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

Deborah Stephanie Surden Katz

CMU-CS-20-129
September 2020

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Claire Le Goues (Chair)
Philip Koopman
Eric Schulte (GrammaTech, Inc.)
Daniel Siewiorek

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

This research was sponsored in part by U.S. Army Contracting Command award number: W900KK16C0006 and in part by the Test Resource Management Center (TRMC) Test & Evaluation / Science & Technology (T&E/S&T) Program through the U.S. Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI) under Contract No. W900KK-16-C-0006, "Robustness Inside-Out Testing (RIOT)." DISTRIBUTION STATEMENT A — Approved for public release; distribution is unlimited. NAVAIR Case #2019-615.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

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

Abstract

Autonomous and robotics systems (ARSs) – systems that are designed to react independently and without human supervision to environmental stimuli – are complex and difficult to supervise but are an increasingly large portion of the systems currently being developed and in use. Quality assurance for these systems is complex, and the software for these systems contains many faults.

My key insight is that typical program behavior is a basis for determining whether a program is operating within its normal parameters. To leverage this, I record summaries of program execution behavior using low-level monitoring to characterize each execution. By aggregating low-level execution data over many executions, I create a picture of typical program behavior; different behavior may indicate unintended behavior. My techniques use the data as input to machine learning algorithms which build models of expected behavior. These models analyze individual program executions to predict 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 demonstrate the efficacy of these techniques to detect software failures on small programs and in simulation on the ARDUPILOT autonomous vehicle and on other ARSs based on the Robot Operating System (ROS).

I observe that ARSs are well-suited to low-level monitoring because they are cyber-physical. Although in other situations such monitoring may create intolerable overhead, these distributed systems that interact with the real world have time or cycles that would otherwise be spent waiting for real world events. As such, ARSs are well-situated to absorb overhead that monitoring generates. However, ARSs often do have timing-sensitive components, for example, deadlines and timeouts that, if missed, cause the system to abort. To this end, I measure the extent to which ARSs can absorb artificially-inserted timing delays.

Acknowledgements

I thank my advisor, Claire Le Goues. Finding Claire as an advisor was the best thing that could have happened for my graduate school career. She is always one hundred percent on my side. She has taught me how to be a researcher and helped me learn in so many ways. Whenever things get tough, I know I can count on her to help me find a way forward.

I would also like to thank my committee for their guidance, input, and support of my research: Phil Koopman, Eric Schulte, and Dan Siewiorek.

I thank my undergraduate thesis advisor, Scott F. Kaplan, for helping me to discover that I enjoy computer science research and helping me get started.

I thank my research colleagues and collaborators for their encouragement, their contributions to this work, their willingness to lend a hand, and our brainstorming sessions. In no particular order and in a necessarily incomplete list: my SquaresLab colleagues: Zack, Mau, Afsoon, Chris, Rijnard, Cody, Jeremy, Leo, Zhen, and Sophia; and my ASTAA and RIOT team colleagues at NREC: Cas, Milda, Dave, Eric, Pat, Bill, Trent, Adi, and Matt.

There have been many others in the computer science community at CMU who have contributed to my success. I especially want to recognize Deb Cavlovich who seems to be the center of the CSD Ph.D. program and who always knows how to help whenever there is an issue.

I have had the great good fortune to be surrounded by many wonderful people through the various stages of my life. They have supported me in ways both practical and intangible and have provided good advice and companionship.

I thank my family for their constant love and support, especially my parents and grandparents to whom education was and is so very important. I cannot thank my parents enough. Esther Surden and Harvey Katz set excellent examples and showed me unwavering support as I pursued graduate degrees in two widely separated fields. Dad taught me the rudiments of programming when I was very young, and Mom taught me about the female pioneers of programming and told me stories about her experiences covering the "minicomputer" industry in the 1970s.

I am extremely fortunate to have so many family members who I can always count on. In another necessarily incomplete list: my brothers Greg and Josh and sisters-in-law Paige and Kiara; Lauren, who is as close as a sister, and my brother-in-law Adam; Molly, who rescued me when I was ill; and my aunts, uncles, cousins, nieces, and grandparents. And of course, I have to thank my exceptional cat Penelope, who joined me near the beginning of grad school and who has been my COVID stay-at-home companion.

I want to specifically acknowledge my recently deceased grandmother, Sara Katz-Zaroff, who valued education so very highly. Long ago, she told her children, in all seriousness, that they would be allowed to leave school only after they had earned their Ph.D.s. She would have been very proud to see this document and would have displayed it on her coffee table.

I have been beyond fortunate to have friends who I value like family. In no particular order and in another necessarily incomplete list: Lisa, Katie, Rikita, Kristin, Chris, Kerri, Maryna, Josh, Kristy, João, and Anna. Thank you.

I also want to thank my grad school friends and the groups in which I found camaraderie during graduate school: Yarnivores, the SCS Grad Musical, and Planworld.

There have been so many others who have touched my life in a positive way and made this possible. I am very fortunate to have each of you. Thank you.

Contents

1 Introduction

It is increasingly important to understand Autonomous and Robotics Systems (ARSs) and guard against unintended behavior. ARSs interact with various stimuli in their environments. They are often designed to react independently and without human supervision. These systems can be big and are often complex, which makes it difficult for humans to fully understand their operation and understand when and whether they are behaving as intended. The difficulty of understanding their behavior is compounded because these systems are often used in situations that can make it difficult to observe and deliver commands to them, such as in systems deployed in space or otherwise operating autonomously. At the same time, these systems are an increasingly large portion of the systems being developed and in use [38].

While the properties of ARSs make it difficult to ensure correct operation, it is extremely important that they operate safely. These systems are being deployed in safety-critical situations, such as large vehicles with autonomous components operating the the presence of pedestrians [9, 38]. They are also being deployed in situations that make their failures expensive. 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. The inaccessibility of the Martian environment limited opportunities to detect and repair the failure as it was happening, before the catastrophic consequences occurred. The problem was likely due to an implementation mistake that failed to account for timing inconsistencies between sensors [3, 56].

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 [39, 67]. 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 [66]. But while these systems should anticipate all possible scenarios, such anticipation is not always possible. Often, their goals and 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, a collision may occur, but there may be no internally-observable failure with respect to that failure mode, as defined by the system's original quality parameters [66]. Similarly, a failure to detect a pedestrian may not result in an externally-observable failure if the vehicle, coincidentally, travels a path that does not cause a collision with the pedestrian.

Because ARS systems pose such difficulties in Quality Assurance (QA), various QA techniques can contribute productively and substantially to improving their quality. The techniques

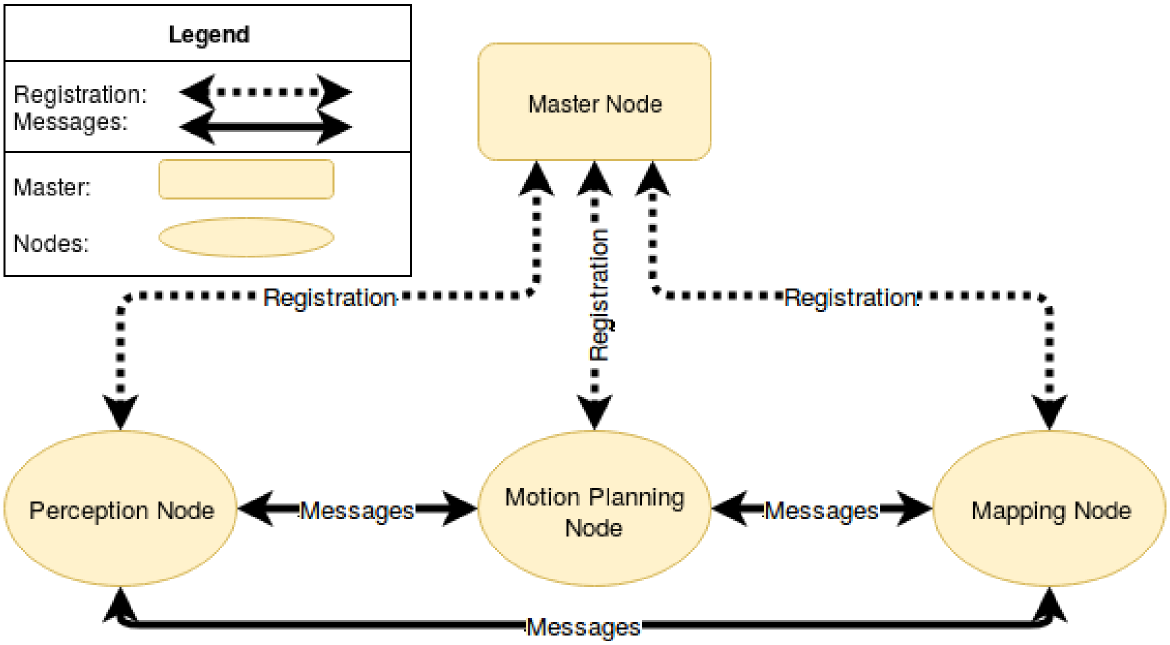

Figure 1.1: Simplified Robot Architecture

presented here are focused on one aspect of software quality – locating unusual program behavior. They are intended to be used in conjunction with other QA techniques to improve ARS software.

In many of these systems, early knowledge of a fault – along with tools that enable intervention – would allow costly and safety-critical failures to be avoided [115]. 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. Alternatively, it is possible to detect faults in simulation, before deployment. Detecting software faults in ARS before they occur has the potential to save catastrophic damage and lives.

1.1 Illustrative Example

The following example is based on a real robotics framework — Robot Operating System (ROS) — and a real bug that my colleagues and I found in a system based on that framework. This example illustrates how the nature of the system architecture lends itself to certain kinds of bugs and also makes the systems amenable to error detection by low-level monitoring of values at the instruction level.

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

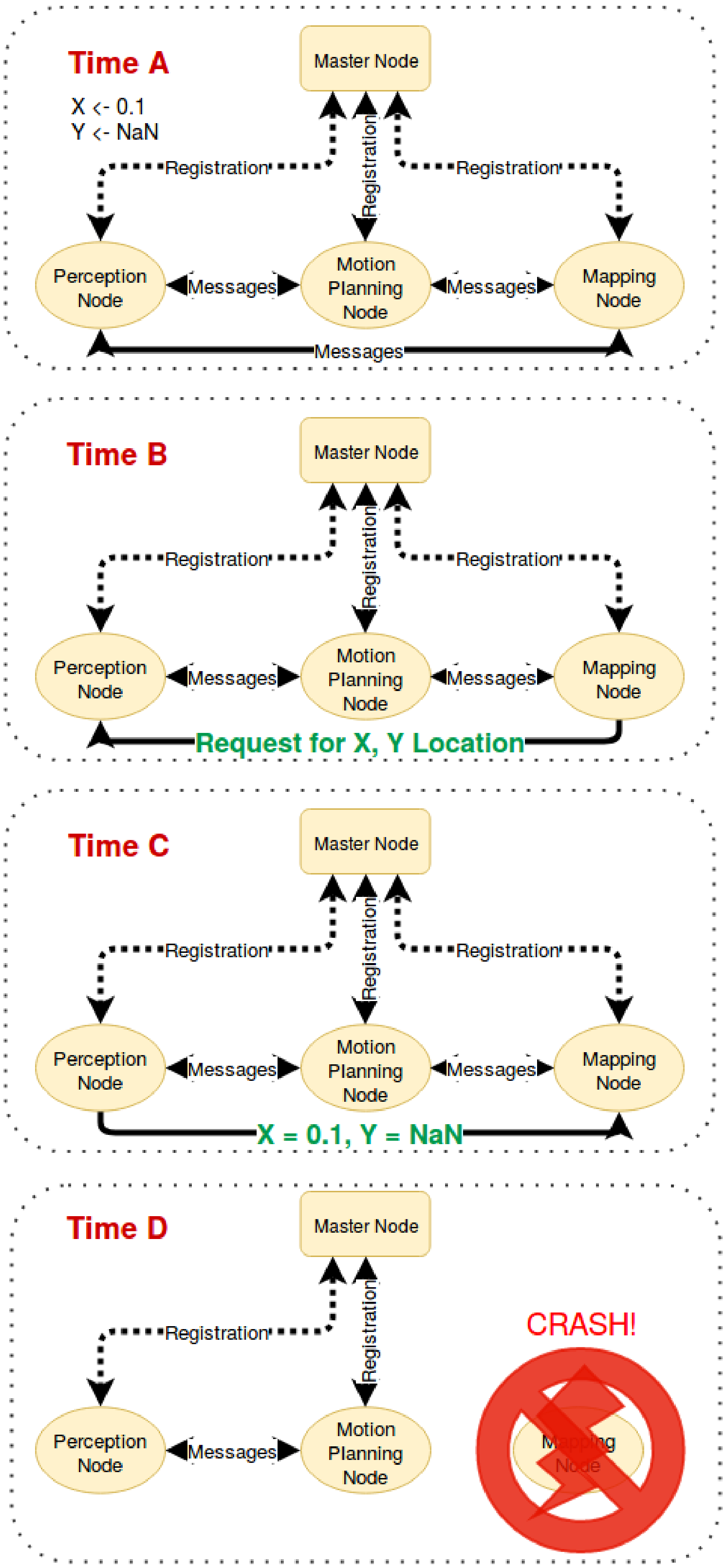

Figure 1.2: A Simplified Crash Sequence

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.

This 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.

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 the unusual and undesired storage of the NaN can be detected 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. Low-level signals, individually or in combination, can be used to detect errors at multiple points in execution, which in turn, can aid in fault diagnosis. Alternatively, low-level monitoring can detect these behaviors in simulation, allowing the problem to be fixed before deployment. 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 and can be detected by monitoring behavior at the instruction level.

1.2 Insights and Thesis Statement

Although there are many techniques for detecting software failures, some of which have been applied to Cyber-Physical Systems (CPSs) and ARS, these techniques have limitations when applied to ARS. I discuss these techniques and their limitations in more detail in Section 2.1.

My key insights are:

- 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.
- ARSs are well-suited to low-level monitoring because much of the overhead can be hidden in time that would have otherwise been spent waiting.

1.2.1 Thesis Statement

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, while hiding monitoring overhead in normal system behavior.

1.3 Approach

For an intuitive understanding of this approach, recall the example presented in Figure 1.2 and the accompanying text. Monitoring low-level signals in a given node, such as the values stored to memory in the perception node, provides 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 of ARS. 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 or specifications.

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 complex 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 be characterized by two or more signals deviating from their normal patterns at the same time.

ARS are better adapted than other systems to the use of low-level monitoring. In many situations such monitoring is considered intrusive and creates overhead at intolerable levels[80, 85]. However, distributed systems that interact with the real world, such as ARS, are well-situated to absorb some of the overhead that such techniques can generate. The components of these systems spend time waiting for events to happen, either events in the real world or messages sent from other components. They are, thus, often not as sensitive to the overhead that monitoring techniques can add, as they can absorb the delays in time that otherwise would have been spent waiting. However, these systems often do have timing-sensitive components, for example, deadlines and timeouts in startup that, if missed, cause the system to abort, so additional overhead must be handled carefully.

