection method starts by separating the data for each program by test case. That is, for every program, we considered the data for each test case across all known versions of the program. Separating the data in this way gave us insight into the differences among versions for a single test case. If we predict that a given version fails a given test case, we can predict that the version is broken. Aggregating the predicted broken versions across all test cases gives us an overall list of predicted broken versions.

Within the single test case data, we used a simple outlier detection technique on each feature. It identifies those data points that are more than two standard deviations from the mean of that feature. If, for a single test case, any given version is identified as a potential outlier for more than one feature, we identified that version as potentially failing for that test case. We then aggregated a list of all versions that are potentially failing for any test cases and predicted a label of "broken" for all of the versions in this list. This unsupervised outlier detection, or clustering, technique differs from other outlier detection technique in that it separates the data by test case and treats each feature individually.

The approach to outlier detection is predicated on the assumption that any given test case will pass on more versions than those on which it fails. Although this assumption can be limiting, this assumption has validity in certain domains. For example, our observations in the search-based automated program repair domain indicate that individual test cases do pass on more program variants than those on which they fail [48]. This approach could therefore also apply in a setting where multiple mutants of the program under consideration could be created automatically for the purposes of creating the outlier model, in the absence of the labeled data required for the supervised learning technique.

3.3 Experimental Design

For each small program, I had 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 also included executions of multiple versions of each program under test. I used the collected signals to build models of behavior.

For the purposes of these experiments, I used PIN[3], a dynamic binary instrumentation toolkit distributed by Intel, to create tools to collect the low-level signals on running programs. For each execution, I collected 165 low-level signals, later using feature extraction to choose 15 that were the most broadly relevant.

For supervised machine learning, I gave a portion of the data points, along with labels in-

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

dicating if each data point represented a correct or incorrect execution, to a supervised learning algorithm, which built a predictive model.[4] 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. Our unsupervised learning technique is built on outlier detection [37]. We start by separating the data for each program by test case. That is, for every program, we consider the data points for each test case across all available versions of the program. Separating the data in this way gives us insight into the differences among versions for a single test case.

Within the data relating to a single test case, we use a simple outlier detection technique on each feature. It identifies data points more than two standard deviations from the mean of each feature. If, for a single test case, any given version is identified as a potential outlier for more than one feature, we identify that version as failing for that test case. This unsupervised outlier detection, or clustering, technique differs from other outlier detection techniques in that it separates the data by test case and treats each feature individually.

This approach to outlier detection requires several versions of a program and is predicated on the assumption that any given test case will pass on more versions than those on which it fails. However, techniques such as mutation testing can generate multiple versions of a program [41], and although the assumption can be limiting, it has validity in certain domains. For example, the search-based automated program repair domain indicates that individual test cases do pass on more program variants than those on which they fail [48]; analyses in fault localization support a similar conclusion [13]. This approach could therefore also apply in a setting where multiple mutants of the program under consideration could be created automatically for the purposes of creating the outlier model, in the absence of the labeled data required for the supervised learning technique.

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.[5] Table 3.1 lists the lines of code, numbers of test cases, and number of variants for each program.

3.4 Results

3.4.1 Supervised Learning

Table 3.2 shows results for supervised learning on the data collected from executions of these programs, using standard machine learning assessment metrics.

[4]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.

[5]http://sir.unl.edu/portal/index.php

Table 3.1: 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 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

Table 3.2: 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

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 3.2 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 3.2 to the center, the decision tree classifier outperforms the support vector machine classifier for most programs.

3.4.2 Unsupervised Outlier Detection

We conduct unsupervised learning experiments to discover whether we can determine whether a program behaves correctly without using any ground truth labels to train a classifier, such as in a situation in which a developer is creating new test suites from scratch and lacks existing cases to

Program	True Pos	True Neg	False Pos	False Neg	*Acc*	*Prec*	*Rec*	*FM*
printtokens	28620	405	79	3936	0.88	1.00	0.88	0.93
printtokens2	39516	1275	962	3512	0.90	0.98	0.92	0.95
replace	165394	2313	965	14214	0.92	0.99	0.92	0.96
schedule	22925	406	383	2786	0.88	0.98	0.89	0.94
schedule2	24963	96	199	4552	0.84	0.99	0.85	0.91
tcas	62854	1155	424	3103	0.95	0.99	0.95	0.97
totinfo	21905	1190	765	1388	0.91	0.97	0.94	0.95

Table 3.3: Outlier Detection. Left: Number of executions correctly predicted as passing, correctly predicted as failing, incorrectly predicted as passing, and incorrectly predicted as failing. Right: accuracy metrics for the same data.

inform a supervised technique.

Table 3.3 reports the performance of the outlier detection method in predicting passing and failing test case behavior in terms of both raw counts (for context, because there are significantly more passing data points than failing data points) and accuracy metrics. We do not balance these datasets, because the discrepancy between numbers of passing and failing test cases is inherent to the assumption underlying this technique (and is consistent with observations of testing behavior in the field [13] and the context of potential clients of our technique, such as program repair [48]).

These results show high accuracy across all metrics for the SIR Siemens artifacts. The generally high accuracy of these results supports our hypothesis that correct and incorrect behavior can be identified at the level of binary execution signals and linked to test case passing/failing behavior.

3.5 Limitations Suggestive of Future Directions

This work shows that the techniques I propose are broadly able to detect various program behaviors that deviate from expected or usual behavior. However, this work suggests certain limitations in the techniques as applied.

- One of the limitations is the limited set of programs on which I tested the techniques.
- One limitation is that the supervised learning techniques limit application to situations in which we have labeled data: data for which we know whether the executions were typical or anomalous.
- One of the limitations is that the instrumentation used in my techniques can perturb program execution or incur unacceptable overhead.
- An additional limitation is that larger trends in program behavior can obscure smaller fluctuations.

4 Dynamic Binary Instrumentation to Detect Errors in Robotics Programs – ARDUPILOT

This section describes proposed work that I have completed.

4.1 The ARDUPILOT System

I conducted this set of experiments on the ARDUPILOT system.[1] This system is an open-source project, written in C++, with autopilot systems that can be used with various types of autonomous vehicles. It is very popular with hobbyists, professionals, educators, and researchers. It runs a control loop architecture. It has approximately 580,000 lines of code and has over 30,000 commits in its GitHub repository.[2]

ARDUPILOT provides a rich ground on which to test robotics systems. It is sufficiently complex to be useful in the real-world. There is a wealth of information about bugs encountered in real world usage, both in the version-control history and in the academic literature [74].

