Robustness Testing of Autonomy Software

Casidhe Hutchison
National Robotics Engineering Center
Carnegie Mellon University
fhutchin@nrec.ri.cmu.edu

Milda Zizyte
Dept. of Electrical & Computer Engineering
Carnegie Mellon University
milda@cmu.edu

Patrick E. Lanigan
National Robotics Engineering Center
Carnegie Mellon University
planigan@nrec.ri.cmu.edu

David Guttendorf
National Robotics Engineering Center
Carnegie Mellon University
dguttendorf@nrec.ri.cmu.edu

Michael Wagner
National Robotics Engineering Center
Carnegie Mellon University
mwagner@cmu.edu

Claire Le Goues
School of Computer Science
Carnegie Mellon University
clegoues@cs.cmu.edu

Philip Koopman
Dept. of Electrical & Computer Engineering
Carnegie Mellon University
koopman@cmu.edu

ABSTRACT

As robotic and autonomy systems become progressively more present in industrial and human-interactive applications, it is increasingly critical for them to behave safely in the presence of unexpected inputs. While robustness testing for traditional software systems is long-studied, robustness testing for autonomy systems is relatively uncharted territory. In our role as engineers, testers, and researchers we have observed that autonomy systems are importantly different from traditional systems, requiring novel approaches to effectively test them. We present Automated Stress Testing for Autonomy Architectures (ASTAA), a system that effectively, automatically robustness tests autonomy systems by building on classic principles, with important innovations to support this new domain. Over five years, we have used ASTAA to test 17 real-world autonomy systems, robots, and robotics-oriented libraries, across commercial and academic applications, discovering hundreds of bugs. We outline the ASTAA approach and analyze more than 150 bugs we found in real systems. We discuss what we discovered about testing autonomy systems, specifically focusing on how doing so differs from and is similar to traditional software robustness testing and other high-level lessons.

KEYWORDS

dependability, robustness testing, safety-critical systems, autonomy

ICSE-SEIP '18, May 27-June 3 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183534

ACM Reference Format:
Casidhe Hutchison, Milda Zizyte, Patrick E. Lanigan, David Guttendorf, Michael Wagner, Claire Le Goues, and Philip Koopman. 2018. Robustness Testing of Autonomy Software. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27-June 3 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3183519.3183534

1 INTRODUCTION

Autonomy and robotics systems are rapidly gaining importance in modern times. From self-driving cars to factory and surgical robots, autonomy systems are transforming the ways that humans travel, manufacture, and perform their jobs. As such systems grow more prevalent, assuring that they operate safely, with acceptable risk to humans and the environment, becomes ever more important. As they grow more complex, assuring safety becomes commensurately more difficult. Such systems operate in environments marked by inaccessibility, unexpected weather conditions, or unpredictable behavior by surrounding humans. One recent example is the crash of the ExoMars Mars Lander, likely due to an implementation mistake that failed to account for timing inconsistencies between sensors [1]. As a result, the lander "thought" it was on the ground when it was actually several kilometers above Mars, costing $350 million in lost equipment and time. Diagnosis was difficult because the environment (Mars) is inaccessible.

Assuring the safe operation of critical software is not a new problem, and various techniques have been proposed to both formally verify important properties [42] and identify dangerous defects in such software. In the latter category, *robustness testing* describes a class of approaches that evaluates the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. At a high level, robustness testing constructs tests of systems or components, drawing inputs from known boundary, exceptional, or invalid cases, and then checks for failures such as aborts. It has found critical defects

in software from operating systems [15, 24], drivers [29], the Java API [9], and libraries [14], to safety-critical software [36].

The considerable evidence supporting the efficacy of robustness testing across many types of software [24, 31] suggests that it may prove a fruitful technique for testing autonomy systems. However, there are critical differences between autonomy systems and the traditional software systems to which these types of techniques have been applied. By contrast, autonomy systems are notably stateful, temporal, distributed, and cyber-physical. These properties have implications for efforts to apply traditional robustness testing approaches to autonomy systems. For example, statefulness suggests that single function call injection on system components is unlikely to be effective; instead, tests must scaffold the system sufficiently before sending erroneous inputs.