Faults in ARS software are well-adapted to being detected by observation of low-level signals. As described above, ARS software has particular challenges that make quality assurance difficult. These challenges include a potential partial or total lack of source code (these systems are often "black boxes"); a lack of specificity in specifications, allowing for many ways in which systems can behave correctly and making it difficult to distinguish failure modes from correct behavior (the systems are "noisy"); a potential to not notice failure modes when they are masked by other behaviors in a complex system; and testing these systems can incur a high cost in time and equipment (the systems are "costly")[56].

Solutions based on deriving patterns from observed low-level behavior are well-adapted to meet these challenges for the following reasons. They do not require source code, allowing for these systems to be treated as black boxes. They can derive patterns and models over any number of usual behavior modes, providing the basis to detect when a system deviates from those usual behavior modes, making these techniques well-adapted to noisy systems. The techniques can be run on robotics software in simulation, detecting potential errors before deployment on real hardware, where they could do damage to the hardware and the real world [56].[1] These techniques can focus on individual components of a complex system at a time, allowing for potential errors to be found in each component; they are also applicable to larger parts of systems and entire systems. Testing on several different levels can reduce the likelihood that a bug is masked by behavior in another system. In addition, the ability to run in simulation makes these techniques far less costly, both in time and equipment, than running on real robotics hardware. Simulations only need compute time on standard computing equipment, as opposed to the specialized robot hardware. These techniques do not require access to system source code, as long as a working system installation can be run and monitored. As such, they can apply when source code is unavailable in whole or in part, as is the case in many situations where one might test an ARS or its components.

Because (a) these low-level monitoring techniques can reveal valuable insights about these systems, and (b) these systems are well-positioned to absorb overhead from these techniques, but (c) there are situations in which overhead is not tolerable, I studied the amount of overhead tolerable to robotics systems and under what circumstances overhead deforms execution. This work is detailed in Chapter 6.

1.4 Contributions

The key contributions of this thesis are:

- I set out a family of techniques that detect unusual or incorrect behavior in software by monitoring low-level execution behavior and use the recorded data to generate machine learning models.
- I demonstrate that these techniques work to detect unusual behavior and errors on small programs, on the ARDUPILOT autonomous software for quadcopters, and on several other

[1] An example of an error detected in simulation that can cause physical damage can be found here: `https://www.youtube.com/watch?v=kK6iKwjKA54`

ARSs.

- I demonstrate that the effects of timing delays on simulated robotics systems can often be absorbed and explore the degree to which and the circumstances under which these systems can absorb the delays. This shows that simulated robotics systems are well-adapted to absorb overhead from low-level monitoring.

1.5 Evaluation and Metrics

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 as 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 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. "Efficiency," refers to the degree to which instrumentation adds additional time to program executions; the less time added, the higher the efficiency. To measure these factors, I evaluate the experiments with respect to the effect that instrumentation has on program execution. To study intrusiveness, I examine the externally-observable behavior of ARS with and without artificially-injected timing delays. This metric is appropriate because monitoring techniques are most useful when they do not heavily alter the system's behavior by adding overhead.

2 Review of Literature and Background

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

2.1 Related 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 [15]. Robotics and autonomous systems often incorporate components from multiple suppliers, and source code is not always available for the components [38]. 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 [96].

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 [89].

Eisenberg et al. [33] 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 [29]." This approach makes use of dynamic analysis through instrumentation [73].

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 [11, 54, 92].

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 in Machine Learning (ML). The third group of techniques creates a model from normal (nominal) data and identifies whether new data fits within that model [54].

This third group of approaches is common and is analogous to a one-class classification problem in the context of ML. 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, The other data are the anomalies — the data that does not fit into the nominal class. One-class algorithms usually assume that the underlying data set contains many more nominal data points than data points corresponding to any anomalous classes [31, 92].

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 as automated invariant detection and specification mining [37].

Many anomaly, novelty, and outlier detection approaches, as described above, make use of clustering algorithms, which arrange similar data points into groups [42]. Clustering approaches for anomaly detection 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 [6].

Application of Clustering Algorithms to Software Failure Detection Work has applied clustering to identification of software failures [30, 52, 79, 83]. Haran et al. set out a general framework for using execution data as a basis for classifying whether program outcomes are successful [51, 52], including the use of clustering. Dickinson et al. focus on using clustering for identifying executions worthy of further study [29, 30]. Mao and Lu [83] propose using more complex clustering based in Markov models to identify which failure executions typify failure modes. Note that in Mao and Lu's approach, clustering is used to find executions that are typical, not those that are anomalies or outliers.

The techniques presented here use off-the-shelf clustering algorithms [42] and modifications of existing clustering algorithms [6], which create the models on which I build anomaly detection.

Intrusion Detection Work on intrusion detection, especially host-based intrusion detection, looks at techniques for determining whether an adversarial attacker has gained access to the host system [27, 106]. It is a subset of dynamic analysis and is strongly related to anomaly detection. At a high level, an Intrusion Detection System (IDS) monitors 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 [79].

IDSs 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 [106]. 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 [57, 58].

Yu et al. use information gathered from hardware performance counters to detect control flow hijacking attacks [109]. Using this low-level information reduces overhead. They establish distinct patterns that occur in particular hardware counters when control flow hijacking attempts occur.

Systems that interact with the real world — CPSs — are subject to temporal constraints. Several approaches have used deviations in the timing properties of CPSs as the basis for an IDS [48, 93].

Statistical Fault Identification Liblit et al. propose a collection of techniques that identify program faults using statistical techniques [76, 77, 110, 111]. The techniques require source code and semantic knowledge of the program. They 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 [76]. Subsequent refinements to the techniques included uniting approaches for deterministic and nondeterministic bugs [110]; increasing the usefulness of the program predicates identified [111]; and separately identifying the effects of different bugs [77].

Automated Invariant Detection and Specification Mining 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 [35].

One of the pioneers in this area is DAIKON [36, 37]. 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. The ability to analyze software without needing software is desirable because many complex ARSs do not have source code available for all or part of the system at the time when the system is tested.

Other approaches such as DIDUCE [49, 50] 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 [89]. 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, tools such as CLEARVIEW could be used in conjunction with techniques such as those described in this work.

Automated invariant detection is closely related to specification mining, which also includes techniques for creating formal models from properties inferred from observation of program execution [70]. Specification mining techniques have grown to include various other approaches, such as the incorporation of machine learning and deep learning techniques [71].

These techniques have also been applied to ARSs [60]. Jiang et al. automatically derive specific invariants from messages passed in robotics systems and develop specific system monitors to correspond to these invariants. This is in contrast to a more general approach, building models that can encompass many kinds of deviations.

The Oracle Problem This work builds on earlier work in software testing, which addresses the problem of trying to figure out whether a program is behaving as intended, which is known as the *oracle problem*. Several works provide a useful summary of oracle problem research [10, 90].

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 [24, 41, 86, 87, 90, 97, 99, 107]. Techniques described in Chapters 3, 4, and 5 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 [61] 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 [20, 43]. While one might expect these approaches to be similar to those described 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, such as in CPSs which interact with the real world. If a CPS exhibits unsafe behavior at any point in operation, it could cause damage to property or life.

Verification and Formal Methods Much work has gone into formal verification of CPSs 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 CPSs, 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 [113, 114].

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 [95].

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 [64, 65, 113].

Testing Autonomous and Robotics Systems (ARSs) Testing ARSs presents problems unique to those domains. Beschastnikh et al. outline some of the challenges and drawbacks to existing approaches to debugging distributed systems, such as ARS [14]. 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 [98]. Tuncali et al. define a *robustness function* for determining how far an ARS is from violating its parameters [105]. Notably, this approach relies on well-defined system requirements, which are absent in many systems. Timperley et al. attempt to categorize real bugs reported in the ARDUPILOT autonomous vehicle software as to whether they can be reproduced and/or detected in simulation [103]. 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 [23, 82]. However, as shown in our survey of developers, there are significant challenges in using simulation for systematic testing of ARS [5].

Recent work in simulation for ARS has looked at a range of issues, including: accurately simulating the human experience of being in a self-driving car [108]; accurately simulating the vehicle hardware and its interaction with the environment [21, 32]; and automatically generating test scenarios for the vehicles in simulation [4, 44, 45].

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, use those evaluations to improve the safety of those vehicles [39].

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 [66]. 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 [67]. Hutchison et al. outline a framework for robustness testing of robotics and autonomous systems, highlighting the differences from traditional software [56, 63]. 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 [38].

Theissler presents work using anomaly detection on data gathered in an automotive context [101]. 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 ARS.

Additional work focuses on testing sub-systems of ARS, especially those that depend on machine learning and deep learning [91, 102]. While much of this work focuses exclusively on the perception systems, this subsystem has some typical properties, such as noise and nondeterminism, that exemplify the difficulties in testing other parts of ARS.

Timing in Cyber-Physical Systems (CPSs) Systems that interact with the real world — CPSs — are subject to temporal constraints, which limits the techniques that can be applied in analysis at execution time [22, 48, 112]. These constraints inspire the testing of these systems in simulation to the extent possible. However, as I demonstrate in Chapter 6, some of these systems' overhead can be absorbed while the components are waiting for real world events.

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 [7, 8, 25, 34, 46, 53, 75, 78, 94, 100]. 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 [8]. 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 (DBI)

Dynamic Binary Instrumentation (DBI) 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. DBI operates on the level of object code (pre-linking) or executable code (post-linking). DBI 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 [80, 84].

PIN My initial experiments use PIN version 2.13, to instrument the subject programs and collect the data for model construction. PIN is a DBI tool [80] 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. 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 DBI 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.

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

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

[3] http://valgrind.org/

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

The experiments described in this document use various machine learning algorithms They 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 [88]. The version of Scikit Learn is indicated with each individual experiment. In some cases, the experiments use custom algorithms or customized versions of existing algorithms. The algorithms used are noted in each instance.

Supervised Models

Supervised learning takes a set of training examples with given labels (e.g., each data point is labeled “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 [69]. 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 [26].

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.

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.

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

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

For some of the preliminary work, I use 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, I use a more sophisticated distance metric in conjunction with a one-class clustering algorithm. This metric is inspired by the LDCOF algorithm [6]. 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 certain 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 to get a supervised algorithm to produce a model that is not weighted towards the more frequent data class. 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.

Assessing Machine Learning Models

I use the following measurements to assess the accuracy of machine learning models. These measurements apply to 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

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

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

- **Precision** *(Prec)* The ratio of returned labels that are correct: $TP/(TP + FP)$.
- **False Detection Rate (FDR)** FDR is related to precision in that FDR equals one minus precision. I assign the positive label to errors, so the FDR represents the portion of detected errors that are not true errors.
- **Recall** *(Rec)* The ratio of true labels that are returned: $TP/(TP + FN)$. Recall is also known as *sensitivity*.
- **F-Measure** *(FM)* The harmonic mean of precision and recall.

3 Dynamic Binary Analysis to Detect Errors in Small Programs

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 [35]. 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 `while` condition.

[1] Adapted from code originally downloaded from `http://pastie.org/349916`

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

```c
void zunebug(int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear(year)){
      if (days > 366) {
        days -= 366;
        year += 1;
      }
    } else {
      days -= 365;
      year += 1;
    }
  }
  printf("current year is %d\n", year);
}
```

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

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 [17, 19]. 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, when the program executes the error, 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, I hypothesize that combining signals can be used to characterize and distinguish between correct and incorrect behavior for programs.

3.2 Approach

This section outlines my approach for predicting whether a program passes or fails on new test inputs. In broad summary, I collect dynamic execution signals from programs executing on test inputs. Each execution results in a list of numbers (a vector of values) corresponding to the values of the dynamic execution signals for that execution. Each of these vectors of values becomes a single data point, representing the given execution of the corresponding program with the corresponding input. I used off-the-shelf machine learning classifiers and a 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 collect all runtime data for the work in this chapter 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. I 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; I 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; I 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 the call stack traversed, as the stack pointer is usually adjusted downwards each time a routine is called [16]. The minimum stack pointer value can provide a rough idea of the maximum depth of the stack trace, which can also be a proxy for recursion exhibited in the program.
- **Maximum Stack Pointer Value** The maximum value of the stack pointer usually corre-

sponds 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. 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 one says 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 that actually execute. This can be a proxy for control flow behavior across functions within the program. This measure is different from call taken count because call taken count also includes calls to library functions, while this count is restricted to routines within the program.

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 the chosen 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, this 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, my approach to unsupervised learning requires domain-specific assumptions. In this case, I used unsupervised outlier detection to group data points into two sets: typical behavior and outliers. However, to give meaning to the groupings, I 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, I 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, I associated each data point with a label corresponding to whether the test case passed or failed. I 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). I used two sets of labels because I wanted to study how to best characterize the intended and unintended program behavior.

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

Within the single test case data, I 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, I identified that version as potentially failing for that test case. I 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, 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 [72]. 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.

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

For supervised machine learning, I gave a portion of the data points, along with labels indicating if each data point represented a correct or incorrect execution, to a supervised learning algorithm, which built a predictive model.[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. My unsupervised learning technique is built on outlier detection [54]. I start by separating the data for each program by test case. That is, for every program, I consider the data points for each test case across all available versions of the program. Separating the data in this way provides insight into the differences among versions for a single test case.

Within the data relating to a single test case, I 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, I 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 [59], 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 [72]; analyses in fault localization support a similar conclusion [18]. 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.

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

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.

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

[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

	True	True	False	False				
Program	Pos	Neg	Pos	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.

other parameters set to their defaults.[6] 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

I conduct unsupervised learning experiments to discover whether these experiments 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 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. I 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 [18] and the context of potential clients of my technique, such as program repair [72]).

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

[6] The analyses in this chapter use Scikit Learn version 0.15.2.

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 there is labeled data: data for which it is known 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.

3.6 Conclusions

The preliminary work on detecting unintended behaviors in these small programs shows that the basic technique works — the technique of using profiles collected with dynamic binary instrumentation to build machine learning models that predict whether errors occur in profiles of new executions. However, the techniques as presented in this chapter are limited to the small, simple programs. In the following chapters, I demonstrate how to expand these techniques for applicability to more complex systems, specifically to ARS.

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

The previous chapter establishes that the combination of dynamic binary instrumentation and techniques based in machine learning can be useful to detect unusual behavior (or unintended behavior) in software.

This chapter describes the expansion of these techniques to detect errors in the ArduPilot system. While the previous chapter proves the concept on small programs, the technique expansion is interesting because ArduPilot is real software that is in use and of interest to many real users. The ArduPilot software is also vastly more complex than the small programs at issue in Chapter 3. Experiments on ArduPilot demonstrate the technique's applicability to programs that are larger and more complex. To build the models for these experiments, I used supervised machine learning.