Our system uses ARDUPILOT in simulation, with the included software-in-the-loop simulator. We use a customized test harness that enables coordinated control over simulations, including simulation of attempts to hack the vehicle remotely.

4.2 ArduPilot Approach

For each set of tests, we use pairs of versions of the ARDUPILOT software. Each of these pairs anchors a *scenario*. The pair consists of a version of the ARDUPILOT software that contains a known defect along with a corresponding version in which that defect is repaired. Each scenario also contains inputs and environmental constraints under which the defect will be activated:

- SCENARIO A contains a buffer overflow seeded in APMROVER2/COMMANDS_LOGIC.CPP. The invalid buffer is prepared when one command identifier is used in ROVER::START_COMMAND but does not cause a crash until the same method is called with a different command identifier. We execute SCENARIO A a total of 4000 times with dynamic binary execution, 1000 times for each of the categories enumerated in Section 4.2.

[1] `http://ardupilot.org`
[2] `https://github.com/ArduPilot/ardupilot`

- SCENARIO B contains an infinite loop, seeded in the `MAV_CMD_DO_CHANGE_SPEED` command case of APMROVER2/COMMANDS_LOGIC.CPP. The seeded defect calls a function that loops. Normal execution of the software requires correct behavior in this function, i.e., without the seeded defect. We execute SCENARIO B (and each following scenario) a total of 2000 times with dynamic binary execution, 500 times for each of the categories enumerated in Section 4.2.
- SCENARIO C seeds a malicious arc-injection attack, injecting a jump to code that ARDUPILOT should not execute at that time.
- SCENARIO D seeds a double-free situation, in which allocated memory is freed more than once.
- SCENARIO E seeds an attempt to access a global variable that should be protected.

There are several refinements to the general approach for identifying anomalous behavior using models built over low-level dynamic signals for ARDUPILOT. Our approach takes as input multiple pairs of versions of the ARDUPILOT system. Each pair of versions corresponds to a known bug fix, that is, (a) a version of the software that contains a known defect and (b) a version in which that defect is repaired. Additionally, we assume the provision of inputs or test cases for each version, with at least one that activates the defect, and at least one that does not. Given these inputs, we execute each program version on each input several times under dynamic binary instrumentation. This results in data on:

1. executions of a version that contains a particular defect that exercise that defect;
2. executions of a version that contains a particular defect that *do not* exercise the defect;
3. executions of a repaired version corresponding to the defect-containing version, running with inputs that would have exercised the defect; and
4. executions of a repaired version corresponding to the defect-containing version, running with inputs that would *not* have exercised the defect.

I use the data collected for a variety of experiments that use machine learning models to identify whether a particular execution exhibits a defect. These experiments approximate various real-world situations.

4.3 Experimental Setup

In this section we provide details relating to how we implement the process described in Section 4.2. We begin by outlining the questions we seek to answer. In Section 4.3.1, we go on to give details on supervised machine learning and how we use it. We have already presented our main program under test and the experimental scenarios we use on it in Section 4.2. We present our experimental setup to evaluate the following research questions:

- *RQ1:* Given multiple executions of a version of ARDUPILOT with a defect in it, some of which activate the defect and others of which do not, how well does supervised learning identify the executions that activate the defect?
- *RQ2:* Given multiple executions of a version of ARDUPILOT with a defect in it and a version of ARDUPILOT in which the defect was repaired, running with an input that activates

the defect, how well does supervised learning detect executions of the version that contains the defect?

- *RQ3:* For each of the above, how does the number of executions affect the accuracy of the predictions?
- *RQ4:* For each of the above, how much overhead is required to collect the data to make the predictions?

We run the experiments pertaining to ARDUPILOT SCENARIO A on a 64-bit machine running Ubuntu 14.04.5, with 8 virtual (4 actual) cores, and 8 GB of RAM. Each experiment is run within a Docker container running the same operating system.

We run the experiments pertaining to ARDUPILOT SCENARIO B, SCENARIO C, SCENARIO D, and SCENARIO E in an Ubuntu 16.04 virtual machine with 768 MB of RAM and one virtual core, hosted on the same machine used to run SCENARIO A.

4.3.1 Supervised Machine Learning

We use out-of-the box machine learning algorithms from Scikit Learn.[3] Specifically, for supervised learning, we use the Decision Tree classifier available from Scikit Learn. Before selecting this classifier, we did an informal survey, analyzing our data with several of the options available from Scikit Learn. We found that Decision Tree provides results similar to or better than the results using other classifiers across a wide range of different programs and collected data. In addition, the Decision Tree algorithm provides the ability to generate explanations of the decision processes in a human-understandable form, to a much greater extent than many other algorithms. After an informal parameter sweep, we decided to use the default parameters of the algorithm.

For all supervised machine learning experiments, we use K-fold cross-validation, with K=10 for all sample sizes greater than or equal to 100 and smaller values of K for smaller sample sizes, to ensure at least 10 points in each test sample. We report the arithmetic mean across all folds.

We balance all data by duplication of pseudo-randomly chosen data points in the minority class. We choose to balance by duplication because it does not reduce the size of the data set; our preliminary experiments show comparable results with balancing by deletion, when the data set is large enough. We also use a fixed random seed for reproducibility.

4.3.2 Collecting Signals with Dynamic Binary Instrumentation

We use dynamic binary instrumentation to collect low-level information about the processes that execute when we run software. We count events that occur during execution and keep track of minima and maxima for certain values. For example, we count the number of machine instructions executed, the number of memory stores, and the minimum and maximum memory addresses associated with each. These counts result in a summary of each execution, consisting of a list of numbers, corresponding to each measurement. We collect the same measurements for each execution, so the summary of each execution can be treated as a feature vector, suitable for

[3] `http://scikit-learn.org/`

input into machine learning algorithms. We aggregate the feature vectors into two-dimensional matrices, representing the overall data set.

We use a customized tool based on the VALGRIND framework, version 3.14, to record low-level information about program executions.