We propose Automated Stress Testing for Autonomy Architectures (ASTAA), a system that automatically generates and runs tests to effectively find defects in real-world, industrial autonomy systems. ASTAA builds on traditional robustness testing, drawing from a dictionary of exceptional values to construct test inputs to systems and components. However, key elements of ASTAA's approach are driven by the above-mentioned observations about the features of autonomy systems as compared to other systems. For example, rather than injecting values on function interfaces, ASTAA *intercepts* messages on a distributed system and injects exceptional values into live system messages, acting as a proxy. ASTAA also goes beyond traditional crash and hang monitoring by implementing runtime monitors to check for violations of system-specific safety invariants, addressing the safety-criticality of cyber-physical autonomy systems.

The National Robotics Engineering Center (NREC) develops and matures robotics technologies and solutions from concept to commercialization. At NREC, our team develops methods and tools to find safety problems in autonomy systems that are unlikely to be discovered by other types of tests or extended field testing. This context affords us access to large, real-world autonomy systems, both open- and closed-source. Over the course of five years, we applied ASTAA to 17 such systems, including humanoid robots, industrial manipulators, and unmanned aerial, ground and sea vehicles. In addition to informing our testing approach, these systems let us quantitatively and qualitatively characterize the types of bugs common to these diverse projects. For example, we uncovered a defect in a mobile manipulator that could result in a collision between the robot arm and the mobile base.[1] In another instance, we exercised the safety-critical speed limits of an Unmanned Ground Vehicle (UGV) and identified a failure mode in which the speed limit could be violated.[2]

When describing our testing approach, we outline both the features of autonomy systems that render them amenable to traditional robustness testing, as well as those that challenge standard approaches. For example, perhaps unexpectedly, despite their statefulness and heavy reliance on control loops, bugs in autonomy systems are often low dimensional, for some definition of "bug dimension." On the other hand, effective robustness testing of such systems almost always requires more than the single function calls to interfaces that characterizes many previous approaches. These observations can inform future system developers and researchers as they seek to build and support better, safer, and more robust autonomy systems.

Overall, we describe (1) a set of testing techniques that build on classic robustness testing, with key innovations specifically targeting autonomy systems, outlined through the development of ASTAA; (2) a characterization of more than 150 bugs that ASTAA found in real-world autonomy systems over five years, with implications for both testing and development of robust autonomous systems; and (3) several substantiated observations about important properties of autonomy systems, and how such properties should influence the way both developers and researchers approach them.

The rest of this paper is organized as follows. Section 2 provides background on robustness testing and autonomy systems. We detail ASTAA in Section 3. Sections 4 and 5 describe the results of applying ASTAA to many real-world systems, validating ASTAA design principles and providing lessons for developers and researchers. Section 7 outlines related work; Section 8 concludes.

2 BACKGROUND

This section overviews important concepts in robustness testing (Section 2.1) and the autonomy systems ASTAA targets, with particular focus on the features that differentiate such systems from other software (Section 2.2). We focus strictly on background necessary to understand our contribution here; we detail additional related work in Section 7.

2.1 Robustness Testing

Testing is rigorously defined as "the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior" [4]. Test automation is beneficial because human effort is expensive [41], and human biases can lead them to forget important off-nominal test cases [43].

Black-box testing refers to the testing of an interface without reference to source code, by drawing inputs from the program's input space. A straightforward way to do this automatically is by providing random inputs to its interfaces, sometimes referred to as *fuzz testing* [30, 31]. *Robustness testing* is a variant of black-box testing that evaluates system robustness, or "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [38].

We previously developed Ballista [26], a well-known robustness testing technique whose design principles are fundamental to the development of ASTAA. At a high level, Ballista sends exceptional inputs, taken from a dictionary thereof, to outward-facing software system interfaces, such as system calls or library functions. Example exceptional inputs include `NULL` pointers, `MAXINT`, or `NaN`. The input dictionary also includes likely-valid values, such as small positive integers. This enables the generation of tests with partially valid parameters, such that all parameters can be tested. This input dictionary was generated by the Ballista developers over years of experimentation.

Ballista executes each function under test using each selected set of parameters and categorizes anomalous test results using

[1] https://www.youtube.com/watch?v=kK6iKwjKA54
[2] https://www.youtube.com/watch?v=W_qBF2OjGtE

the CRASH scale [26], focusing in particular on Catastrophic, Reset, and Abort errors. In total, the Ballista project tested 30 systems, including Windows 95 through XP, as well as many POSIX systems, and found a robustness failure rate of over 10% for most of them, along with 57 system killer functions, to which a single call with invalid arguments would bring down the entire OS [24]. The input dictionary approach is more scalable than truly random fuzz testing, because exceptional values for each parameter are drawn from a small input dictionary, rather than the entire input space of the program. A key takeaway from the success of the effort is that the same dictionary discovered robustness vulnerabilities in many systems [24]. Interface testing is effective because in practice, many errors are caused by one or a combination of at most two inputs [35].