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 [103].

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

[1] `http://ardupilot.org`

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

4.2 ArduPilot Approach

For each set of tests, I 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. I execute SCENARIO A a total of 4000 times with dynamic binary execution, 1000 times for each of the categories enumerated in Section 4.2.
- 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. I 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. This 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, I 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, I 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 I provide details relating to how I implement the process described in Section 4.2. I begin by outlining the questions I seek to answer. In Section 4.3.1, I go on to give details on supervised machine learning and how I use it. I have already presented the main program under test and the experimental scenarios I use on it in Section 4.2. I present the 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?

I 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.

I 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

I use out-of-the box machine learning algorithms from Scikit Learn.[3] Specifically, for supervised learning, I use the Decision Tree classifier available from Scikit Learn. Before selecting this classifier, I did an informal survey, analyzing the data with several of the options available from Scikit Learn. I 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, I decided to use the default parameters of the algorithm.

For all supervised machine learning experiments, I 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. I report the arithmetic mean across all folds.

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

[3] `http://scikit-learn.org/` The analyses in this chapter use Scikit Learn version 0.18.2.

4.3.2 Collecting Signals with Dynamic Binary Instrumentation

I use dynamic binary instrumentation to collect low-level information about the processes that execute when I run software. I count events that occur during execution and keep track of minima and maxima for certain values. For example, I 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. I collect the same measurements for each execution, so the summary of each execution can be treated as a feature vector, suitable for input into machine learning algorithms. I aggregate the feature vectors into two-dimensional matrices, representing the overall data set.

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

For these experiments, I 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. I took several approaches to reducing the effects that instrumentation has on timing sensitive executions. Wherever possible, I relaxed timeout parameters in the software. In addition, I worked to reduce the overhead of my instrumentation.

VALGRIND is a powerful tool, able to measure many events at the low level. However, I restrict the events I 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 I could collect data about whether branches are taken and whether they correspond to what VALGRIND considers inverted conditions, I do not collect those data because collecting them would involve more calculations and calls into VALGRIND'S API at runtime.

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

4.4 Results

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

Recall that my 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 I can use supervised machine learning to identify program executions that exhibit a defect, as opposed to executions of the same defect-containing program version that do not exhibit the defect. To evaluate this question, I return to my main program under test: ARDUPILOT.

Results can be found in Table 4.1. Note that my technique can virtually 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. 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 this problem, as posed, may be a problem that is particularly well-adapted to the supervised learning approach. In this task, the algorithm is telling the difference between executions that do and do not exhibit a particular defect, when the data set is restricted to similar executions that do and do not exhibit that same defect. The extremely high accuracy suggests that the technique may be amenable to handling more difficult and complex scenarios.

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

This question evaluates the subgoal of determining whether I can use supervised machine learning to identify program executions of a defective program version as opposed to executions of its repaired counterpart. I 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

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

results come from a constrained experimental setup, their high accuracy metrics suggest that the approach might extend to setups with more noise.

For control, I 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

This question evaluates the subgoal of determining how much data is necessary to make useful predictions. To answer this question, I 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. I 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, 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 I 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, I get reasonably accurate data and that, although accuracy generally increases with additional samples, returns diminish after 500 samples.

4.5 Conclusions

The constrained experiments evaluated in this chapter result in extremely high accuracy, for all accuracy metrics. This suggests that the technique presented here is very well-adapted to detecting execution defects under these constrained circumstances. While these circumstances are artificially constrained (e.g., including training data that exhibits the defect of interest), it may be possible to create data sets in real circumstances that exhibit some of the elements that contribute to the success of these experiments.

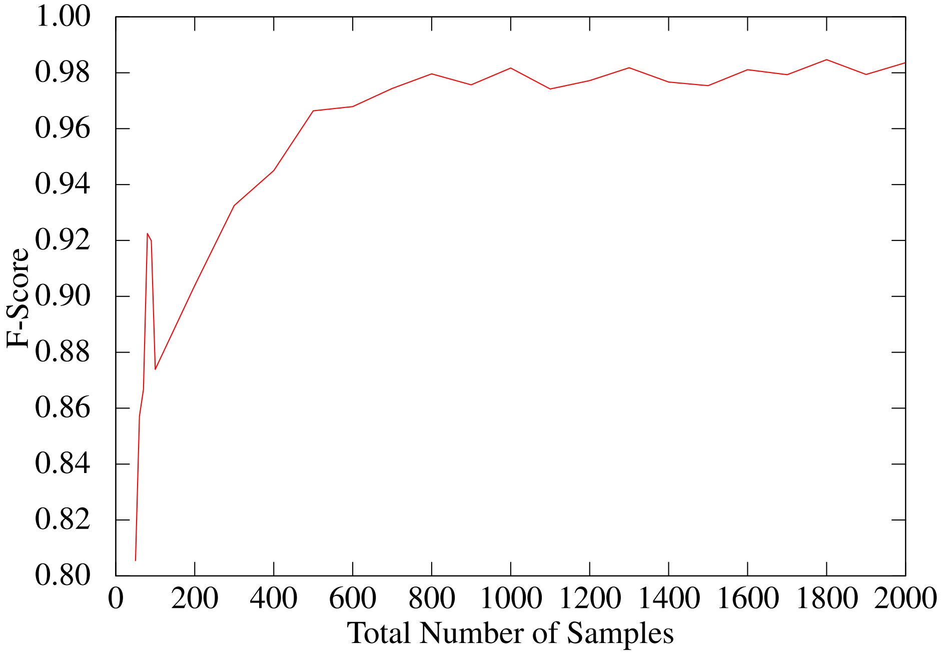

Figure 4.1: Supervised Learning on a Single Defective Version with Varied Amounts of Data

Because these techniques are so highly successful in these limited circumstances, the next step is to apply the techniques to situations in which the input space is more varied.

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

The work in previous chapters demonstrated the viability of dynamic binary instrumentation, used as described, to detect errors in small programs and in the ArduPilot robotics system. To determine whether this insight extends to other robotics systems, I performed similar experiments on additional systems. This chapter describes this work, portions of which were also published in Katz et al. [62].

While the previous chapters used unsupervised and supervised machine learning models, the work in this chapter uses a semi-supervised machine learning approach based on clustering data points representing *nominal* executions. Because this semi-supervised — one class — model uses only the data from nominal executions, the resulting model serves as a representation of nominal behavior. I use techniques based in anomaly detection to determine whether new (unknown) data points fall within the model (and therefore represent nominal behavior) or outside the model (and therefore represent unexpected behavior). Like the work in earlier chapters, this work is based in the assumption that unusual behavior is more likely to exhibit errors than typical behavior and, therefore, that unusual behavior might represent unsafe behavior [35]. In other words, the resulting model serves as a test oracle for data points representing additional unknown executions.

The approach detects a significant portion of faults on two simulated robotics systems. I present initial experiments on these systems in simulation, demonstrating the use and effectiveness of this technique.

This chapter also introduces the idea of using the technique as an oracle for system robustness and safety [10, 13]. Robustness testing involves tests to ensure that a system functions correctly when given invalid or unexpected inputs, stressful environmental conditions, or other conditions [56]. For ARSs, it is vital to ensure that the system operates safely at all points of execution, as they are CPSs that interact with their environment. This chapter demonstrates using the technique to detect potentially unsafe behavior in ARSs during robustness testing.

The success of the work in this chapter provides further support for the idea that systems that interact with the real world are well-suited to execution monitoring because most of the overhead can be absorbed into time the system would have otherwise spent waiting. The absorption of delays in ARS is analyzed in more depth in Chapter 6.

Section 5.1 outlines the method, combining low-level system characterization techniques, clustering, and anomaly detection. Section 5.2 discuses the evaluation of my technique, including the Systems Under Test (SUTs) and test inputs. Section 5.3 discuses experiment results.

Section 5.4 fits this technique into context and explores implications.

5.1 Method

Figure 5.1 illustrates the technique used in this chapter for using anomaly detection to find unexpected behaviors in robotics systems in simulation. This techniques assumes an SUT — a robotics system that can be executed with a system characterization technique, either on real hardware or in simulation [98, 103]. It also assumes at least one *nominal* system input — an input not known to cause any safety violations or unintended behavior. Given this input, the approach is comprised of two steps: setup and testing.

Setup involves running an instrumented system on nominal inputs and collecting summaries of system execution (Section 5.1.1). These summaries are — or are parsable to — a collection of values (either fixed-length or time series). I use the affinity propagation algorithm [42] to automatically cluster these representations, with clusters each corresponding to common modes of nominal operation. Affinity propagation works especially well when the number of modes is not known *a priori*. The nominal data set used in setup is relatively small, so this step does not impact overall performance significantly.

In *testing*, the technique detects anomalies by comparing instrumentation-produced execution summaries of previously unseen inputs against the clusters of nominal behaviors. The comparison uses Local Density Cluster-Based Outlier Factor (LDCOF), a clustering-based anomaly detection technique (Section 5.1.2). LDCOF produces an outlier score for a new input, representing how far the new execution is from the closest cluster of nominal operation. A higher outlier score signifies that a given execution may be indicative of unsafe system behavior.

5.1.1 System Characterization Techniques

This technique uses three system profiling tools: two off-the-shelf tools and one custom VALGRIND tool.

PS Utility

The PYTHON package PSUTIL[1] collects information on running processes by interacting with underlying system services. The resulting data include: User time — the amount of CPU time the system has spent in the process, not the kernel; System time — the amount of time spent in kernel mode; Resident set size — the amount of RAM allocated to the process; and Virtual memory size — the amount of virtual memory the process has access to. I collect this data at a fixed frequency, resulting in a time series describing system behavior over the course of execution.

[1] https://pypi.org/project/psutil/

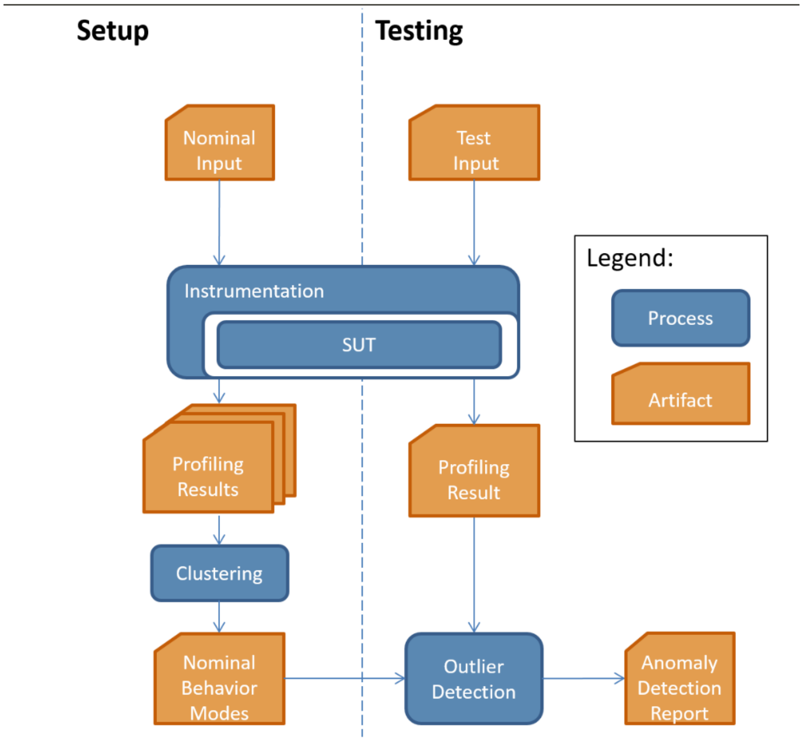

Figure 5.1: The architecture diagram for this testing approach, outlining the setup and testing phases.

Valgrind Memcheck

A popular program profiling method is VALGRIND,[2] an instrumentation framework for developing dynamic analysis tools. VALGRIND's default tool, MEMCHECK, tracks a program's memory accesses, such as memory initialization and freeing resources. It wraps most instructions with instrumentation. I parse MEMCHECK's log file to extract values that summarize the system behavior.

Customized Valgrind Tool: SignalSeer

I designed a custom dynamic binary instrumentation tool on the VALGRIND framework, to collect a broader set of low-level data about the executions of an SUT. I call this tool SIGNALSEER. It counts execution events and keeps track of minima and maxima for certain values, such as: number of machine instructions executed, number of memory stores, and minimum and maximum memory addresses for each. The tool is designed to have low overhead. As such, the choice of what to count is inspired by VALGRIND's instrumentation structure. I add an instrumentation point to each instruction and, at each of these points, only collect data that can be gathered efficiently. The tool minimizes calls to libraries and VALGRIND's API at runtime. This tool provides a summary of execution behavior that is ready to use for the analysis.

5.1.2 Detecting Anomalies in System Execution

To detect execution anomalies, I need a way of measuring how anomalous a given execution is. I use Local Density Cluster-Based Outlier Factor, an anomaly detection algorithm that uses clustering to find outliers [6]. LDCOF works by applying a clustering algorithm to the data, separating the clusters into large and small clusters, then for each data point calculating an outlier score as follows:

$$LDCOF(p) = \min_{C_i \in large\ clusters} \frac{d(p, C_i)}{avg\_dist(C_i)}$$

where $d(p, C_i)$ is the distance between a point p and the center of cluster C_i, and

$$avg\_dist(C_i) = \frac{\sum_{p \in C_i} d(p, C_i)}{|C_i|}$$

.

In this use case, there are separate training and target data sets. Thus, I adapt the LDCOF algorithm to consider *all* clusters of the nominal data to be "large clusters". This ensures that the outlier score is calculated against every mode of nominal operation.

LDCOF suits this use case well because it accounts for variance in operating modes. With LDCOF, a data point that is *X* distance away from a cluster with a lot of variance would be considered less anomalous than one that is the same distance away from a cluster with little variance. This behavior analogizes well to modes of system operation.

[2] `http://valgrind.org`

I use the estimate from $LDCOF(p)$ of how much of an outlier a given data point p (and its corresponding execution) is, as an oracle. Any execution with an $LDCOF(p)$ greater than some threshold is sufficiently far away that it is likely to represent an anomalous execution. For these experiments, I set the threshold equal to one.