For these experiments, we collected a set of data for each execution, making use of information that is easy to track with low overhead within VALGRIND'S framework. These data include, but are not limited to, values for:

- Minimum and maximum instruction address
- Number of instructions executed
- Minimum and maximum address of memory loads
- Number of memory loads

One key difficulty in using a dynamic binary instrumentation approach with a timing-sensitive system, such as ARDUPILOT or virtually any other robotics software, is that the overhead of collecting the information changes the timing in the program execution. These timing changes can change the program's control flow by, for example, causing various timeouts to trigger or causing events to happen in an order that the program does not expect. We took several approaches to reducing the effects that instrumentation has on timing sensitive executions. Wherever possible, we relaxed timeout parameters in the software. In addition, we worked to reduce the overhead of our instrumentation.

VALGRIND is a powerful tool, able to measure many events at the low level. However, we restrict the events we measure to those that can be measured with a minimum of added overhead. The selection is largely inspired by the structure of VALGRIND'S instrumentation, collecting that information that comes nearly, "for free," when instrumenting a given instruction and collecting those pieces of information efficiently. For example, although we could collect data about whether branches are taken and whether they correspond to what VALGRIND considers inverted conditions, we do not collect those data because collecting them would involve more calculations and calls into VALGRIND'S API at runtime.

4.4 Results

4.4.1 *RQ1:* Supervised Learning on a Single Defective Version of ARDUPILOT

Recall that our overall goal is to determine whether an approach that combines dynamic-binary-instrumentation with machine-learning analysis is useful in detecting behavior that exhibits defects in software. This question evaluates the subgoal of determining whether we can use supervised machine learning to identify program executions that exhibit a defect, as opposed to executions of the same defect-containing program version. To evaluate this question, we return to our main program under test: ARDUPILOT.

Results can be found in Table 4.1. Note that our technique can almost always tell the difference between executions with the input that activates the defect and executions that do not. All accuracy metrics are very high, showing a lack of bias towards false positives or false negatives.

Table 4.1: Accuracy Metrics for Supervised Learning on a Single Defective Program Version with an Input that Activates the Defect and One That Does Not.

Scenario	Mean Acc.	Mean Prec.	Mean Rec.	Mean F-Score
Scenario A	0.9835	0.9793	0.9883	0.9836
Scenario B	0.9991	0.9982	1.0000	0.9991
Scenario C	0.9990	1.0000	0.9980	0.9990
Scenario D	0.9963	0.9964	0.9963	0.9963
Scenario E	0.9963	0.9983	0.9944	0.9963

The lowest score, a precision of 0.9793 in SCENARIO A, should be acceptable to users in nearly all circumstances. These very accurate results suggest that the technique may be amenable to handling more difficult and complex experiments.

4.4.2 *RQ2:* Supervised Learning on a Defective Version of ARDUPILOT and Its Repaired Counterpart

Recall that our overall goal is to determine whether an approach that combines dynamic-binary-instrumentation with machine-learning analysis is useful in detecting behavior that exhibits defects in software. This question evaluates the subgoal of determining whether we can use supervised machine learning to identify program executions of a defective program version as opposed to executions of its repaired counterpart. We execute the defective program version and the repaired counterpart 500 times each with the same input, which is an input that activates the defect. Table 4.2 shows results. Note that supervised learning can nearly always determine whether an execution with the defect-activating input is on the defective version or the repaired version. Although these results come from a constrained experimental setup, their high accuracy metrics suggest that the approach might extend to setups with more noise.

For control, we run supervised learning on SCENARIO A with an input that does *not* activate the defect. In these experiments, the supervised learning cannot tell the difference between the defective version and the repaired version, achieving an accuracy of 0.5125 and F-Score of 0.4936, which is nearly even chance. This shows that, on SCENARIO A the differences detected are due to the difference in execution when the defect exhibits as opposed to not.

4.4.3 *RQ3:* Prediction Accuracy on Varied Amounts of Data

Recall that our overall goal is to determine whether an approach that combines dynamic binary instrumentation with machine-learning is useful in detecting software executions that exhibit defects. This question evaluates the subgoal of determining how much data is necessary to make useful predictions. To answer this question, we compute the analyses used to evaluate *RQ1* in Section 4.4.1 with varying amounts of data on SCENARIO A, which is described in Section 4.2. We show results for all accuracy metrics in Table 4.3 and graph results for the F-Score metric in Figure 4.1 As mentioned earlier, the F-Score is a harmonic mean of precision and recall,

Table 4.2: Accuracy Metrics for Supervised Learning on a Defective Version and its Repaired Counterpart.

Scenario	Mean Acc.	Mean Prec.	Mean Rec.	Mean F-Score
Scenario A	0.9817	0.9837	0.9798	0.9817
Scenario B	0.9961	0.9944	0.9980	0.9962
Scenario C	1.0000	1.0000	1.0000	1.0000
Scenario D	0.9990	1.0000	0.9980	0.9990

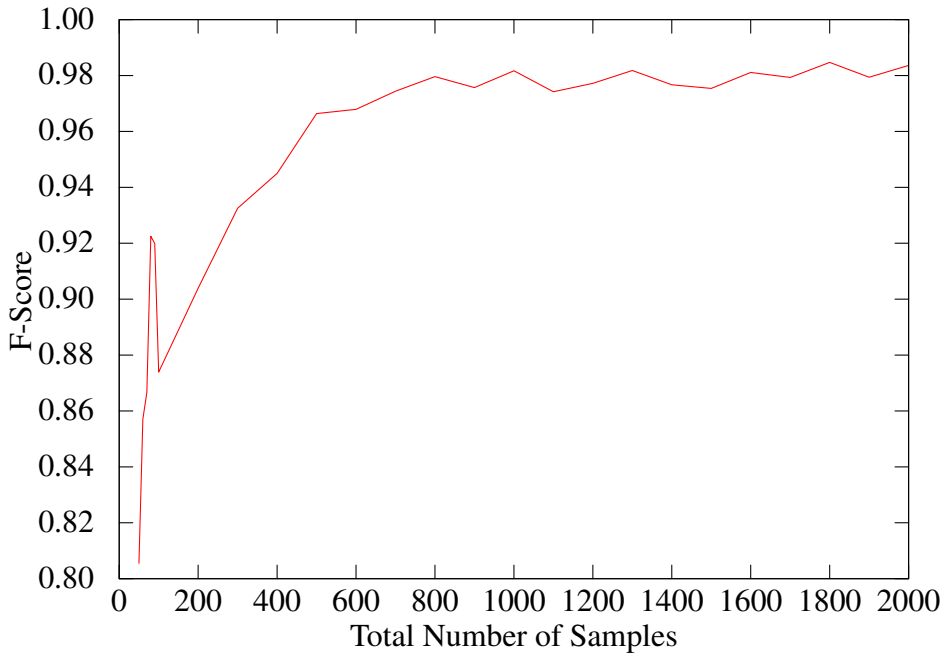

Figure 4.1: Supervised Learning on a Single Defective Version with Varied Amounts of Data