2.2 Autonomy systems

Broadly, robotic and autonomy systems are software systems that interact with the physical world, usually to assist or automate some human task. Such systems typically comprise components that communicate by publishing messages on a bus, where a component is a "natural chunk of software" [23] such as an object, separately compiled module, library, subroutine, or sensor driver. For example, a simple autonomous robot is composed of motion planning, mapping, and sensing/actuation components.

Because such systems control physical components, they display several important properties that differentiate them from typical software such as OSs, frequently the target of previous robustness testing efforts. In particular, autonomy systems are:

Stateful. Autonomy systems typically use large amounts of internal state to track and interact with the physical world, as well as long-term or complex user intentions, such as a mission or plan. For example, an autonomous robot might use a camera and/or lidar to build a model of the world that it updates as it moves to previously unexplored areas. By contrast, OS state is frequently incidental, involving memory boundaries or file state, rather than deliberate [26].

Temporal. Autonomy systems frequently have temporal and sequential requirements, as these systems are designed to execute checklists/recipes the way a human would. For example, a common behavior for an autonomous mobile robot is to go to a point in the map while gathering new data, and then use the new data to compute the next point. By contrast, much of traditional software is transformational [6], where inputs are received, calculations performed, and outputs emitted, without consideration for timing.

Distributed. Autonomy systems typically comprise multiple controllers/software modules that communicate over a network interface. As a result, autonomy system execution is typically conducted via the sending of messages over a communication channel. For example, an autonomous robot might have controllers to command its wheels to attain a certain velocity, a map generator, and a perception system, all communicating over a bus. Although the specific communication protocol can vary between systems, messages typically comprise a header and message data. For example, messages capturing odometry information might start with a timestamp and sequence number, with content composed of vectors of floats representing pose (position and orientation) and linear and angular velocity.

Cyber-Physical. Autonomy systems are typically Cyber-Physical Systems (CPSs) that integrate both software and physical components, which makes them different from traditional software in two important ways. First, autonomy systems are almost always safety critical. The physical components (e.g., a vehicle, or a robot arm) that allow them to interact with the world usually impose safety properties (e.g., "don't collide with obstacles"). Failure to adhere to these properties risks injury or loss of life, as well as damage to the system itself. Second, CPSs characteristically contain control loops that compensate for error in the actuation of their physical components. Loop closure, where sensor readings are used to calculate the next actuation, makes autonomy systems very different in behavior from classical software, which is largely devoid of feedback [6].

3 APPROACH

ASTAA is a robustness testing framework that seeks to automatically discover safety problems in autonomy systems. At a high level, ASTAA *injects* exceptional values at component interfaces to test the robustness of modules within such a system. However, ASTAA also includes several key features necessary to effectively target traditional robustness testing to autonomy systems:

Use of safety invariants. Because autonomy systems are typically CPSs, ASTAA goes beyond the crashes and hangs that are traditionally checked for in robustness testing. ASTAA also monitors safety invariants, which can express safety properties that are more complex than those of the traditional oracles.

Messages as the injection interface. ASTAA manipulates values in system messages sent over the internal network. That is, test definitions in ASTAA are runs of the system, often spanning multiple messages, rather than single function calls. This leverages the typically distributed nature of autonomy systems.

Interception testing. A key conclusion of the development of ASTAA is the necessity of *interception testing*, which mutates live message values. This both reduces ASTAA's reliance on user-provided sequences to test stateful and temporal properties, and is critically important for closure of CPS control loops.

Figure 1 illustrates the three main phases of the ASTAA testing process. The process begins by using existing documentation from the System Under Test (SUT) to manually produce an ASTAA Test Specification (Section 3.1). The test specification is then combined with information from an Exceptions Database to generate a set of test cases (Section 3.2). The ASTAA Test Runner automatically executes the test cases on the SUT, monitors for invariant violations, and logs results (Section 3.3). The section concludes with a discussion of properties of certain systems that ASTAA exploits when possible (Section 3.4).