5.1.3 Dynamic Time Warping

For system characterization techniques — such as PSUTIL — that produce time series data, I adjust the clustering algorithm to compare these data using Dynamic Time Warping (DTW) [12] instead of Euclidean distance. DTW calculates the disparity between two sequences under warping of the time axis by finding a sequence of index pairs (i, j) that minimizes

$$\sum_{k=1}^{end} |seq_a[i_k] - seq_b[i_k]|$$

subject to the following constraints

$$(i_1, j_1) = (1, 1)$$

$$(i_{k+1}, j_{j+1}) \in \{(i_k + 1, j_k), (i_k, j_k + 1), (i_k + 1, j_k + 1)\}$$
$$(i_{end}, j_{end}) = (len(seq_a), len(seq_b))$$

In prose: it calculates the shortest distance between elements of two sequences, allowing adjustment of the relative ordering of elements between the sequences, as long as the ordering of elements within each sequence remains fixed. DTW is not an actual distance function but instead a distance-like function, which means that one cannot calculate the center point of a cluster using DTW. Instead, I approximate the distance $d(p, C)$ as:

$$d(p, C) = \underset{q \in C_i, q \neq p}{\text{avg}} \; d(p, q)$$

5.2 Experiment Setup

Each experiment centers on one *SUT* (Section 5.2.1). In the *setup* phase, the SUT is repeatedly executed in simulation with a nominal input under the chosen system characterization technique.[3] For the *testing* phase, I generate test inputs by using mutational fuzzing on the nominal input. In these experiments, the nominal input is an input file provided with the SUT. I run the SUT under the system characterization technique with each test input to generate a profiling result and classify the profiling result using my LDCOF-based anomaly detection method. I compare detected anomalies against violations detected by using a set of manually written system invariants, which is a proxy (approximate stand-in) for true safety. Here, an invariant is a rule that must hold over the entire execution, covering any property, such as message transmission frequency or speed limit. Test and invariant construction is described in more detail in Sections 5.2.2 and 5.2.4.

[3] All simulations described in this chapter are run on machines with the following configuration: Intel®Core$^{\text{TM}}$2 Duo CPU (E8500) 2 Cores @ 3.17GHz, 4 GB RAM.

5.2.1 Systems Under Test (SUTs)

These experiments use two SUTs: a research system that includes common robotics libraries and a commercial system. I chose these systems for insight into real-world factory robotics systems and the behavior of some of the underlying libraries that are common in robotics development.

I drew nominal inputs from simulations provided with the robots, as demonstrations of intended system behavior. The inputs take the form of ROSBAGS, files that contain a series of messages of different types that are sent among components of a ROS system during execution. To exercise the system multiple times with the same input, I replay the ROSBAGS on different executions.

I use the GAZEBO simulator for system execution, following the instructions in the SUT documentation. This gives a consistent environment in which to exercise the SUTs while applying the nominal and test inputs.

BenchMark Bot (BMB)

BMB is an artificial system designed by my collaborators at the National Robotics Engineering Center (NREC) for research on the Robot Operating System, common libraries, and interactions. It is a lightweight wrapper for key ROS functionalities, including path planning and switching between sources of control. The robot is designed to compute a path based on waypoints and what it senses in its environment. Because BMB is modular, it can be configured to use any of several planning algorithms.

I tested BMB using a recording of a simple exploration scenario. To do this, I ran one node rather than the entire system. The node I tested was GLOBAL PLANNER, which takes in a point and outputs a likely path.

Fetch Robotics' Fetch

Fetch robots are commercially available "autonomous mobile robots that operate safely in commercial and industrial environments shared by people."[4] They pick up and transport heavy payloads, such as items in a warehouse environment. They have a mobile base and an arm for manipulation. The robots are built on ROS. Fetch Robotics provided simulation and the DISCO DANCE scenario, which I tested. Disco dance demonstrates the movement planners, taking the robot through a range of arm motions, with dummy "collision objects" for the arm to avoid.

5.2.2 Test Inputs

I use mutational fuzzing to create test inputs likely to induce safety failures. Changing values in inputs to exceptional values that are, nevertheless, sometimes seen in real systems (such as MAX_INT, NaN, or -1), has been shown to effectively induce robotics systems failures [56, 68]. Specifically, for each selected message type in a nominal input file, I inject roughly ten exceptional values using a randomized value injection technique. I generate at least six test inputs per message type.

[4] http://fetchrobotics.com/automated-material-transport-v3/

```
if "torso_lift_joint" in joint:
    if abs(joint_vel) > 0.1:
        return True
    if abs(joint_effort) > 450:
        return True
    if joint_pos < 0 or joint_pos > 0.4:
        return True
```

Figure 5.2: Code listing for sample FETCH invariants.

These test inputs give a variety of input patterns, some of which are known to cause safety violations, which provides a data set against which to measure false positives and negatives. I determined whether each input contained faults by using it with the robot in simulation with no instrumentation and evaluating whether the simulation violated any invariants.

5.2.3 Metrics

To evaluate the effectiveness of the anomaly detection based oracle, I find the FDR and sensitivity (a.k.a. recall). FDR is related to precision in that FDR equals one minus precision. I assign the positive label to safety violations, so the FDR represents the portion of detected anomalies that are not safety violations. Lower is better because false discovery results in unnecessary testing effort.

Sensitivity represents the proportion of test cases containing safety violations that were correctly detected. Ideally, this should be one, catching every safety-violating test input. Sensitivity is not influenced by the populations of safety violations in the test inputs, reducing the impact of the choice of testing method and parameters.

While there is no absolute ground truth for safety violations, I evaluate anomaly detection performance by calculating FDR and sensitivity against a proxy for ground truth: a set of invariants, as described in Section 5.2.4. To provide context, I also evaluate a reference oracle detection of crashes via core dump.

5.2.4 Invariants

The evaluation uses an explicitly-written set of invariants as a proxy for ground truth. Invariants are rules which must hold true over the execution of the program. Here, the invariants represent safety constraints: a violation means that the system has become unsafe.

Specifically, the invariants in this experiment are PYTHON functions, human coded from portions of the system documentation. These functions are evaluated over the output log from a test input to determine if the rules were violated. An example of a subset of the invariants in FETCH is in Figure 5.2. This subset of invariants represents restrictions on the allowable values for properties of the TORSO_LIFT_JOINT.

Invariants provide a more in-depth analysis of system behavior and can help find faults beyond those that cause core dumps [56]. Invariants can closely approximate the real world safety

Table 5.1: False Detection Rate ("FDR") and Sensitivity ("Sens.") vs. manually-written invariants using process monitoring ("PS"), Memcheck, and SignalSeer. (Core dumps are also evaluated against invariants, for context. Core dumps FDR is zero by definition.)

		PS		Memcheck		SignalSeer		Core	
System	Scenario	FDR	Sens.	FDR	Sens.	FDR	Sens.	FDR*	Sens.
BMB	Global	0.00	0.44	0.45	0.66	0.00	1.00	0.00	0.44
Fetch	Disco	0.50	0.11	0.25	0.35	0.25	0.50	0.00	0.06

of the system, but they require substantial effort and expertise to create a comprehensive set.

Core Dumps

For additional context, I evaluate a simple core dumps oracle — whether or not an execution crashes and produces a core dump. This very basic restriction on system behavior (that it should not crash) provides a bare minimum of fault detection in testing scenarios. Crash rate is often used as a simple oracle for system safety in robustness testing [28, 47].

5.3 Experiment Results

For evaluation, I compare the technique's detected violations against the ground truth proxy. A positive label indicates a safety violation (detected behavioral anomaly or invariant violation). A negative label means no violation is found (no anomaly is detected or invariant violated, respectively). I calculate accuracy metrics by comparing labels from the technique against the ground truth proxy.

5.3.1 Results

Experiment results are in Table 5.1. Each row represents one of the SUTs. For each system, the table gives FDR and Sensitivity against manually written invariants for each system characterization technique. The table also reports FDR and Sensitivity of core dumps against manually written invariants. The FDR of core dumps against manually written invariants is zero by definition because a check for core dumps is included in the manually written invariants.

False Detection Rate FDR highlights the number of test inputs identified as anomalies that were not actual safety violations. These false alarms can consume testing budgets because they lead to investigation of test inputs that are not actual faults. The false detection rate varied from 0.00 to 0.50, over two systems and three system characterization techniques: up to half of detected anomalies were not safety violations. SignalSeer performed best, with a maximum FDR of 0.25, meaning that 1 in 4 detection is incorrect.

Sensitivity The sensitivity metric captures the proportion of safety violations that were detected as anomalies. Sensitivity was high, reaching 100% when analyzing the behavior of BMB with SIGNALSEER. For Fetch, a commercially available test system, the worst case sensitivity of the anomaly detection oracles (PSMON: 0.11) is double that of the reference oracle (0.06). In the best case, SIGNALSEER gives a nearly 6 fold improvement.

5.3.2 Experiment Discussion

The anomaly detection oracles achieved sensitivity that was as good as or, more often, better than that of the reference oracle. The reference oracle only detects system crashes, but there are many other kinds of potential safety violations, such as the example invariant for Fetch discussed in Section 5.2.4. The higher sensitivity represents the ability to detect safety violations beyond system crashes. However, sensitivity is also far from perfect, never reaching above 50% on Fetch. I believe that one reason is that I used monitoring tools that do not track data values, only execution behavior. It cannot detect a safety violation that only manifests in data, such as a speed limit violation. Given that limitation, I think the results are quite promising. Likewise, I consider that 0.50 and below to be excellent results for FDR given that this technique is highly automated. It does not require the domain-specific knowledge, such as that needed to write invariants.

The effect of using a proxy for system safety Invariants are an imperfect proxy for ground truth system safety. The expressiveness of the invariant checker is limited, and there can be human error in encoding. A larger issue is that it is difficult to ensure a set of invariants is complete. Human operators are very bad at writing accurate and complete invariants, as teams I have worked with have observed in our experience with testing robots and as is noted in the literature [81]. For this reason, I expect the invariants to incorrectly label some executions negative (in which there is a safety violation not found by the invariants).

Because these experiments use invariants as a proxy for real-world safety, it is possible that some False Detections — in which anomaly detection finds an issue not found by the invariants — are actually valid real-world safety violations, in which case the real-world FDR of these oracles would be lower. It is much more difficult to reason about the effect of this uncertain proxy for ground truth on the sensitivity results. It is not known which trials the proxy mislabels, so it is difficult to be sure sure whether they were detected as anomalous. False positives from the proxy translate to uncertainty in sensitivity.

5.4 Discussion

In this section, I discuss several interesting features of the approach and outline threats to the validity of this study.

5.4.1 Monitoring Techniques Can Be Used Together

I suspect that designing a different custom monitoring tool or combining the outputs of different monitoring tools may allow the technique to detect violations the current setup misses. In these experiments, I observed that a portion of the execution anomalies were detected when using one monitoring approach and not others — that is, different techniques detected different bugs. Because the overall technique is general, it is possible to create composite inputs to the anomaly detection algorithm that incorporate the outputs (profiling results) from more than one monitoring technique. Such an approach may lead to improved detection. I also suspect that adding additional low-level elements to be tracked by the customized tool — such as elements that capture data values — may result in finding additional anomalies. For administrative and practical reasons, I am unable to provide further detail here about which specific bugs were detected by which technique.

5.4.2 Manually-Written Invariants are an Imperfect Proxy for Real-World System Safety

While I use explicit invariants taken from system documentation as ground truth, these invariants are far from perfect. In fact, they were even sometimes violated by normal system behavior. For example, one of the designer-provided invariants for BMB specified a minimum transmit frequency of 5Hz, while the code (and associated comments) set the target transmit frequency at 1Hz. In such cases, I chose to modify the invariants because labeling the nominal behavior of the system as faulty would make further analysis difficult.

These conflicts between documented and implemented behavior are not uncommon [81] and are one of the difficulties of creating explicit testing oracles.

5.4.3 Case Study — A 'False Positive" Reveals An Actual Fault

Because invariants represent an imperfect ground truth, it is possible for the technique to detect an anomaly that my analysis erroneously evaluates as a false positive, when the invariants are incapable of identifying the anomaly.

In fact, I found such a case, in BMB, for executions for which anomaly detection with MEMCHECK identified anomalies but there were no invariant violations. By manually examining the mutated inputs corresponding to these executions, I found that each perturbed the /PERCEPTION/MAP field. Manual inspection revealed that improper values in this field may lead to memory corruption, even when they do not cause a core dump. The invariants only detected a problem when there was a core dump. The anomalies detected using the technique with MEMCHECK are genuine faults not found by invariants, even though the analysis labels them as false positives.

5.4.4 Use in Debugging Techniques

One primary area of application for this work is in use with other software testing and debugging tools. Anomaly detection techniques, such as those described here, can serve as an automated oracle for these tools. Automated testing and debugging tools can provide important information to the developer, but they typically require an oracle that describes if a system behaved correctly or not during a test. Using the approach described here increases the amount of automation provided by these tools by removing the need for users to write oracles.

For highly automated tool chains such as RIOT [63] — a testing framework for the robotics and autonomous systems robustness domain — where many testing features are already automated, an automated oracle drastically reduces the amount of user involvement and expertise required.

5.4.5 Clustering to Find Modes of Behavior

This technique to find clusters of nominal behavior relies on affinity propagation, which does not require that you know ahead of time how many modes of behavior there are. As such, it captures all the modes of nominal behavior as nodes in the model. Other models may be more appropriate when the modes of normal behavior are known beforehand. The size and density of these nodes indicate the strength of the cluster. When a new data point is provided, the determination of whether it is an outlier relies not only on how far it is to the nearest cluster but also on the density of that cluster.

Clusters can form on different scales. If you have a program that executes vastly different modes, such as a helicopter mode and a submarine mode, the differences between clusters representing the submarine and the clusters representing the helicopter may obscure any differences of behavior within each of the modes. This speaks to the need for judicious construction of the applicable data set.

5.4.6 Threats to Validity

Certain anomalies may occur in simulation that would not occur on actual robotics system hardware and vice versa. However, testing in simulation is a valid approach to discovering real bugs in autonomy systems [98, 103]. Additionally, the faults detected typically trace to code defects that exist regardless of platform, such as memory faults due to lack of bounds checking or CPU spikes due to busy loops.

In addition, the technique may not generalize beyond the systems I tested, or beyond the context of mutational input testing; the oracles are thus specific to this context and less general than, e.g., explicit invariants. However, the math behind the approach does not depend on this setup and could work with input data generated in different ways, without any requirement for labeling which executions exhibit correct behavior. I also evaluate on more than one system to provide evidence of potential generalizability.

Another threat is that system monitoring may introduce errors, such as timing errors, due to overhead; this would be exacerbated for testing on hardware, as real-world robotics processors have limited capacity. I advocate that the oracles created using this technique be used primarily in testing, rather than deployment. This threat is partially mitigated because I measure core dumps and explicit invariants on uninstrumented systems; thus, any problem in the SUT behavior detected by those techniques cannot be due to instrumentation, and instrumentation-induced failures will manifest as false positives. Finally, I observe that distributed systems that operate in real time, such as the target systems, spend a lot of time waiting. In practice, I have observed that much of the monitoring overhead can often be absorbed into this wait time, with little observable overhead.