and represents a measure of the ability of the technique to detect all executions that exhibit the defect without flagging executions as defective when they are not. Higher is better. The X-axis represents the total number of samples included in K-fold validation for supervised learning. Note that the Y-axis starts at the value of 0.84 and that the lowest F-Score we observe is 0.8452 for the value of 50 samples. The F-Score remains above 0.96 for all sample sizes greater than or equal to 500. These data show that even with a comparatively small sample size of 50 samples, we get reasonably accurate data and that, although accuracy generally increases with additional samples, returns diminish after 500 samples.

Table 4.3: Accuracy Metrics and Approximate Training Time for Supervised Learning on a Single Defective Version With Varying Numbers of Samples.

Num. Samples	Mean Acc.	Mean Prec.	Mean Rec.	Mean F-Score	Train Time (min)
50	0.80	0.83	0.80	0.81	83.2
60	0.87	0.90	0.82	0.86	99.9
70	0.87	0.89	0.85	0.87	116.5
80	0.91	0.92	0.94	0.92	133.2
90	0.92	0.93	0.92	0.92	149.8
100	0.89	0.91	0.86	0.87	166.5
200	0.90	0.90	0.91	0.90	333.0
300	0.93	0.92	0.94	0.93	499.5
400	0.94	0.94	0.95	0.95	666.0
500	0.97	0.96	0.97	0.97	832.5
600	0.97	0.97	0.97	0.97	999.0
700	0.97	0.98	0.97	0.97	1165.5
800	0.98	0.98	0.98	0.98	1332.0
900	0.98	0.98	0.98	0.98	1498.5
1000	0.98	0.98	0.98	0.98	1665.0
1100	0.97	0.97	0.98	0.97	1831.5
1200	0.98	0.98	0.98	0.98	1998.0
1300	0.98	0.98	0.98	0.98	2164.5
1400	0.98	0.98	0.97	0.98	2331.0
1500	0.98	0.97	0.98	0.98	2497.5
1600	0.98	0.98	0.98	0.98	2664.0
1700	0.98	0.98	0.98	0.98	2830.5
1800	0.98	0.99	0.98	0.98	2997.0
1900	0.98	0.98	0.98	0.98	3163.5
2000	0.98	0.98	0.99	0.98	3330.0

5 Novelty Detection on Varied Robotics Programs

In ongoing work, I am testing techniques based on collecting low-level execution data on running robotics programs and using that data as an input to outlier-detection algorithms based on machine learning to determine which executions correspond to atypical or unsafe behavior.

I simulate two real robotics systems: ERLE and FETCH FREIGHT and one designed for research: BENCHMARK BOT. I replay inputs that produce expected behavior to gather nominal data. I then use structured transformations to generate inputs that may produce different behavior.

5.1 Method

I run several simulated robotics systems with dynamic binary instrumentation: Fetch Freight, BenchMark Bot, and Erle. Using event logs ("rosbags") that, when used as input to the corresponding simulation, produce nominal behavior, I obtain data corresponding to typical program behavior, which I use as baselines from which to build models of expected behavior in each of these systems. I collect additional data from additional executions on these inputs for evaluation. I also use a modified form of structured fuzzing (using a process based on processes described in Hutchinson et al. [38]) to create inputs that may cause the simulated program to exhibit errors. I run the simulations with each of the fuzzed inputs.

To create the model, I use a one-class classification algorithm to cluster the nominal data. Using an algorithm related to the LDCOF algorithm presented by Amer and Goldstein [4], I calculate each cluster based on the distance of each execution's feature vector from the center of the nearest cluster. The measure of local density of each cluster is:

$$\max_{v \in c}(\operatorname*{avg}_{u \in c}(d(v,u)))$$

To locate anomalies, I measure the distance of the feature vectors for each new execution from the center of the nearest cluster. For each system, I measure and compare three different anomaly detection techniques. In addition to low-level data collected by a customized VALGRIND tool, I also assess a technique based on process monitoring using Linux's built-in process monitoring utilities; and VALGRIND'S default memory profiler, MEMCHECK.

5.2 Evaluation

I measure anomaly detection against two rough proxies for an oracle for program behavior: core dumps and invariant violations. Each of these rough proxies can be a significant undercount of faults in program behavior. I measure whether anomaly detection based on low-level instrumentation with the customized VALGRIND tool, the Linux process monitoring utility, and the default VALGRIND MEMCHECK tool, fail to detect any faults that can be found by invariants or core dumps. These false negatives show the limitations of anomaly detection but also promote the case for using the technique in combination with other techniques. I also measure whether the anomaly detection technique identifies any anomalies that invariants or core dumps fail to identify. These identified anomalies may be false positives normal program behavior that is identified as anomalous or anomalies that the other techniques fail to find.

I also measure the stability and replicability of anomaly detection. I test the model on the data produced by each of several sets of a fixed number of executions: all but one of these sets of data are from fuzzed inputs and one set from the nominal input. The anomaly detection algorithm identifies a subset of the data points as coming from anomalous executions, i.e., executions that produce unintended behavior. To determine a label for each of the inputs, I determine whether the anomaly detection algorithm more commonly labels the data associated with the input as nominal or anomalous. I compare these labels with labels from the baseline performance metrics: invariants and core dumps.

5.3 Evaluation Subjects

A good data set on which to evaluate these experiments would include several simulated robotics systems. I have begun this work with simulations of versions of ARDUPILOT, FETCH FREIGHT, BENCHMARK BOT, and ERLE COPTER software. These simulations fit the desired criteria. I may gain access to simulations of additional systems. If so, I will assess whether any has properties that make it more desirable to evaluate, such as properties that are significantly different in any way from the existing systems. I plan to evaluate on at least three systems.