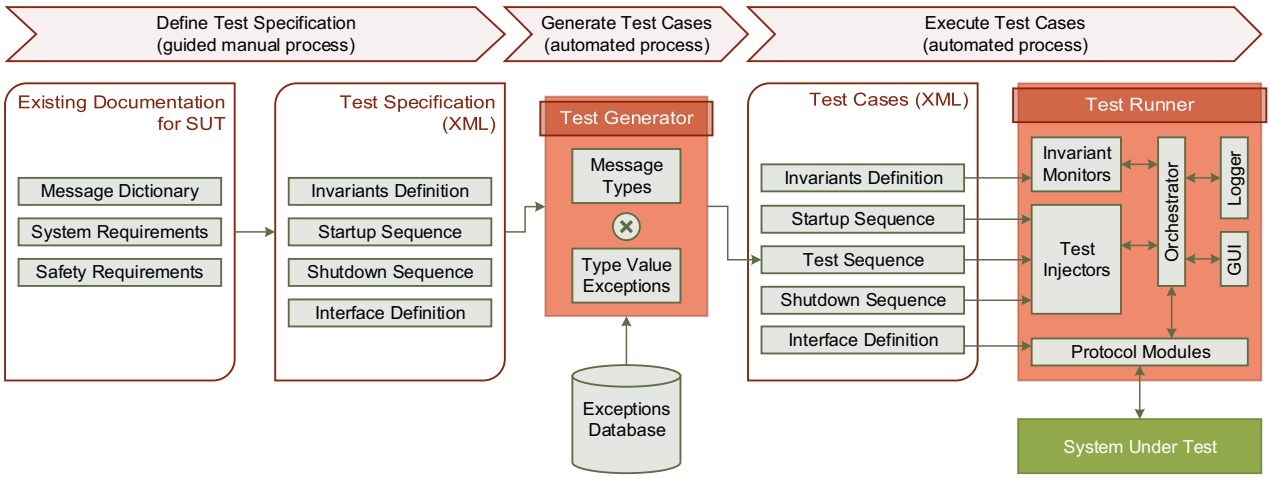

Figure 1: ASTAA architecture diagram.

3.1 Define Test Specification

The user performs several guided steps to tell ASTAA what a new system looks like and how it is exercised. This information is stored in the ASTAA Test Specification. It describes the distributed system interfaces (including message definitions describing how the system modules communicate over the network), necessary startup and shutdown sequences, and the system safety invariants. This information is typically found in the system documentation, including the safety requirements which can in many cases be translated directly from English to formal language invariants using a template [13, 19].

"Operating safely" in the autonomy system context not only includes avoiding crashes or hangs, but also maintaining safety-critical properties of the SUT. Indeed, safety property violations can be more dangerous than traditional software failures, like crashes. An autonomous vehicle that violates its speed limit around human operators can cause a lot more harm than one that aborts and stops moving. Safety invariants in ASTAA are encapsulated in evaluable expressions, such as a state machine or a formal language clause (such as bMTL[20]). System safety invariants must always be valid for the system to be considered safe (e.g., "No matter what the input, the system should not move if the E-stop is engaged"). We hypothesize that important autonomy bugs would remain uncaught if ASTAA only identified crashes or hangs.

3.2 Generate Test Cases

Given a Test Specification, ASTAA automatically generates Test Case descriptions, following the Ballista approach of injecting spurious data from an exceptions database (Section 3.2.2). However, autonomy systems are typically distributed software/hardware components communicating via message passing over a network. As such, a more natural approach for this class of systems is to treat system messages as the injection interface. That is, rather than providing invalid function call arguments, ASTAA injects invalid values into the system message fields. Moreover, rather than generating entire messages, ASTAA manipulates the fields of messages during live system execution (what we call *interception testing*).