5.5 Conclusions

This chapter presented a method that uses anomaly detection to detect potentially unsafe behavior in ARSs, as applied to two robotics systems in simulation. The algorithm uses system monitoring techniques to obtain profiles of executions. It uses a clustering algorithm to create clusters of those executions, representing modes of nominal execution. A distance metric (LDCOF) determines whether additional execution profiles belong to the existing nominal clusters or should be considered anomalies. The method is suitable for identifying faults in robotics and autonomy systems.

Future extensions of this work would involve evaluating the technique on situations in which the initial training data is derived from more diverse executions. These evaluations in this chapter were based on data derived from executions not known to have errors — in this case, data for which no core dumps or invariant violations are detected. In theory and with small modifications, this technique could derive a model from executions that are not all nominal, without necessarily needing labels identifying nominal executions. I would like to try the approach on this kind of data. I would also like to extend the approach to situations in which the input data is derived from more varied execution behavior.

6 Overhead Timing Effects on Autonomous and Robotics Systems

6.1 Introduction

As demonstrated in Chapters 4, and 5, Autonomous and Robotics Systems (ARSs) are amenable to detection of faults by the use of low-level program monitoring. As discussed in those chapters, one primary concern about using these types of monitoring techniques is that the techniques can cause high overhead. CPSs such as ARSs can be sensitive to overhead that interferes with the timing of events — a missed deadline or a sequence of messages received in an unexpected order can cause the system to fail. However, at the same time, these systems are particularly prone to variability in operating conditions because of their interaction with the real world and the unpredictable conditions therein. There are many situations in which the architectures of CPSs can absorb timing delays, when they take place during times when the system would otherwise be spent waiting for physical events or communication from other parts of the system.

I hypothesize that the same properties that allow ARSs and CPSs to absorb timing delays that occur due to real-world unpredictability also allow these systems to absorb some of the delays that would be caused by program monitoring.

I conduct a series of experiments to gain a more precise understanding of the amount and nature of delays that these systems can absorb. To do so, I run experiments in simulation. The nominal executions examine the behavior of an unmodified simulated ARS while the executions with artificial delays examine the behavior of the same systems when message passing is delayed for various topics.

6.2 Experimental Methodology

This section sets out the approach to the experiments in determining the extent to which timing delays interfere with the behavior of ARS. To evaluate the extent to which timing delays deform the observable execution of an ARS, I insert artificial timing delays in a controlled manner in the following experiments:

For a given robot, I establish a set of commands, called a *mission*. For the purposes of these experiments, each mission is represented as a series of destinations in three dimensional space (two dimensional space for robots that move in only two dimensions), with the final destination being a return to the first destination. I create a series of missions within a simulation environment for each robot.

The experiments consist of running two types of executions: *nominal baseline* executions in which the system is run without modifications and *experimental* executions in which the system is run with artificially-inserted timing delays. These two categories of executions are explained in the following subsections.

6.2.1 Nominal Baseline Executions

To establish a *nominal baseline* — a baseline for how a robot behaves under normal conditions, without any artificially-inserted delays — I run each unmodified ARS repeatedly on each of its missions.

The executions to establish a nominal baseline serve several purposes in these experiments. First, the nominal executions establish a baseline for how often the unmodified ARS fails. ARSs often behave in a nondeterministic manner, even in simulation. Factors that can affect the nondeterminism are variations in the order in which messages are received, sensor noise, perception systems' interpretation of the sensor data, autopilot systems, and obstacles in the environment. There can be nominal (unmodified) executions that fail in significant ways, such as failing to reach one or more waypoints; getting "stuck" and discontinuing attempts to follow the mission (e.g., when the perception system cannot determine the robot's location); software crashes; or liveness failures (e.g., hitting a timeout). Because there are failures in the nominal, unmodified system, I cannot simply measure the failure rate of the artificially-modified system. For a realistic measurement of the effect of the modifications, I must measure the extent to which the modified system fails as compared to the extent to which the unmodified nominal system fails. As an approximate metric of these failures, I measure the percent of executions in which the robot never reaches the final waypoint. This metric allows comparison between the failure rate in nominal executions and the failure rate in the modified system.

Second, the nominal executions establish a representative trajectory and other execution characteristics against which the characteristics of modified executions can be compared. Other potential characteristics of interest include the time taken for completion of the mission and the rate at which messages are sent on various topics.

Third, the nominal executions establish the range of variation in nominal trajectories and other execution characteristics. As mentioned above, there is significant nondeterminism in the observed behavior of simulated robots, even when unmodified. Establishing the range in the nominal executions provides a basis to tell when the modified, experimental executions are within the range of nominal behavior or outside of it.

6.2.2 Experimental Executions

For the *experimental* executions, I add controlled artificial delays to the execution of the ARS code. The points within the program at which these delays are inserted, number, and length of these delays are experimental parameters. The method of inserting delays is set out below.

Experimental executions are evaluated against the nominal baseline and against the set of waypoints that they are supposed to reach.

6.2.3 Method of Inserting Delays

This subsection explains how I insert artificial delays for the experimental executions.

ARDUCOPTER

For the ARDUCOPTER experiments, the artificial delays are introduced by modifying source code in C++. I insert a sleep statement before a return statement in the code. To do so, I identified the program point immediately before each return statement in all .CPP files in the ARDUPILOT/ARDUCOPTER source code directory. For each of these program points, there was the possibility of inserting an artificial delay. The choice of whether to insert a delay was determined probabilistically, with a weighted coin flip. Different modified versions of the code were created, [1] each of which had (a) a fixed coin flip weight and (b) fixed delay amount added at each delay location. The weights for the weighted coin flip ranged from 0.1 to 1.0, with 1.0 meaning a delay was inserted before every return statement, and the length of each delay ranged from 0.001953125 seconds to 8 seconds, with delay lengths chosen as powers of 2.

ROS-based Systems

For the ROS experiments, the artificial delays are introduced at communications barriers on ROS topics, taking advantage of the architecture of ROS-based systems.

To give a simplified overview of the architecture of ROS-based systems, these systems consist of various nodes which communicate with each other by sending messages over a bus, as shown in Figure 6.1.

A publish-subscribe system determines which nodes receive which messages. A node can publish messages to a *topic*. To receive those messages, another node subscribes to the same *topic*. Generally, each topic only accepts messages of one type. ROS makes it easy to query a running system to find out information such as (a) the topics in that system; (b) the type of messages published to each topic; (c) the node or nodes that publish to a particular topic; and (d) the node or nodes that subscribe to the particular topic. This information makes it easy to infer certain properties about the relationships among nodes and the purposes of particular messages. I use this information to choose the topics to which I add artificial delays. For example, in HUSKY, I run experiments that delay each topic published by or subscribed to by the /MOVE_BASE/ navigation node. I do this because navigation is a vital function, and I expect disruptions in navigation to have an effect on robot behavior. Conversely, I do not conduct experiments in which I delay the topics related to displaying logging messages, as these are unlikely to affect anything other than the logging messages displayed.

Once I set a TOPIC or TOPICS to delay on a particular ROS system for a particular set of experiments, I insert delays on these topics by intercepting messages using topic renaming. ROS allows configuration of nodes such that topics can be renamed. For example, if Node A is originally designed to publish a topic named /A_VERY_GOOD_TOPIC, I can change the system's configuration so that when published, the topic is known in the namespace as something else, such as /A_VERY_GOOD_TOPIC_INTERCEPTED. Because the topic now has a different name

[1] I used the tool COMBY for these program transformations. https://comby.dev/

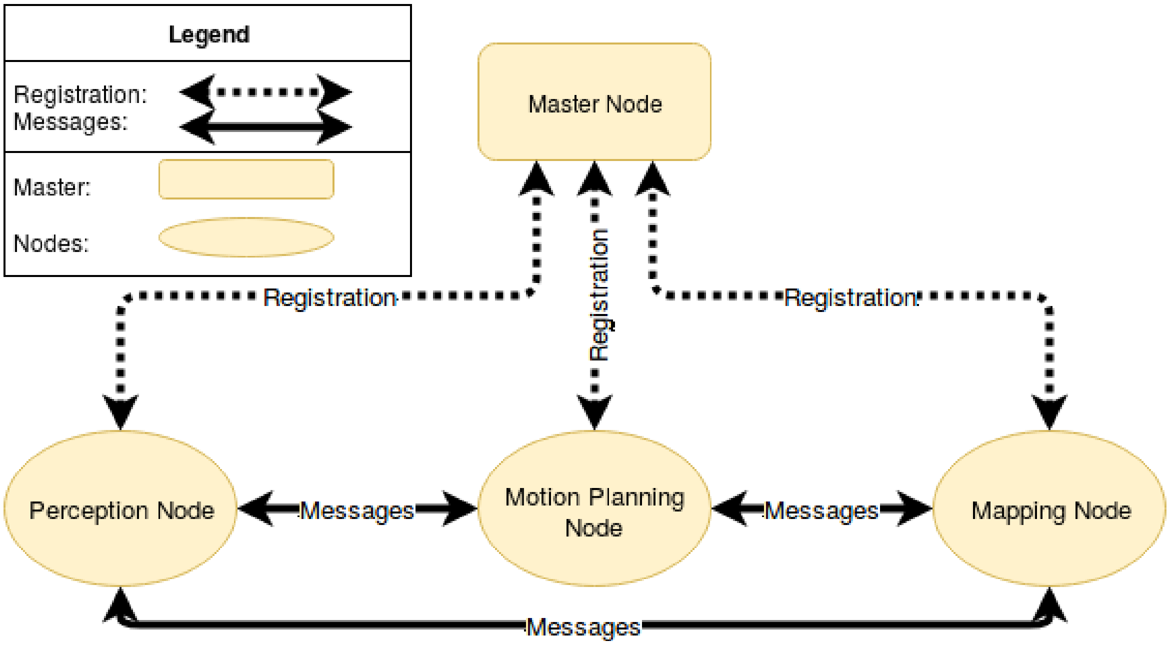

Figure 6.1: Simplified ROS Architecture

than the system expects, the node or nodes that would have originally subscribed to the topic will not receive the messages published on the new topic. However, I wrote an extra node to be included with the ROS system. This node reads each message on a given topic, in this case /A_VERY_GOOD_TOPIC_INTERCEPTED. It then waits for the designated amount of time and then republishes the same message on the topic that was originally expected: /A_VERY_GOOD_TOPIC. The nodes that originally expected the messages on this topic from Node A now receive the same messages from the delay node.

Delays ranged in length from 0.00390625 seconds to 1 second and were inserted for every message in a topic. This range was chosen after a parameter sweep revealed that they result in a representative range of behaviors.

6.2.4 Subject Systems

I evaluate the experiments on the following systems.

- ARDUPILOT: The open-source ARDUPILOT project, written in C++, uses a common framework and collection of libraries to implement a set of general-purpose autopilot systems for use with a variety of vehicles, including, submarines, helicopters, multirotors, and airplanes. ARDUPILOT is extremely popular with hobbyists and professionals. It is installed in over one million vehicles worldwide and used by organizations including NASA, Intel, and Boeing, as well as many institutions of higher-education [103]. These experiments focus on the ARDUCOPTER software, designed for helicopter and multi-rotor aircraft.[2]

[2] https://ardupilot.org/copter/

- HUSKY: The HUSKY unmanned ground vehicle by Clearpath Robotics[3] is a real world robot with an extensive simulation infrastructure. It is rugged, designed to be deployed in uneven terrain, and it is capable of carrying and integrating with a variety of input sources (sensors) and actuators. Husky is popular among researchers for its straightforward design and real world usage history.

6.3 Evaluation

I evaluate the following research questions.

RQ1: To what extent do the presence of timing delays in robot systems have an effect on observable behavior as defined by a set of performance metrics?

RQ2: Are certain kinds of robotics components more robust or resilient to timing delays?

RQ3: Under what circumstances do timing delays lead to system crashes?

6.3.1 Metrics

To evaluate the research questions, I use the following metrics.

Metrics Based on Euclidean Distances These metrics are based on comparing the position in 3-dimensional space between the robot in the deformed execution versus the waypoints to which the robot was instructed to go. Specifically, these include:

- The Euclidean distance between the final position of the robot in the deformed execution and the final waypoint or home point
- The sum of closest distances from each waypoint
- The average of the closest distances from each waypoint
- The greatest closest distance from each waypoint

Completeness metrics These metrics are based on whether the robot completes its mission. They are closely related to RQ3, as discussed in Section 6.3.4.

- Whether the execution navigates to each waypoint and returns home.
- Whether there is a system crash before the route is executed.
- Whether a timeout occurs.

Timeliness metrics

- The amount of time before completion of the execution (either successfully or unsuccessfully)
- Total amount of time taken to reach each waypoint ('reach' defined as when the system issues the instruction to go to the next waypoint)

[3] `https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/`

6.3.2 Effects on Observable Behavior

RQ1: To what extent do the presence of timing delays in robot systems have an effect on observable behavior as defined by a set of performance metrics?

To evaluate RQ1, I look at the metrics enumerated in Section 6.3.1.

The clearest and most obvious effects on observable execution are crashes, both software crashes and crashes in physical space. I evaluate these deviations separately in RQ3 (Section 6.3.4).

HUSKY

Table 6.1 shows, for the nominal and artificially-deformed HUSKY executions, how much their Euclidean distance deviates from the waypoints the robot had been instructed to visit. Data for each mission is listed on its own line. For the purposes of this chart, I look at the trajectories of all experimental runs that reach all of the waypoints for a given mission. I take the robot's minimum distance from each waypoint for each of these experimental executions. I then take the mean, over all of these experimental executions, of the minimum distance for each waypoint. The same information is provided for the nominal executions for comparison. Note that the experimental executions include varying amounts of delay and delays on different ROSTOPICS. I will explore the effects of different delay amounts and delays on different topics in Table 6.3.

Note that in Table 6.1, the mean closest distance to the waypoint is always smaller in the nominal group (which is taken from unmodified executions) than in the experimental group (which is taken from executions with delays). While the closest distance from the destination waypoint usually increases over subsequent waypoints, that is not always the case.

6.3.3 Different Effects on Different Components

RQ2: To evaluate RQ2, are certain kinds of robotics components more robust or resilient to timing delays, I conduct separate experiments in which I insert delays that affect different components. For example, on ROS systems, different experiments have delays on different ROSTOPICS. A ROSTOPIC conveys messages to the nodes that subscribe to it. When I delay the messages on a particular ROSTOPIC, those delays affect the nodes that subscribe to that topic. By delaying different topics separately, I can evaluate the different effects on the subscriber nodes. In ARDUPILOT, I achieve a similar effect by conducting several separate experiments involving modifying only one source code file at a time. This file conceptually corresponds to the component being tested.

Results

Table 6.4 shows the effect of delays on different topics in Husky.

6.3.4 When Delays Cause Software Crashes

RQ3: Under what circumstances do timing delays lead to system crashes?