5.4 Expected Contributions

I expect to demonstrate that the combination of low-level instrumentation and an anomaly detection model trained on nominal data can detect most anomalous behavior that is detected by invariant violations and core dumps. In addition, I expect it to detect additional anomalous behavior. Some of this anomalous behavior may correspond to behavior that is simply unusual in the nominal data. For example, if there is error handling code that is not exercised in the nominal program behavior, the algorithm may detect the exercise of that code as an anomaly because the behavior is different than in the nominal case for which we trained. But I also expect to find anomalous behavior that corresponds to unintended program behavior that is not found by invariant detection or core dumps.

6 Intrusiveness Reduction

Time permitting, I propose to investigate several techniques to potentially reduce the intrusiveness of instrumentation while still collecting sufficient information to be useful to evaluate whether a program error is present. Recall that *intrusiveness* refers to the extent to which execution is perturbed by instrumentation. To this end, I propose to develop a measure of intrusiveness that makes sense in this context. Such a measure would be its own research contribution.

I then propose to evaluate several points in the design space of possible different techniques for reducing intrusiveness. I do not propose to exhaustively explore the space of all such potential techniques. Instead, I propose to evaluate approximately three points in the design space, including at least two of the techniques described below and at least two different configurations within one of the given techniques.

I will evaluate these techniques with respect to the accuracy metrics defined in Section 2.2.2, with particular reference to precision, recall, and F-measure. Further details on how these evaluation metrics are applied here can be found in Section 6.4. The work proposed in this section is additional work that will only be completed if time and circumstances permit.

6.1 Analysis of Smaller Units In Robotics Software

Most software in robotics and autonomous systems is large and complex. There are several distinct disadvantages to measuring the behavior of these programs over an entire system at once. One disadvantage is that instrumenting the entire system is impractical because it can incur overhead in portions of the program that are most sensitive to timing. My initial work shows that, in timing sensitive robotics systems, overhead from instrumentation can deform execution significantly enough that important portions of the programs are not reached. An additional disadvantage is that the distributed nature of many of these systems makes it difficult to keep an accurate count over all components without interfering with the systems' internal memory or communications.

Furthermore, an overall count is not the most precise measurement possible. It may lack the precision to be informative if, for example one portion of the program exhibits unusual behavior, which can be seen in a comparatively-small perturbation of the values of particular low-level signals over the course of running that portion of the program. If, in the normal variation of program executions, the values of those signals vary by more than the perturbation, the larger program behavior can obscure the behavior in the portion of the program. When we take summary data over the entire system, larger trends in system behavior may hide the smaller variations in the one node.

Focusing on smaller program units may have additional benefits. Restricting the scope of data collection to particular units or events may allow analysis of the data while the overall program is still running, allowing for flaw detection in an active program.

There is a possible pitfall of missing some systemic behavior by focusing on particular program portions. However, any decision about what to test or measure runs a similar risk of missing something not captured at that level.

6.1.1 Method

I propose restricting instrumentation to individual program units. I plan to primarily restrict instrumentation to portions of the programs in binary without reference to semantics. I plan to choose program portions simply, such as by arbitrarily choosing an address range to monitor or monitoring subprocesses.

One limitation to this approach is that, when I restrict analysis to individual program units, I may not choose the program units where the actual faults occur.

I will restrict data collection to the chosen program units. I will then test the data collected on those program units to see if that selection was useful.

For each program unit, I will conduct experiments in the same vein as the earlier experiments, but collecting instrumentation data only on the isolated program unit for each experiment.

The experiments will collect nominal executions that do not activate a known error and error executions that do activate that known error. I will build models based on the data collected and evaluate the models on held-out data corresponding to executions that do and do not exhibit the known error.

6.1.2 Expected Contributions

I expect to find that some types of errors are easier to detect when the scope of data collection is restricted. I also expect to gain insight into which approaches to scope restriction are most effective in the different types of programs. I hope to provide insight on how to identify program units that are useful for this kind of analysis, without using semantic program knowledge.

6.2 Signal Selection and Reduction

I have been picking signals somewhat arbitrarily, based on what is easy to measure and would add the least overhead to programs. Keeping overhead small is important for several reasons. For the technique to be reasonable to deploy, users will not accept a lot of overhead. In addition, in timing-sensitive programs, such as autonomous and robotics software, overhead may distort execution enough to change the behavior of the program. I have observed this phenomenon to manifest in timeout deadlines being missed, program events occurring in an order other than the one the program expects, and other errors. When the instrumentation changes program behavior to a significant extent, it is not useful as a tool of analysis for what the program would do under un-instrumented conditions.

However, it is important to choose signals intelligently in order to maximize predictive power. Different signals may have more predictive power on different programs. I would like to investigate which signals are most predictive for different situations.

6.2.1 Method

I will write a new customized tool based on the VALGRIND framework to collect many signals without adding so much additional overhead that the underlying program does not perform its basic functions. Writing this tool is a significant engineering challenge for several reasons. First, it is difficult to determine whether some of the subject programs are meeting their basic functions. Second, the programs under test are complex and exhibit nondeterministic behavior. By, "nondeterministic," behavior, I mean that, given the same set of inputs and same environment these programs may exhibit different behaviors. The nondeterministic nature of the tests add to the challenge of determining the extent to which instrumentation interferes with execution. In addition, different signals may increase overhead differently from each other and the amount of incremental overhead added by collecting a particular signal may vary based on the other signals already being collected. I will design the tool so that it is easy to select subsets of signals to be collected for each set of experiments, although such a design may interfere with some optimizations in the tool.

Once I have the enhanced tool I will run experiments on several different systems. The basic design of the experiments will be similar to those described in earlier sections, running on several different systems. However, in these experiments, I will first run the experiments with the entire signal set. I will then use techniques based in machine learning feature reduction to determine subsets of the signals that have strong predictive power for individual programs, across several programs, and with regard to different kinds of program errors.

I will then run new experiments in which I collect only reduced feature sets, as chosen by the feature selection described above. I will compare the effectiveness and overhead by conducting the following experiments and analyses:

- New experiments only collecting the selected reduced features.
- Comparing the speed and overhead of the new experiments with the ones that use the full feature set. Note that this will involve developing more sophisticated overhead analysis.
- Comparing the accuracy of the new experiments with the ones that use the full feature set.
- Analyze whether the features that are most predictive for any given program, set of programs, or bug, may provide insight about the differences among programs or types of bugs.