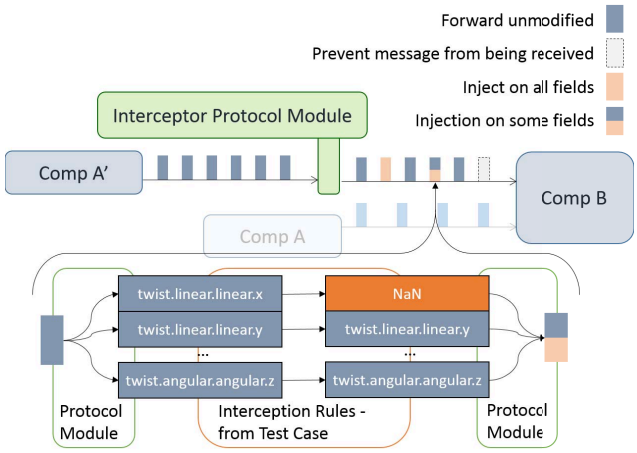

Figure 2: Interception in the ASTAA architecture

3.2.1 Interception testing. Injecting exceptional values at the message level does not intrinsically preclude the use of traditional approaches. However, autonomy systems are ill-suited to such approaches. First, highly stateful systems respond poorly to direct injection testing and fuzzing because they typically require substantial setup/handshaking to reach a responsive state [18, 24]. While providing constructors can address autonomy system setup [3, 34] in some cases, they do not suitably address autonomous control loops, which require ongoing feedback from the environment. This need for ongoing interaction means that simply getting the system into a responsive state is necessary, but not sufficient.

To address this challenge, ASTAA implements *interception testing*, which "intercepts" (changes) specific messages or fields in a running system, leaving other fields in a message unchanged. This process is displayed pictorially in Figure 2. Interception testing applies principles of mutational fuzzing [46] and man-in-the-middle [18] attacks. This allows ASTAA to perform a variety of tests that would otherwise be impossible in the presence of control loops. For example, ASTAA can test whether a given control loop is stable in the presence of the *occasional* spurious data point, important for systems operating in noisy environments. Traditional injection testing is unable to conduct such a test because doing so requires loop closure.

3.2.2 ASTAA Test Cases. Test generation produces ASTAA Test Cases, which are XML files that describe how to test the system. A test case consists of a list of interception policies describing how to modify a stream of messages on a running system. The policies take the form:

for all messages of type A on channel X
between time t_i and t_j, set field f to value v.

Channels (X), message types (A), and fields (f) come from interface definitions; timestamps (t_i, t_j) are randomly generated. To produce invalid field values (v), ASTAA uses a dictionary of type-dependent exceptional values, consistent with previous robustness testing approaches (see Section 2.1). The values in the dictionary come from previous work, augmented over the course of ASTAA development and deployment.

Note that ASTAA does not manipulate every message in every test execution, and instead passes some unmodified messages through to the system. This approach has two important benefits. First, it addresses setup/scaffolding automatically, by forwarding unaltered system behavior. As such, ASTAA can manipulate a SUT past startup/initialization and into a more interesting testable state without the need for user defined constructor/startup sequences. Second, ASTAA can test the effects of exceptional values in different states of the autonomous system, without needing to explicitly architect that state. Moreover, other unaffected messages can cause injected exceptional values to be pulled from memory and exercised. We hypothesize that nominal messages, unaffected by interception, allow ASTAA to find deeper bugs than could be discovered by traditional injection, which would be unlikely to generate nominally correct messages, and could thus only activate shallow faults.

3.3 Execute Test Cases

Given a running SUT, the ASTAA Test Runner uses the Test Case descriptions to modify communication between components of the SUT, mediated by the *Protocol Modules*. The rest of this section details runtime message manipulation and monitoring (Section 3.3.1) and test logging and minimization (Section 3.3.2).

3.3.1 Message manipulation and monitoring. ASTAA is designed to be universal, interacting with the SUT through an extensible test interface. This interface uses *Protocol Modules* to translate the messages generated by the test case to network messages in the SUT, and vice versa for analysis. These modules simply pack and unpack data at the network message layer, and are thus reusable across systems; We have written several for general use, including ROS (Robot Operating System) and CAN modules. Given appropriate Protocol Modules, ASTAA requires no additional user involvement beyond initial setup.

ASTAA detects safety violations and crashes/hangs in the SUT and its subsystems by monitoring system behavior while exercising it. Crashes or hangs can usually be detected through the middleware layer (e.g., ROS) or operating system (e.g., by polling active processes). ASTAA evaluates safety invariants from the Test Specification using runtime monitoring, and interface monitoring in particular, to examine the values passed on the bus. Interface monitoring can effectively detect system faults without requiring source modification [21, 37]. Such monitoring can be done online, on a live system, or offline, on a collected log file [17]. Both online and offline monitoring were used in testing systems over the course of the ASTAA project, depending on which was more suitable for a given SUT.