Table 6.1: Husky: Mean Minimum Euclidean Distance from Waypoints in Meters for Nominal Executions and Experimental Executions

Mission	Distance from Waypoint						WP	WP
	W1	W2	W3	W4	W5	Final	Total	Mean
Nominal								
M1	0.23	0.66	1.65	1.47	2.01	1.10	7.13	1.19
M2	0.08	0.18	0.25	0.15	0.30	0.41	1.37	0.23
M3	0.22	0.38	0.54	0.58	2.85	1.45	6.02	1.00
M4	0.52	1.91	1.78	2.28	1.02	1.92	9.44	1.57
M5	0.39	1.01	1.12	0.57	2.09	2.67	7.86	1.31
M6	0.25	0.31	0.70	0.27	0.76	1.44	3.71	0.62
M7	0.31	0.27	0.28	0.35	0.42	0.55	2.18	0.36
M8	0.28	0.45	0.23	0.98	0.68	0.95	3.57	0.60
M9	0.80	0.53	0.75	0.59	0.83	3.43	6.93	1.16
M10	0.07	0.07	0.08	0.66	1.09	1.91	3.89	0.65
Experimental								
M1	1.00	1.20	3.56	2.92	3.61	1.89	14.19	2.37
M2	0.89	0.52	1.67	1.00	1.40	1.12	6.60	1.10
M3	1.85	2.38	2.25	2.34	5.04	3.48	17.34	2.89
M4	1.42	2.83	3.26	3.06	2.03	2.34	14.94	2.49
M5	1.86	2.01	2.48	1.72	3.56	3.73	15.36	2.56
M6	1.03	1.34	2.67	0.79	2.96	2.05	10.83	1.80
M7	1.61	0.86	1.15	1.73	2.21	1.69	9.24	1.54
M8	1.21	2.85	1.87	3.85	2.70	1.93	14.42	2.40
M9	1.41	0.87	1.32	1.07	1.25	3.10	9.02	1.50
M10	2.57	1.73	0.34	2.39	2.68	4.12	13.82	2.30

To evaluate RQ3, I look at several indicators of software crashes that can be observed from experiments. It is interesting to find out when timing causes a system crash because system crashes have different practical implications for recovery techniques than other failures, such as incorrect trajectories or delays. System crashes can lead to, for example, losing contact with the system or damage to the hardware. Under some circumstances, a system that has crashed without hardware damage can simply be restarted. It is important to separate system crashes from other successful executions so that I can exclude any trajectories and timing data that are invalid because of system crashes.

I establish a baseline of software crashes that occur in the nominal data set. Robotics systems are often nondeterministic and difficult to simulate and, therefore, even nominal executions can experience software crashes. I compare the rate of software crashes in nominal executions against the rate of software crashes under the experimental conditions.

The presence of a core dump file — such as would be produced when a segmentation fault occurs — indicates a system crash The absence of logs that would have normally been produced

Table 6.2: ArduCopter: Mean Minimum Euclidean Distance from Waypoints in Meters for Nominal Executions and Experimental Executions

Mission	Distance from Waypoint						WP	WP
	W1	W2	W3	W4	W5	Final	Total	Mean
Nominal								
M1	52.37	52.37	52.37	52.36	52.36	52.37	314.20	52.37
M2	176.79	176.79	176.78	176.78	176.78	176.78	1060.70	176.78
M3	81.98	81.98	81.98	81.98	81.98	81.98	491.86	81.98
M4	60.56	60.56	60.57	60.57	60.57	60.56	363.39	60.56
M5	63.32	63.32	63.32	63.32	63.32	63.31	379.91	63.32
M6	94.99	94.99	94.99	94.99	94.99	94.99	569.93	94.99
M7	172.64	172.64	172.65	172.65	172.64	172.64	1035.86	172.64
M8	88.61	88.61	88.61	88.61	88.61	88.60	531.64	88.61
M9	51.51	51.51	51.51	51.51	51.51	51.52	309.08	51.51
M10	105.83	105.83	105.83	105.83	105.83	105.83	634.98	105.83
Experimental								
M1	52.37	52.37	52.37	52.36	52.36	52.37	314.20	52.37
M2	176.79	176.79	176.78	176.78	176.78	176.78	1060.70	176.78
M3	81.98	81.98	81.98	81.98	81.98	81.98	491.86	81.98
M4	60.56	60.56	60.57	60.57	60.57	60.56	363.39	60.56
M5	63.32	63.32	63.32	63.32	63.32	63.31	379.91	63.32
M6	94.99	94.99	94.99	94.99	94.99	94.99	569.93	94.99
M7	172.64	172.64	172.65	172.65	172.64	172.64	1035.86	172.64
M8	88.61	88.61	88.61	88.61	88.61	88.60	531.64	88.61
M9	51.51	51.51	51.51	51.51	51.51	51.52	309.08	51.51
M10	105.83	105.83	105.83	105.83	105.83	105.83	634.98	105.83

Table 6.3: Husky: Mean Minimum Euclidean Distance (in Meters) from Waypoints for Varied Delays on Mission 1, Topic /HUSKY_VELOCITY_CONTROLLER/ODOM

Delay	Distance from Waypoint						WP	WP
(s)	W1	W2	W3	W4	W5	Final	Total	Mean
Mean								
0.0	0.23	0.66	1.65	1.47	2.01	1.10	7.13	1.19
0.00390625	4.45	3.59	10.63	8.84	8.70	4.50	40.71	6.79
0.015625	4.44	3.58	10.61	8.83	8.75	4.51	40.72	6.79
0.0625	4.42	3.57	10.55	8.81	8.66	4.47	40.48	6.75
0.25	4.43	3.59	10.57	8.81	8.65	4.44	40.49	6.75
1.0	4.45	3.60	10.65	8.87	8.73	4.47	40.77	6.80
Standard Deviation								
0.0	0.41	1.13	2.62	2.89	3.49	1.80	12.35	2.06
0.00390625	0.86	0.64	1.87	1.65	1.56	0.81	7.39	1.23
0.015625	0.90	0.67	1.88	1.61	1.39	0.77	7.21	1.20
0.0625	0.93	0.71	2.08	1.70	1.67	0.81	7.89	1.32
0.25	0.91	0.66	2.02	1.73	1.68	0.85	7.84	1.31
1.0	0.86	0.63	1.79	1.54	1.47	0.80	7.09	1.18

during a proper execution indicates a system crash. If the test harness exits abnormally, I classify that execution as a system crash.

Results

Table 6.5 shows the percent of Husky executions that either crash or do not reach the end goal (within a tolerance of one meter). For the purposes of these results, I group all executions that do not reach the final goal within the established tolerance for any reason. Here, failure to reach the end goal within the established tolerance is a proxy for the execution having crashed. It is based on the assumption that a system crash will occur before the end of the designated mission and prevent the robot from reaching its goal. There is also an assumption that, if the robot has gone so badly wrong that it does not reach its goal within the established tolerance, it is functionally equivalent to crashing.

6.4 Discussion

This section discusses threats to validity, future directions, and implications of these experiments. While ARS systems are largely able to absorb overhead timing, the effect varies based on the amount of delay and the location of the injected delay.

Table 6.4: Husky: Mean Minimum Euclidean Distance from Waypoints for Varied Topics on Mission 1, Delay 0.25 Seconds

Topic	Distance from Waypoint						WP	WP
	W1	W2	W3	W4	W5	Final	Total	Mean
Mean								
/gazebo/link_states	4.34	3.51	10.34	8.63	8.54	4.37	39.72	6.62
/husky_velocity_controller/cmd_vel	4.41	3.54	10.45	8.71	8.54	4.42	40.06	6.68
/husky_velocity_controller/odom	4.48	3.68	10.80	9.06	8.84	4.48	41.34	6.89
/imu/data	4.44	3.57	10.54	8.78	8.61	4.46	40.41	6.74
/imu/data/bias	4.48	3.61	10.74	8.86	8.69	4.48	40.84	6.81
/navsat/fix	4.44	3.60	10.57	8.82	8.69	4.45	40.58	6.76
Standard Deviation								
/gazebo/link_states	1.09	0.83	2.50	2.14	1.94	1.00	9.51	1.59
/husky_velocity_controller/cmd_vel	0.97	0.76	2.39	1.97	1.94	0.91	8.93	1.49
/husky_velocity_controller/odom	0.80	0.33	1.07	0.86	1.12	0.78	4.96	0.83
/imu/data	0.88	0.69	2.19	1.82	1.79	0.78	8.15	1.36
/imu/data/bias	0.81	0.60	1.51	1.63	1.60	0.78	6.93	1.15
/navsat/fix	0.88	0.60	2.06	1.68	1.59	0.82	7.64	1.27

6.4.1 Threats to Validity

Simulation I run these experiments in simulation. While it is possible to gain many insights about robotics in simulation [98, 103], simulation may not accurately reflect the influence of overhead on timing in real hardware. For example, real robotics hardware often has distributed computing resources which may not be accurately reflected in the centralized computing power available in simulation. A component with less computing power may encounter bottlenecks that are not seen in simulation. Simulation also has imperfect fidelity to real world situations [5].

However, this threat is mitigated by the fact that much of the monitoring and bug detection can also take place in simulation.

Limited Input Data These experiments are limited to the input data provided, which includes relatively simple simulated environments and missions. More complex behaviors may not have been tested.

Limitation to Observed 3-Dimensional Position These experiments did not test effects other than deviations in the observed three-dimensional position of the robot. It is possible that delays can affect other properties. However, this threat is mitigated by the idea that any major failures in robot execution are likely to affect three-dimensional position.

6.4.2 Future Directions

There are several questions that arise directly from the work presented here.

Table 6.5: Husky: Percent of Executions That Crash or Do Not Reach all Waypoints on Mission 1, With a Tolerance of One meter

Topic	Percent Failure by Delay Amount					
	0.0	0.00390625	0.015625	0.0625	0.25	1.00
/gazebo/link_states	60.08	2.50	0.83	2.50	5.00	2.50
/husky_velocity_controller/cmd_vel	60.08	3.33	2.50	1.67	4.17	4.17
/husky_velocity_controller/odom	60.08	2.50	0.83	4.17	0.00	4.17
/imu/data	60.08	0.00	2.50	3.33	3.33	2.50
/imu/data/bias	60.08	1.67	1.67	3.33	1.67	0.83
/navsat/fix	60.08	3.33	1.67	3.33	3.33	0.00

Table 6.6: Husky: Mean Time Taken (seconds) for Executions That Reach (P) and Do Not Reach (F) the Final Waypoint on Mission 1, With a Tolerance of One Meter

Topic	Delays (Seconds)									
(abbreviated)	0.00390625		0.015625		0.0625		0.25		1.0	
	P	F	P	F	P	F	P	F	P	F
/gazebo/link_states	27.45	33.81	28.14	33.62	26.97	33.84	27.91	33.67	27.13	33.69
/husky.../cmd_vel	27.03	33.77	27.82	33.72	27.74	33.58	27.39	33.76	27.35	33.71
/husky.../odom	27.36	33.77	26.49	33.59	27.42	33.65	n/a	33.61	27.23	33.88
/imu/data	n/a	33.67	27.59	33.70	27.60	33.69	27.32	33.85	27.06	33.70
/imu/data/bias	28.07	33.86	27.58	33.66	27.69	33.76	27.63	33.68	28.44	33.58
/navsat/fix	27.11	33.68	28.21	33.63	28.01	33.75	27.67	33.80	n/a	33.70

Violations of Other Desired Properties The work presented here looks at the extent to which artificial timing delays deform execution in robotics programs in simulation by looking at whether the software crashes and physically-observable properties, such as how far the robot is from the expected position in physical space and how long the robot takes to reach waypoints. However, there are other desired properties in robotics execution. For example, there are safety properties that robots should maintain during execution, such as that they should not crash into an obstacle or that they should not violate speed limits. In addition, robots should maintain liveness — they should not time out. It would be interesting to investigate the extent to which timing delays cause these properties to be violated.

Error Handling and Desired Corner Case Behavior It would be further interesting to investigate to what extent timing delays cause robotics systems to enter into error-handling behavior. For example, many systems are designed with *fail safe* behavior, in which the robot is designed to shut down in a non-damaging state when the system encounters an unrecoverable error. Error handling for less severe faults may cause the robot to execute a recovery behavior, such as clearing its position and using its sensors to attempt to identify where it is with respect to its environment. Such a recovery behavior can occur even in nominal execution and is a normal part of providing resiliency and accounting for nondeterminism in normal robotics executions. However, timing delays may cause these behaviors to be more frequent (because the timing delays may cause errors).

Examination of Variation in Nominal Behavior ARSs are noisy. There is considerable variation in their nominal behavior, especially when a perception system, an autopilot system, and obstacles are involved. This leads to considerable variation in paths taken by an unmodified system. The unmodified system sometimes fails to reach all waypoints or simply gets stuck. Additional work should examine the expected amount of variation in unmodified systems and the causes of that variation.

Varied Amounts of Timing Delays The strategy for inserting timing delays in these experiments is relatively simple — a constant delay amount added to every message in the ROS experiments and a constant delay amount added before probabilistically selected return statements in the ARDUPILOT experiments. More targeted delay injections may reveal more precisely the circumstances under which overhead is absorbed versus produces observable behavior deviations.

6.4.3 Discussion of Timing Amounts

Amount of Timing Delays as Compared to Expected Event Frequencies When systems expect events to occur at a given frequency, such as when there is a control loop, a timing delay greater than the given frequency will almost certainly cause unintended behavior. This is reflected in these experiments, as the timing delays were chosen without regard to the various control loop and other expected frequencies in the underlying systems. A portion of the delays are smaller than the various expected frequencies, while a portion of them are larger. Smaller delays, when incurred multiple times in the same program region, can translate into larger delays. There is, however, redundancy and fault tolerance built into many ARSs. A delay greater than an expected event frequency may appear to be absorbed when the redundancy behaviors mask it.

Amount of Timing Delays as Compared to Instrumentation Delays These timing delays are intended to mimic delays caused by instrumentation and monitoring. While the timing delays inserted are not chosen by exact measurement to make them congruent with monitoring delays, they mimic those delays in other ways. The timing delays caused by monitoring are very small and occur very frequently — at every machine instruction. The timing delays inserted in these experiments are generally larger, but they occur less frequently. They are intended as a rough approximation to explain the principle behind why monitoring delays can be absorbed. Extensions of these experiments could be used to designate practical tolerance levels for monitoring and translate those tolerance levels into actual monitoring tools that work within those boundaries.

6.5 Conclusions

As this chapter has demonstrated, timing delays can be absorbed into simulated robotics systems in varying amounts. These experiments support the observations in earlier chapters that overhead caused by dynamic binary instrumentation does not cause as much runtime extension as expected. In addition, because of inherent nondeterminism in the underlying SUTs, the changes in behavior caused by delays are often within expected ranges of behavior under nominal circumstances. If instrumentation can be calibrated to avoid interfering with critical points in the software, it is a suitable tool for analyzing ARSs.