6.2.2 Expected Contributions

I expect to either demonstrate a connection or a lack of connection between the types of signals that are most predictive for a particular program type or particular bug type. I also expect to use these insights, in conjunction with insights drawn from my other proposed work, for developing techniques for instrumenting programs to gain meaningful information while incurring less overhead.

6.3 Sampling

As discussed in Section 6.2, keeping overhead small is important for several reasons. For the technique to be reasonable to deploy, users will not accept a lot of overhead. In addition, in timing-sensitive programs, such as autonomous and robotics software, overhead may distort execution enough to change the behavior of the program. I have observed this phenomenon to manifest in timeout deadlines being missed, program events occurring in an order other than the one the program expects, and other errors. When the instrumentation changes program behavior to a significant extent, it is not useful as a tool of analysis for what the program would do under un-instrumented conditions.

I propose to investigate whether I can make a tool that is less intrusive and still reasonably predictive by making use of sampling.

6.3.1 Method

I'm going to build a sampling based tool (based on others in the literature). Part of this work will involve a literature review to determine existing best practices in sampling and especially as applied to instrumentation and similar tasks.

The method for testing this tool's efficacy will be much like that discussed in Section 6.2.1, and I refer the reader to that section.

6.3.2 Expected Contributions

I expect to demonstrate the usefulness (or lack thereof) of sampling for instrumenting programs for low-level data to use to build accurate models.

6.4 Evaluation Metrics

I will evaluate the experiments in this sub-part by the accuracy metrics defined in Section 2.2.2, with particular reference to precision, recall, and F-measure. *Precision* is an appropriate metric because it measures the extent to which the technique identifies executions that actually exhibit abnormal behavior. Put another way, a precise algorithm does not deliver many false positives. False positives reduce the usefulness of the algorithm for any human or automated repair technique that consumes its output. *Recall* is an appropriate metric because it measures the extent to which the technique identifies all abnormal behavior and does not miss any, i.e., that it does not have many false negatives. False negatives reduce the usefulness of the algorithm because, with a high false negative rate, the algorithm could not be counted on to detect most of the behavior it is supposed to detect. *F-measure* balances precision and recall, so that neither metric is trivially maximized at the expense of the other.

I will also evaluate the experiments in this sub-part with reference to instrumentation intrusiveness. I will compare the program points reached at three different levels of instrumentation: minimal, this reduced level, and a heavier-weight instrumentation used in earlier experiments.

This metric is appropriate because the technique is most useful when it does not heavily influence the program's behavior by adding overhead.

6.5 Evaluation Subjects

A good data set on which to evaluate these experiments would include several simulated robotics systems. These systems should have logical smaller units and have timing-sensitive aspects of their operation. I already have access to and have worked with simulations of versions of ARDUPILOT, FETCH FREIGHT, BENCHMARK BOT, and ERLE COPTER, software. These simulations fit the desired criteria. I may gain access to simulations of additional systems. If so, I will assess whether any has properties that make it more desirable to evaluate, such as properties that are significantly different in any way from the existing systems. I plan to evaluate on at least three systems.

7 Proposed Timeline

Below, I list my proposed schedule with an expected defense date of December 2019. At the time of writing this proposal, I have completed the small program work detailed in Section 3 and the supervised learning on ARDUPILOT work detailed in Section 4. I have started the work on varied robotics programs detailed in Section 5. I have made initial investigations into the work proposed in Section 6.

During this time, I plan to submit papers on the results of the work to software engineering conferences and conferences in related fields, as appropriate.

November – December 2018

- Complete work on varied robotics systems at NREC and submit to DSN (December 7 deadline).

January 2019

- Rework ARDUPILOT work for new paper submission (ISSTA Deadline Jan 28 or FSE Deadline Feb 20).
- Literature review on sampling work.
- Implement sampling tool.

February 2019

- Seek additional systems and scenarios to test and conduct any additional needed work to make them work in experimental frameworks.
- Begin experiments using sampling tool.

March 2019

- Continue experiments using sampling tool on programs under test.

April 2019

- Begin implementing work on signal selection and reduction.
- Conduct signal selection and reduction experiments.

May 2019

- Begin implementing tools to detect and limit data collection to small program units, without making use of source code or debug symbols.
- Write up sampling and signal selection and reduction work (possible ISSRE deadline).

June 2019

- Continue implementation work for small program units.
- Begin experiments on small program units.

July 2019

- Continue experiments on small program units.

August 2019

- Implement tools that combine signal selection and reduction, sampling, and small program units.
- Conduct experiments to unite work on signal selection and reduction, sampling, and small program units.

September 2019

- Write up combined work (venue TBD).

October 2019

- Begin writing thesis document.

November 2019

- Write thesis document and prepare defense.

December 2019

- Defend and graduate.

8 Conclusion

In summary, I am working to use low-level execution information collected through dynamic binary instrumentation to build machine learning models to be used to determine whether program executions represent intended or usual behavior. I have completed work on small programs and on ARDUPILOT. I propose to extend the work to novelty detection on varied robotics programs, and to make use of analysis of smaller units in robotics software, signal selection and reduction, and sampling.