7 Discussion and Conclusion

In summary, I have provided empirical evidence to substantiate the insights of my thesis. I have presented techniques 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 (usual) behavior. I have demonstrated the efficacy of these techniques on small programs, on the ARDUPILOT drone autopilot system, and on several ARSs based on the Robot Operating System and have presented analysis of the prediction accuracy, instrumentation intrusiveness, and calculation efficiency. I have also demonstrated that the overhead generated by the monitoring techniques can be absorbed into system time, under certain circumstances that are likely to occur with ARSs. I have further demonstrated this principle by injecting artificial timing delays into simulated ARSs and observing the effects on the systems.

7.1 Limitations

This section outlines various limitations of the work presented in this document.

7.1.1 Assumption that Unusual Behavior is Bad Behavior

The techniques in this work are based on the assumption that unusual behavior in software is unwanted behavior. While this assumption is well-supported in the literature [35], there are additional reasons why unusual behavior might occur, such as in error-handling cases or cases that handle unusual inputs. Unusual inputs can be anything that is not well-represented in the test input space. Inputs that are unlike test inputs are likely in ARS, given their varied and potentially infinite input space. There are many potential approaches to mitigating this threat, some of which I discuss more extensively in Section 7.3.4.

7.1.2 Limitations on Inputs

The potential input space for CPSs and ARSs is infinite. Any testing-based approach to assuring software quality in these systems is naturally limited to the inputs tested. The techniques presented here can be applied to the systems running on any inputs or environmental conditions. However, the conclusions from the experiments presented here are limited by the range of inputs and environmental conditions represented in the experiments. This is a limitation common to much QA work on systems with large input spaces. However, because there are such dramatic gaps in QA in ARSs, even limited improvements on QA techniques can improve software quality significantly. Companies that are developing autonomous vehicles and other ARSs often make a point of generating as much and as varied input data as they can for their testing processes [82]. A further discussion of the implications of the input space on the quality of the results is in Section 7.3.1.

7.1.3 Execution in Simulation

I run these experiments in simulation. While it is possible to gain many insights about robotics in simulation [98, 103], simulation may not accurately reflect the ARS's behavior in real hardware. For example, real robotics hardware often has distributed computing resources which may not be accurately reflected in the centralized computing power available in simulation. A component with less computing power may encounter bottlenecks that are not seen in simulation. Simulation also has imperfect fidelity to real world situations [5].

However, this threat is mitigated by the fact that much of the monitoring and bug detection can also take place in simulation. A further discussion of the implications of use of these techniques at various points in the development and deployment process can be found in Section 7.3.2.

7.1.4 Limitation to Software Faults

These techniques are limited to software faults. Many faults in ARS can be characterized as faults in requirements or in the interaction of hardware and software [74]. These faults will only be detected by the techniques presented here if they show up in unusual software behavior.

7.2 Future Directions

There are many possible ways in which to build off the work included in this dissertation.

Intrusiveness Reduction One possible future direction is work to reduce the intrusiveness of instrumentation to monitor program behavior. Recall that *intrusiveness* refers to the extent to which execution is perturbed by instrumentation.

In Chapters 3, 4, and 5, I demonstrated the benefits of instrumentation for detecting unexpected behavior in various programs, including ARSs. In Chapter 6, I demonstrated that autonomous and robotics systems, in particular, can absorb many of the kinds of delays that instrumentation overhead may cause. Further investigation may reveal techniques for tailoring instrumentation to incur overhead only in situations in which it can be readily absorbed.

Analysis of Smaller Units in Robotics Software An additional technique for intrusiveness reduction is analysis of smaller units in robotics software. Most software in ARSs 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 work in Chapter 6 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 I take summary data over the entire system, larger trends in system behavior may hide the smaller variations in the one node.

In addition, a focus on smaller program units may facilitate data analysis to occur while the overall program is still running, allowing for flaw detection in an active program. One possible drawback of restricting focus to a smaller program unit is that it may miss systematic behavior. However, this risk is similar for any decision about which data to measure and analyze — there is always a risk that the phenomena of interest will not be included.

A sub-problem of this approach is an attempt to identify units in robotics software in a black-box manner, from the binary machine code associated with the software. This would present several challenges but would have the benefit of being a realistic technique to apply to systems for which source code is lacking.

Strategic Selection of Which Data to Collect to Maximize Hiding Overhead An additional approach to reducing intrusiveness could involve a more strategic approach to the choice of data to collect. My experiments in using dynamic binary instrumentation to collect information about programs have focused on data collection approaches that add the least overhead to programs. This has influenced the selection of which data to collect, such as which signals to monitor and to take summaries rather than keep more detailed traces. However, as I analyze in Chapter 6, certain robotics and cyber-physical systems can absorb overhead in certain circumstances. It would be useful to determine if collecting different data can provide additional information and to evaluate which data can be collected while staying within the bounds of the overhead that the system can absorb. An approach to evaluate this may include a set of controlled experiments to determine which data can be collected without deforming execution performance to an unacceptable degree. These data can be used in experiments involving information gain to assess their usefulness in determining whether programs are behaving in an unexpected manner.

Sampling Another approach to reducing intrusiveness could involve sampling. The instrumentation approaches discussed in Chapters 3, 4, and 5 involve keeping count of all occurrences of each of the kinds of program events they are tracking. An alternate approach could decrease overhead by taking measurements at fewer instrumentation points. The approach to choosing these instrumentation points may be combined with the work in Chapter 6 on determining when adding overhead deforms observable execution. Work on this approach would involve evaluating the trade offs between accuracy of predictions and reducing intrusiveness by sampling.

Diversity of Input Data Future extensions of the techniques in this work could involve evaluating the techniques on situations in which the data is derived from more diverse executions. The possible input space for ARS is potentially-infinite. More creativity could be applied to generating interesting inputs for these systems, potentially building on work taking place extensively in the community that is developing autonomous vehicles for use as driverless cars. The use of more diverse inputs could lead to more nuanced fault identification.

7.3 General Discussion

This section draws out some of the implications of the techniques and results presented in the previous chapters.

7.3.1 The Impact of Data Scope on Experimental Approach and Accuracy

For each set of experiments in Chapters 3, 4, and 5, I used a set of data that was appropriate to the nature of the programs under test and the test infrastructure available for those programs, building off of others' use of those programs as Systems Under Test (SUTs). I will compare and contrast the data and approaches used in each set of experiments for the purposes of highlighting the differences in the experiments, especially where the differences impact the accuracy of the experiments.

All of these experiments relied on the same underlying techniques: collecting information about individual executions and using that information to build models based in ML techniques. The models, in turn, provide predictions — determinations about whether individual executions are behaving as intended. However, the specific instantiations of these components varied in the three sets of experiments.

Small Programs In Chapter 3, the proof of concept experiments involved small programs — the SIR Siemens data set that have been commonly used in testing research [55]. There are seven base programs, drawn from real software. Engineers seeded bugs into each of these base programs, creating various buggy versions, each with one bug. They also created a set of test cases with the goal of maximizing coverage. The test cases for each program are numerous and have much duplicate coverage and functionality. The large set of test cases led to an unbalanced data set – many more executions result in intended behavior than unintended behavior. There was a small, finite, and known number of errors in each program. Each version of each program only had one error. This created a data set in which supervised learning was primed to do very well well. Even after balancing the data so that the supervised algorithm would not trivially choose the more common class (intended behavior), there were many examples of intended functionality. Even better, each error was distinct and was likely to be represented in the training set, despite using standard 10-fold cross validation techniques. Because each type of error was usually represented in the training set, the system could benefit from overfitting to its data set.

However, it may be possible to construct a data set whose characteristics mimic some of the characteristics that made this data set so amenable to the application of supervised learning. Doing so would involve applying any domain knowledge known about the underlying system to generate a test suite that is rich in data representing nominal behaviors. The more extensive the representations of as many nominal behaviors as possible in the test set, the more likely the model created will correctly identify those nominal behaviors. If using supervised learning, the models will likely be best at identifying unintended behavior modes that are similar to those represented in the data set as examples of unintended behavior. Therefore, the test designers should use whatever domain knowledge they have about the system to represent all known error modes of interest in the training data. The system is designed to detect unknown error modes, but it will be best at those that resemble those error modes in the training set.

For unsupervised learning on the SIR programs, I used domain knowledge to tailor the unsupervised learning algorithm. While the techniques are generally black-box, it is advisable to use domain knowledge to advantage when it is available.

ARDUPILOT In Chapter 4, the ARDUPILOT experiments focused on a set of scenarios exercised in ARDUCOPTER. In each scenario, there was a single known defect. The experiments focused on whether the given defect was activated in the given scenario. The models created relied on the knowledge of which experiments belonged to which scenario. Because each model was constrained to a single scenario with a single defect, the accuracy of the models was extremely high. Each model may have simply looked for whether the markers of that particular defect were activated. This reflects the fact that, if you can constrain the domain of your model to a comparatively small world, you may be able to increase accuracy.

Varied ROS Systems In Chapter 5, the techniques extended to other ROS systems, I created the data corpus in a different manner. I established a set of nominal data modes by exercising the system with inputs not known to activate faults. I created a one-class model from these nominal executions. This one-class model allows for many different modes of behavior in the nominal data, which is a different approach from the experiments in the previous two chapters. However, it is similar to the experiments in the ARDUPILOT chapter because all of the nominal data did derive from the same basic set of instructions. The experiments in Chapter 5 are broader in potential applicability, though. They detect not one form of defect but many. Not a finite set of known defects but a set of potentially unknown defects. They allow for many different modes of behavior in the nominal data and can detect many different modes of deviance in the off-nominal data. As such, this is a harder problem, and the accuracy rates are lower than in the previous chapters. However, it requires less domain knowledge to be incorporated into the data set construction and setup. It is applicable to situations in which domain knowledge is not available and is highly automatable.

The contrasts among these sets of experiments suggest factors to consider when applying the techniques presented in this dissertation to a new system or situation. If the test designer has domain knowledge, it might be desirable to encode it into the setup, leveraging it to increase accuracy, reduce the amount of training data needed, increase model efficiency, or improve other desired properties. This might be doable without significant human effort — translating a small amount of human effort into a drastically more accurate or otherwise better system. However, one must be careful when using domain knowledge. It is easy to encode the tester's biases (e.g., about which types of failures are possible in a given system) into any use of domain knowledge. Another factor to consider is the scope of failures you would like to catch and the diversity of the nominal behavior. It is easiest to create a model to catch one kind of failure from homogeneous data. With increasing complexity comes increasing difficulty. Models derived from these techniques are expected to perform best in the most constrained situations. They still apply to situations in which data is diverse, but there are tradeoffs in reduced accuracy.

7.3.2 Applicability of These Techniques within the Development and Deployment Process

While these techniques can be broadly applied to systems in any phase of development or deployment, they are particularly well-adapted to early phases of system development and when systems are tested in simulation. Systems should be tested early and often. Errors that are discovered and fixed when the software is still under development in simulation are errors that do not need to be discovered later, when the system is operating on real hardware that is more difficult to repair.

When robotics software runs on real hardware in mass deployment, the system has often been engineered to have no more capacity than is strictly necessary to run the system's basic functions. Although there may still be excess capacity in the deployed robots' components, especially during non-critical points, the components will have been designed for less "waste" capacity. Therefore, in deployment, there may be fewer opportunities for hiding monitoring overhead in excess capacity, and there may be more timing-critical points in which overhead from monitoring can deform execution.

Nonetheless, because of the nature of the workloads in robotics systems, components do not face constant and consistent workloads. The variations in the workload will often provide some measure of excess capacity, even when the system is optimized.

However, the techniques presented in Chapters 3, 4, and 5 are particularly well-adapted to being run in simulation. These techniques rely on repeated executions of robotics software under somewhat controlled circumstances. (E.g., by log replay; by holding an environment constant; by providing the same set of waypoint instructions; by providing inputs that have been modified in a particular way.) These data are particularly well-adapted to collection in simulation. Simulation makes it easy to repeatedly run controlled experiments.

While there is nondeterminism in many of the popular simulators used for testing robotics systems, which limits the strict control and reproducibility in simulated testing, the nondeterminism does not prevent simulation from being a useful tool for testing. Repeated executions with the same or similar inputs can cover a range of behaviors. Furthermore, this nondeterminism mirrors some of the nondeterminism found in behaviors of real robotics systems, causing the simulation to be analogous to the real systems in these ways.

There is untapped potential in testing in simulation, as further detailed in Section 7.3.3. It can be used to find software errors before deployment. Robotics systems have several sources of error. One way to characterize these sources is (a) errors that exist exclusively in the software (e.g., using the wrong inequality operator in an IF statement condition); (b) errors that come from the interaction of the software with the hardware or other real world components (e.g., improperly representing the amount of friction in a given environment); and (c) errors that exist solely in the hardware (e.g, a flat tire). Each of these categories is quite large and encompasses many possible problems. Simulation has the potential to find many of the errors that fall into categories a and b [103].

In addition to the applications in simulation, it should also be possible to apply these techniques to systems in the early stages of deployment — before the systems have been optimized to remove excess capacity.

Before deploying monitoring techniques on real hardware, whether in development or in deployment, I would suggest undertaking an analysis of the amount of overhead the system can absorb, starting with the kinds of analysis presented in Chapter 6. The basic principle would be to determine how much overhead the system can tolerate (and where it can tolerate that overhead). Then you can develop a custom monitoring tool that stays roughly within these capacity limits. (Such development may involve trial and error as to targeting the monitoring and controlling the amount of overhead.)

7.3.3 Testing in Simulation is Very Useful — Untapped Potential

Testing robotics systems in simulations has many advantages, but there are also many practical drawbacks that hinder its wider use [5]. Advantages of simulation involve:

- Lack of opportunity to damage any real robotics hardware
- Ability to simulate situations that are impractical to create in real life
- Ability to simulate the same situations repeatedly
- Ability to generate training data from repeated simulations
- Ability to run more executions than would be practical to run in real life

However, the drawbacks include:

- Reality gap — an inability to adequately represent the reality of the robot or its environment
- Complexity — it takes too much time and resources to set up an adequate simulation
- Automation features — the simulator is not designed to be used for automated and repeated testing

As discussed in the previous subsection, the approaches presented here for detecting unusual behavior in robotics systems are perhaps best used in simulation. Because these techniques are specifically directed at finding problems in the software, the reality gap should matter less than it might in other circumstances. It is, however, still important that the simulated robot operate in a setting that reasonably simulates the environments in which it will be operating — it otherwise might not activate the relevant code paths and behaviors.