Bibliography

[1] Decision trees. http://scikit-learn.org/stable/modules/tree.html. 2.2.2

[2] Microsoft Zune affected by 'bug'. December 2008. URL http://news.bbc.co.uk/2/hi/technology/7806683.stm. 3.1

[3] Schiaparelli landing investigation makes progress, 2016. URL http://www.esa.int/Our_Activities/Space_Science/ExoMars/Schiaparelli_landing_investigation_makes_progress. Accessed Mar. 1, 2018. 1

[4] Mennatallah Amer and Markus Goldstein. Nearest-neighbor and clustering based anomaly detection algorithms for RapidMiner. In *RapidMiner Community Meeting and Conference*, RCOMM '12, pages 1–12, 2012. 2.1, 2.2.2, 5.1

[5] Sara Abbaspour Asadollah, Hans Hansson, Daniel Sundmark, and Sigrid Eldh. Towards classification of concurrency bugs based on observable properties. In *Complex Faults and Failures in Large Software Systems*, COUFLESS '15, pages 41–47, 2015. 2.1

[6] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004. 2.1

[7] Earl T. Barr, Mark Harman, Phil Mcminn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing : A survey. *IEEE Transactions on Software Engineering*, 41 (5):507–525, 2015. 2.1

[8] Irad Ben-Gal. Outlier detection. In *Data mining and knowledge discovery handbook*, pages 117–130. Springer, 2010. 2.1

[9] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. Debugging distributed systems. *Queue*, 14(2):91–110, 2016. 2.1

[10] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *USENIX Security Symposium*, SS '07, pages 15:1–15:16, 2007. 2.1

[11] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010. ISBN 0136108040, 9780136108047. 3.2.1

[12] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight

detection of infinite loops at runtime. In *Automated Software Engineering*, ASE '09, pages 161–169, 2009. 3.1

[13] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. Entropy-based test generation for improved fault localization. In *Automated Software Engineering*, ASE '13, pages 257–267, 2013. 3.3, 3.4.2

[14] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with Jolt. In *European Conference on Object Oriented Programming*, pages 609–633, 2011. 3.1

[15] W.K. Chan, S.C. Cheung, Jeffrey C.F. Ho, and T.H. Tse. PAT: A pattern classification approach to automatic reference oracles for the testing of mesh simplification programs. *Journal of Systems and Software*, 82(3):422–434, 2009. 2.1

[16] D. Coppit and J.M. Haddox-Schatz. On the use of specification-based assertions as test oracles. In *Software Engineering Workshop, 2005. 29th Annual IEEE/NASA*, SEW '05, pages 305–314, 2005. 2.1

[17] Domenico Cotroneo, Michael Grottke, Roberto Natella, Roberto Pietrantuono, and Kishor S. Trivedi. Fault triggers in open-source software: An experience report. In *International Symposium on Software Reliability Engineering*, ISSRE '13, pages 178–187, 2013. 2.1

[18] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*. 2000. ISBN 0-521-78019-5. 2.2.2

[19] Dorothy E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987. 2.1

[20] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: The distribution of program failures in a profile space. In *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering*, ESEC/FSE '01, pages 246–255, 2001. 2.1, 2.1

[21] William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *International Conference on Software Engineering*, ICSE '01, pages 339–348, 2001. 2.1

[22] Andrew David Eisenberg and Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *International Conference on Software Maintenance*, ICSM '05, pages 337–346, 2005. 2.1

[23] Khaled El Emam and Isabella Wieczorek. The repeatability of code defect classifications. In *International Symposium on Software Reliability Engineering*, ISSRE '98, pages 322–333, 1998. 2.1

[24] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, 2001. 2.1, 3

[25] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on*

Software Engineering, 27(2):99–123, 2001. 2.1

[26] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, NguyenNumericalInvariants2017, ErnstDaikon2001, LorenzoliBehavioral2008, HangalIodine2005, BeschastnikhTemporal2011, RatcliffInvariants2011, ErnstDaikon200569:35–45, 2007. 2.1, 2.1

[27] Stephanie Forrest and Westley Weimer. The challenges of sensing and repairing software defects in autonomous systems. Technical report, Regents of the University of New Mexico, 2014. 1, 1, 2.1, 2.1

[28] Laura Fraade-Blanar, Marjory S. Blumenthal, James M. Anderson, and Nidhi Kalra. Measuring automated vehicle safety: Forging a framework. Technical report, RAND Corporation, 2018. URL `https://www.rand.org/pubs/research_reports/RR2662.html`. 1, 2.1

[29] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *Transactions on Software Engineering*, 38(2):278–292, 2012. 2.1

[30] Kambiz Frounchi, Lionel C. Briand, Leo Grady, Yvan Labiche, and Rajesh Subramanyan. Automating image segmentation verification and validation by learning test oracles. *Information and Software Technology*, 53(12):1337–1348, 2011. 2.1

[31] Michael Grottke, Allen P. Nikora, and Kishor S. Trivedi. An empirical investigation of fault types in space mission system software. In *Dependable Systems Networks*, DSN '10, pages 447–456, 2010. 2.1

[32] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, ICSE '02, pages 291–301, 2002. 2.1

[33] Sudheendra Hangal, Naveen Chandra, Sridhar Narayanan, and Sandeep Chakravorty. IODINE: A tool to automatically infer dynamic invariants for hardware designs. In *Design Automation Conference*, DAC '05, pages 775–778, 2005. 2.1

[34] Murali Haran, Alan Karr, Alessandro Orso, Adam Porter, and Ashish Sanil. Applying classification techniques to remotely-collected program execution data. *SIGSOFT Software Engineering Notes*, 30(5):146–155, 2005. ISSN 0163-5948. 2.1

[35] Murali Haran, Alan Karr, Michael Last, Alessandro Orso, Adam A. Porter, Ashish Sanil, and Sandro Fouche. Techniques for classifying executions of deployed software to support software engineering tasks. *IEEE Transactions on Software Engineering*, 33(5):287–304, 2007. 2.1

[36] Kennet Henningsson and Claes Wohlin. Assuring fault classification agreement - an empirical evaluation. In *International Symposium on Empirical Software Engineering*, ISESE '04, pages 95–104, 2004. 2.1

[37] Victoria J. Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004. 2.1, 3.3

[38] Casidhe Hutchison, Milda Zizyte, Patrick E. Lanigan, David Guttendorf, Michael Wag-

ner, Claire Le Goues, and Philip Koopman. Robustness testing of autonomy software. In *International Conference on Software Engineering - Software Engineering in Practice*, ICSE-SEIP '18, pages 276–285, 2018. 1, 2.1, 5.1

[39] Cylance Inc. Cylance delivers first AI driven endpoint detection and response solution with introduction of cylanceoptics, . URL https://www.cylance.com/en-us/company/news-and-press/press-releases/cylance-delivers-first-ai-driven-endpoint-detection-and-response-soluti
html. 2.1

[40] Cylance Inc. Cylance(R) prevention-first security with CylancePROTECT(R) and CylanceOPTICS(TM), . URL https://s7d2.scene7.com/is/content/cylance/prod/cylance-web/en-us/resources/knowledge-center/resource-library/briefs/CylanceOPTICS_Solution_Brief.pdf. 2.1