For all these reasons, I advocate for the improvement of ARS simulation technology so that it can be readily used to achieve reasonable simulations of these systems before deployment. Specifically, I advocate for the development of sufficient simulation technology to analyze the software and its failures. Accurate and accessible simulation, while difficult to achieve, would be of significant use in developing accurate, efficient, and safe software for ARS. Even when simulation is imperfect, a reasonably accurate simulation can serve as a starting point for an iterative process involving simulation and real-world testing. Given a sufficiently feature-rich simulation platform, simulations can be improved based on observation of the ARS's behavior in the real world. This would, in turn, allow more nuanced fault detection in simulation.

7.3.4 Separating Intended but Unusual Behavior from Unintended Behavior

This work relies on the assumption that intended behaviors are more common than unintended behaviors, so that unusual behaviors can be identified as bugs. While this assumption is well-supported in the literature [35], it does not cover every case. Even in simple software, there is often code that handles unusual inputs. For example, a major bug in Microsoft's Zune music players occurred in code that was intended to handle unusual but expected situations — leap days. In a simplistic sampling of all program behaviors, the mere fact that the code entered this unusual code path could cause the code to be flagged as an unusual situation and potentially an error. However, the error was not in the mere fact that the player executed unusual code — the error was in a computation in that code path that caused an infinite loop [2]. In many complex systems, there are even more edge and corner cases that might activate intended but unusual behaviors, and this handling behavior might be very complex. So, merely identifying all unusual behaviors will result in false positives.

There are several possible approaches to separate unusual but intended behavior from unintended behavior. Each approach involves trade-offs.

In the worst case, one could rely on humans to tell the difference between intended unusual behavior and unintended behavior. For example, every time the system detects an unusual behavior, the system shows the corresponding execution information to a human and asks the human if it represents a real problem.

This human-reliant approach has several drawbacks. One is that human time and expertise is expensive. It may require a great deal of time for the human to learn enough about the system to be able to tell the difference between defects and false alarms. Another drawback is that humans can get it wrong. Humans tend to find these kinds of tasks difficult. By analogy, I saw in my experiments that humans erred in writing invariants to represent the correct behavior of a system (see Section 5.4). And, in fact, it is well-documented that it is difficult for humans to write invariants [81]. A human asked to figure out if an alert represents a real bug may very well suffer from similar difficulties. Another issue is that the system will have to generate all the relevant information for the human to interpret this problem. The information that a human would find most relevant to determining whether there is a problem may not be easily accessible. For example, the techniques described in this dissertation are applicable to software that does not include source code, but humans may want source code access to determine whether an alert corresponds to an actual bug. Also, humans are likely to think that such a task is a waste of their time and fail to see value in the underlying tool that requires such supervision.

Alternatives to humans each involve different tradeoffs. Many of these possible alternatives involve shaping the data used to create the relevant models.

One data-driven approach involves using coverage metrics to generate inputs that explore all code paths in the underlying software. (Or, for black box techniques without source code, this is more properly stated as exploring all paths and portions of the machine code.) The inputs can be used to generate executions that will populate the machine learning models. The idea is that this approach is more likely to capture the intended edge and corner case behavior as input data and, therefore, is less likely to flag it as unintended behavior, However, a drawback is that this approach is that it can inflate the perceived frequency of unusual code paths and diminish the perceived frequency of usual ones. This can make an unintended code path seem artificially more frequent and, therefore, intended. Another drawback is that it pushes the responsibility to the test suite designer. Test suite design is known to be a difficult problem [40]. Although there has been much research into designing good test suites, it is possible that a lot of effort will go into creating a test suite, with a comparatively small corresponding benefit.

Using data in a different way, it would be possible to use the inputs that generate unusual behavior as starting points to generate other inputs. Observation of the behaviors of the program in the input spaces near the inputs that produced the unusual behavior can allow more sophisticated conclusions about what factors influence the unusual behavior. As an example, I can use a technique analogous to that presented in my work on Robustness Inside Out Testing [63]. In that work, we use a four step process to expand an input associated with a single detected (unit-level) error. As a part of that process, we use optimization techniques to build a set of user-understandable rules that describes the input space that triggers the error. I could use a similar kind of drilling down on inputs identified as unusual to create explainable rules about the circumstances under which similar executions arise. These rules can, for example, be presented to humans as a more understandable input space for them to determine whether or not it represents unintended behavior. Or this can be used as an input to other computational techniques.

7.3.5 Clustering with Nondeterminism in Nominal Data

I have mentioned several times that behaviors in robotics systems are nondeterministic. This means that certain input scenarios will result in anomalies sometimes and not other times. The approaches to anomaly detection presented in this dissertation focus on individual executions, so they identify the anomalies in a particular run with a given input. For example, it might say that the 5th, 28th, and 37th time that the system ran with a given input, it encountered an anomaly. I can find anomalies in executions run with so-called nominal inputs because I know that robots do not always behave as intended even with inputs that should not (and do not usually) cause them trouble.

One implication of nondeterminism is in how the techniques establish nominal data clusters. I have spoken about establishing nominal data clusters by running the system with nominal inputs which are not known to activate any defects. However, if nominal inputs can sometimes activate defects, the nominal models may incorporate those defects. This means that these techniques will not necessarily identify the faulty behavior in these executions as faulty — the technique establishes them as part of the nominal model.

I could take several approaches to mitigating this potential problem of incorporating the nondeterministic errors of the nominal experiments into the model. The first is already built into the clustering approach. The nondeterministic errors are less frequent compared to the actual nominal behavior when running with the nominal inputs. They will likely represent a smaller cluster or clusters of behavior, making it less likely that many other (test) behaviors will be considered as close enough to belong to that cluster, because the distance metric for belonging to a cluster includes adjustment for size and density of the cluster. Second, I could try to filter out data points that represent errors from the initial data set, before I use it to build a model. This is a common technique and involves running outlier detection on the original nominal data set before using it as nominal data to create a model [54]. However, this approach has some tradeoffs as well. The data points identified as outliers may actually represent recovery behavior or other desired behavioral modes that I would want to incorporate into the models. This approach relies on the principle that desired behaviors occur more frequently than undesired behaviors and that occurrences of undesired behaviors differ from each other, so that they will not create large clusters. (By analogy to the principle of “All happy families resemble one another; every unhappy family is unhappy in its own way” [104].)

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, 7.3.4

[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] Yasasa Abeysirigoonawardena, Florian Shkurti, and Gregory Dudek. Generating adversarial driving scenarios in high-fidelity simulators. In *International Conference on Robotics and Automation*, ICRA '19, pages 8271–8277, 2019. 2.1

[5] Afsoon Afzal, Deborah S. Katz, Claire Le Goues, and Christopher S. Timperley. A study on the challenges of using robotics simulators for testing. *arXiv preprint arXiv:2004.07368*, 2020. 2.1, 6.4.1, 7.1.3, 7.3.3

[6] 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, 2.1, 2.2.2, 5.1.2

[7] 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

[8] 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

[9] S. S. Banerjee, S. Jha, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer. Hands off the wheel in autonomous vehicles?: A systems perspective on over a million miles of field data. In *Dependable Systems and Networks*, DSN '18, pages 586–597, 2018. 1

[10] 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, 5

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

[12] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time

series. In *Knowledge Discovery and Data Mining Workshop*, KDD Workshop '94, pages 359–370, 1994. 5.1.3

[13] Antonia Bertolino. Software testing research and practice. In *Abstract State Machines*, ASM '03, pages 1–21, 2003. 5

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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] Shitao Chen, Yu Chen, Songyi Zhang, and Nanning Zheng. A novel integrated simulation and testing platform for self-driving cars with hardware in the loop. *IEEE Transactions on Intelligent Vehicles*, 4(3):425–436, 2019. 2.1

[22] Matthew Clark, Xenofon Koutsoukos, Ratnesh Kumar, Insup Lee, George Pappas, Lee Pike, Joseph Porter, and Oleg Sokolsky. Study on run time assurance for complex cyber physical systems. Technical Report ADA585474, Air Force Research Lab, April 2013. Available at `https://leepike.github.io/pubs/RTA-CPS.pdf`. 2.1

[23] E. Coelingh, J. Nilsson, and J. Buffum. Driving tests for self-driving cars. *IEEE Spectrum*, 55(3):40–45, 2018. 2.1

[24] 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

[25] 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

[26] 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

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

[28] John DeVale and Philip J. Koopman. Robust software – no more excuses. In *Dependable Systems and Networks*, DSN '02, pages 145–154, 2002. 5.2.4

[29] 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

[30] 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

[31] Rémi Domingues, Maurizio Filippone, Pietro Michiardi, and Jihane Zouaoui. A comparative evaluation of outlier detection algorithms: Experiments and analyses. *Pattern Recognition*, 74:406–421, 2018. 2.1

[32] Tong Duy Son, Ajinkya Bhave, and Herman Van der Auweraer. Simulation-based testing framework for autonomous driving development. In *International Conference on Mechatronics (ICM)*, ICM '19, pages 576–583, 2019. 2.1

[33] 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

[34] 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

[35] 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, 5, 7.1.1, 7.3.4

[36] 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

[37] 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, Ernst-Daikon2001, LorenzoliBehavioral2008, HangalIodine2005, BeschastnikhTemporal2011, RatcliffInvariants2011, ErnstDaikon200569:35–45, 2007. 2.1, 2.1

[38] 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, 2.1, 2.1

[39] 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

[40] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013. 7.3.4

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

[42] Brendan J. Frey and Delbert Dueck. Clustering by passing messages between data points. *Science*, 315(5814):972–976, 2007. 2.1, 2.1, 5.1

[43] 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

[44] Alessio Gambi, Tri Huynh, and Gordon Fraser. Generating effective test cases for self-driving cars from police reports. In *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering*, ESEC/FSE '19, pages 257–267, 2019. 2.1

[45] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *International Symposium on Software Testing and Analysis*, ISSTA '19, pages 318–328, 2019. 2.1

[46] 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

[47] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of Linux kernel behavior under errors. In *Dependable Systems and Networks*, DSN '03, pages 459–468, 2003. 5.2.4

[48] Mohammad Hamad, Zain A. H. Hammadeh, Selma Saidi, Vassilis Prevelakis, and Rolf Ernst. Prediction of abnormal temporal behavior in real-time systems. In *Symposium on Applied Computing*, SAC 18, pages 359–367, 2018. 2.1, 2.1

[49] 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

[50] 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

[51] 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

[52] 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

[53] 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

[54] Victoria J. Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004. 2.1, 3.3, 7.3.5

[55] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *International Conference on Software Engineering*, ICSE '94, pages 191–200, 1994. 7.3.1

[56] Casidhe Hutchison, Milda Zizyte, Patrick E. Lanigan, David Guttendorf, Michael Wagner, 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, 1.3, 2.1, 5, 5.2.2, 5.2.4

[57] 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-solut` `html`. 2.1

[58] 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

[59] 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

[60] Hengle Jiang, Sebastian Elbaum, and Carrick Detweiler. Inferring and monitoring invariants in robotic systems. *Autonomous Robots*, 41(4):1027–1046, 2017. 2.1

[61] 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

[62] Deborah S. Katz, Casidhe Hutchison, Milda Zizyte, and Claire Le Goues. Detecting execution anomalies as an oracle for autonomy software robustness. In *International Conference on Robotics and Automation*, ICRA '20, pages 9367–9373, 2020. 5

[63] Deborah S. Katz, Milda Zizyte, Casidhe Hutchison, David Guttendorf, Patrick E. Lanigan, Eric Sample, Philip Koopman, Michael Wagner, and Claire Le Goues. Robustness inside out testing. In *Dependable Systems and Networks – Industry Track*, DSN-I, page to appear, 2020. 2.1, 5.4.4, 7.3.4

[64] 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

[65] 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

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

[67] 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

[68] Philip Koopman, Kobey Devale, and John Devale. *Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project*, chapter 11, pages 201–226. 2008. 5.2.2

[69] 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

[70] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *Foundations of Software Engineering*, FSE '14, pages 178–189, 2014. 2.1

[71] Tien-Duy B. Le and David Lo. Deep specification mining. In *International Symposium on Software Testing and Analysis*, ISSTA '18, pages 106–117, 2018. 2.1

[72] 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

[73] 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

[74] Nancy Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. Engineering Systems. MIT Press, 2011. ISBN 9780262016629. URL `https://mitpress.mit.edu/books/engineering-safer-world`. 7.1.4

[75] 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

[76] 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

[77] 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

[78] 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

[79] 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

[80] 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. 1.3, 2.2.1, 2.2.1

[81] Robyn R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Requirements Engineering*, RE '93, pages 126–133, 1993. 5.3.2, 5.4.2, 7.3.4

[82] Alexis C. Madrigal. Inside Waymo's secret world for training self-driving cars. *The Atlantic*, August 2017. 2.1, 7.1.2

[83] 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

[84] 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

[85] 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. 1.3

[86] 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

[87] 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

[88] 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

[89] 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 *Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, 2009. 2.1, 2.1

[90] Mauro Pezze and Cheng Zhang. Automated test oracles: A survey. *Advances in Comput-*

ers, 95:1–48, 2014. 2.1

[91] Zachary Pezzementi, Trenton Tabor, Samuel Yim, Jonathan K. Chang, Bill Drozd, David Guttendorf, Michael Wagner, and Philip Koopman. Putting image manipulations in context: Robustness testing for safe perception. In *Safety, Security, and Rescue Robotics (SSRR)*, pages 1–8, 2018. 2.1

[92] 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

[93] Aakarsh Rao, Nadir Carreón, Roman Lysecky, and Jerzy Rozenblit. Probabilistic threat detection for risk management in cyber-physical medical systems. *IEEE Software*, 35(1): 38–43, 2018. 2.1

[94] 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

[95] 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

[96] 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

[97] 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

[98] 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, 5.1, 5.4.6, 6.4.1, 7.1.3

[99] 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

[100] 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

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

[102] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *International Conference on Software Engineering*, ICSE '18, pages 303–314, 2018. 2.1

[103] 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, 5.1, 5.4.6, 6.2.4, 6.4.1, 7.1.3, 7.3.2

[104] Leo Tolstoy. *Anna Karenina*. T. Crowell & Company, 1899. 7.3.5

[105] 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

[106] 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

[107] 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

[108] Dohyeon Yeo, Gwangbin Kim, and Seungjun Kim. Toward immersive self-driving simulations: Reports from a user study across six platforms. In *Conference on Human Factors in Computing Systems*, CHI 20, pages 1–12, 2020. 2.1

[109] Miao Yu, Basel Halak, and Mark Zwolinski. Using hardware performance counters to detect control hijacking attacks. In *International Verification and Security Workshop*, IVSW '19, pages 1–6, 2019. 2.1

[110] 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

[111] 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

[112] X. Zheng, C. Julien, R. Podorozhny, F. Cassez, and T. Rakotoarivelo. Efficient and scalable runtime monitoring for cyberphysical system. *IEEE Systems Journal*, 12(2):1667–1678, 2018. 2.1

[113] 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

[114] 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

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