[41] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5):649–678, 2011. 3.3

[42] Upulee Kanewala and James M. Bieman. Techniques for testing scientific programs without an oracle. In *Software Engineering for Computational Science and Engineering*, SE-CSE '13, pages 48–57, 2013. 2.1

[43] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, Jun 2010. 2.1

[44] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, Feb 2014. 2.1

[45] Philip Koopman and Michael Wagner. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine*, 9(1):90–96, 2017. 1, 2.1

[46] Philip Koopman and Michael Wagner. Toward a framework for highly automated vehicle safety validation. In *WCX World Congress Experience*, WCX '18. SAE International, 2018. URL https://doi.org/10.4271/2018-01-1071. 1, 2.1

[47] Sotiris B. Kotsiantis, I. Zaharakis, and P. Pintelas. Supervised machine learning: A review of classification techniques. In *Emerging Artificial Intelligence Applications in Computer Engineering*, pages 3–24, 2007. 2.2.2

[48] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, ICSE '12, pages 3–13, 2012. 3.2.2, 3.3, 3.4.2

[49] David Leon, Andy Podgurski, and Lee J. White. Multivariate visualization in observation-based testing. In *International Conference on Software Engineering*, ICSE '00, pages 116–125, 2000. 2.1

[50] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai.

Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Architectural and System Support for Improving Software Dependability*, ASID '06, pages 25–33, 2006. 2.1

[51] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, PLDI '03, pages 141–154, 2003. 2.1

[52] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Programming Language Design and Implementation*, PLDI '05, pages 15–26, 2005. 2.1

[53] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS '08, pages 329–339, 2008. 2.1

[54] Sixing Lu and Roman Lysecky. Analysis of control flow events for timing-based runtime anomaly detection. In *Workshop on Embedded Systems Security*, WESS '15, pages 3:1–3:8, 2015. 2.1, 2.1

[55] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005. 2.2.1, 2.2.1

[56] Alexis C. Madrigal. Inside Waymo's secret world for training self-driving cars. *The Atlantic*, August 2017. 1, 2.1

[57] Chengying Mao and Yansheng Lu. Extracting the representative failure executions via clustering analysis based on Markov profile model. In *Advanced Data Mining and Applications*, ADMA '05, pages 217–224, 2005. 2.1

[58] Nicholas Nethercote. Dynamic binary analysis and instrumentation. Technical Report UCAM-CL-TR-606, University of Cambridge, Computer Laboratory, 2004. URL `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf`. 2.2.1

[59] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, PLDI '07, pages 89–100, 2007. 2.2.1

[60] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *European Converence on Object-Oriented Programming*, ECOOP '05, pages 504–527, 2005. 2.1

[61] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, ICSE '07, pages 75–84, 2007. 2.1

[62] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot,

and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011. 2.2.2

[63] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, and Greg Sullivan. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102, 2009. 2.1, 2.1

[64] Mauro Pezze and Cheng Zhang. Automated test oracles: A survey. *Advances in Computers*, 95:1–48, 2014. 2.1

[65] Marco A. F. Pimentel, David A. Clifton, Lei Clifton, and Lionel Tarassenko. A review of novelty detection. *Signal Processing*, 99:215–249, 2014. 2.1

[66] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *International Conference on Software Engineering*, ICSE '10, pages 485–494, 2010. 2.1

[67] Muhammad Usman Sanwal and Osman Hasan. Formal verification of cyber-physical systems: Coping with continuous elements. In *Computational Science and Its Applications*, ICCSA '13, pages 358–371, 2013. 2.1

[68] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 317–328, 2013. 2.1

[69] K. Shrestha and M.J. Rutherford. An empirical evaluation of assertions as oracles. In *Software Testing, Verification and Validation*, ICST '11, pages 110–119, 2011. 2.1

[70] Thierry Sotiropoulos, Hélène Waeselynck, and Jérémie Guiochet. Can robot navigation bugs be found in simulation? an exploratory study. In *Software Quality, Reliability and Security*, QRS '17, pages 150–159, 2017. 2.1

[71] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl. Automated oracle creation support, or: How i learned to stop worrying about fault propagation and love mutation testing. In *International Conference on Software Engineering*, ICSE '12, pages 870–880, 2012. 2.1

[72] Gerald Steinbauer. A survey about faults of robots used in robocup. In Xiaoping Chen, Peter Stone, Luis Enrique Sucar, and Tijn van der Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*, pages 344–355. Berlin, Heidelberg, 2013. 2.1

[73] Andreas Theissler. Detecting known and unknown faults in automotive systems using ensemble-based anomaly detection. *Knowledge-Based Systems*, 123:163–173, 2017. 2.1

[74] Christopher Steven Timperley, Afsoon Afzal, Deborah S. Katz, Jam Marcos Hernandez, and Claire Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *International Conference on Software Testing, Validation, and Verification*, ICST '18, pages 331–342, 2018. 2.1, 4.1

[75] C. E. Tuncali, T. P. Pavlic, and G. Fainekos. Utilizing S-TaLiRo as an automatic test generation framework for autonomous vehicles. In *Intelligent Transportation Systems*, ITSC '16, pages 1470–1475, 2016. 2.1

[76] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Conference on Computer and Communications Security*, CCS '02, pages 255–264, 2002. 2.1

[77] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming*, ECOOP '06, pages 380–403, 2006. 2.1

[78] Alice X. Zheng, Michael I. Jordan, Ben Liblit, and Alex Aiken. Statistical debugging of sampled programs. In *Neural Information Processing Systems*, NIPS '04, pages 603–610, 2004. 2.1

[79] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *International Conference on Machine Learning*, ICML '06, pages 1105–1112, 2006. 2.1

[80] Xi Zheng and Christine Julien. Verification and validation in cyber physical systems: Research challenges and a way forward. In *Software Engineering for Smart Cyber-Physical Systems*, pages 15–18, 2015. 2.1

[81] Xi Zheng, Christine Julien, Miryung Kim, and Sarfraz Khurshid. Perceptions on the state of the art in verification and validation in cyber-physical systems. *IEEE Systems Journal*, 11(4):2614–2627, Dec 2017. 2.1

[82] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. *IEEE Security and Privacy*, 7(2):87–90, 2009. 1