3.3.2 Test logging and minimization. Because test cases are lists of rules for modifying a run of a system influenced by non-deterministic noise (e.g., environmental, timing noise), test cases themselves are not deterministic; different runs can produce different results. Thus, ASTAA provides *test logs* to describe the totality of system runs that lead to error. This log includes both the modified and unmodified system messages. Unlike the test cases, a test log contains all the information needed to replicate a failure and can be replayed.

However, developers may benefit from knowing more precisely which message conditions are necessary for a violation. Thus, when possible, ASTAA attempts to find a minimal failure-generating test case. It first uses delta debugging [47] to produce a minimal set of log messages necessary to trigger the failure during replay. It then uses Hierarchical Product Set Learning (HPSL) to identify the fields in those messages relevant to the bug, and what values they must assume [45].

Initial test logs can be very high dimensional, containing many messages, each with many fields. A key question in ASTAA development is the degree to which bugs in autonomy systems are similarly high-dimensional, particularly with respect to their activation distance from nominal behavior. The definition of dimensionality is nuanced in this context, as we discuss further in Section 4.2.1.

Test case minimization is extremely informative when possible; however, not all systems can be replayed. When minimization is impossible, the ASTAA Test Log can still provide useful information, giving a specific instance of a system failure, which can substantially aid diagnosis.

3.4 Testing Shortcuts

The techniques discussed above allow ASTAA to test systems that exhibit characteristics that are common to autonomy systems. However, not every system or module within a system exhibits all of these characteristics, and in many cases simpler testing methods can be used to great benefit. For example, when SUTs don't have control loops, log replay is possible, aiding in bug finding through minimization, as mentioned in Section 3.3.2.

3.4.1 Replay and subsystem testing. Certain subsystems lack the feedback/control loops that motivate live interception testing. Some are simple transformational systems that take input and produce output without feedback (e.g., coordinate transform libraries or

planners). Such subsystems can be tested without a full simulation by starting them up in isolation and replaying logs, as without feedback or control loops, changing the log does not cause data consistency problems.

This can be substantially simpler, as many SUTs have existing facilities for log replay, whereas interception of live system messages can be difficult. For example, to intercept between n nodes on a bus, ASTAA must replace the bus with a star topology of n nodes, intercepting at the gateway. This applies to both hardware, where physical buses must be modified, and to simulations, where network channels need to be rerouted, increasing network load. Log modification and replay precludes a need for rerouting, simplifying testing.

More than half of the bugs found by the ASTAA project were found via subsystem testing, with many of them traced to system-level activation, such as the bug detailed in [45].

3.4.2 Injection with constructors and interleaving. Two additional techniques allow ASTAA to function effectively even when live interception is difficult. First, injection with constructors [3, 34] supports effective testing of transformative subsystems without control loops. This is similar to how log replay interacts with the system, but it allows a tester more control over the states of the system being tested at the cost of automation.

Another simplified approach is to *interleave* fully generated messages in between live system messages, which splits the difference between interception and traditional injection. This allows some live system messages to pass through (e.g., those relevant to a control loop), while still injecting spurious data. Since interleaving only involves adding messages to the bus, not modifying or removing them, it can be substantially easier to implement than interception. While it does provide some of the benefits of interception, it is also more limited in its ability to test various interface aspects (e.g., fields in the same message as the control loop feedback cannot be altered). Interception is more powerful, but we found interleaving useful when interception was impracticable.

3.4.3 Brittle fields. Over the course of system testing, we identified several special fields that have meaning to the middleware layer (e.g., timestamps and sequence numbers), rather than just the SUT itself. We found that testing on these fields seldom uncovered meaningful bugs, and usually instead resulted in simple message rejection or shallow assertion failures. We therefore frequently achieved deeper system testing by *not* intercepting on the following types of fields: timestamps and sequence numbers, meta-values (e.g., values indicating how to parse message content), and string signatures (e.g., version numbers). While testing these fields can find bugs, we found it generally more productive to bypass them, as spurious data in these fields often cause early failures that mask other, more-critical faults.

4 TESTING EXPERIENCES

From autonomous land, sea, and air vehicles, to legged robots and mobile manipulators, ASTAA has tested a diverse range of robotic systems. We tested both open-source and proprietary systems across commercial, defense, and academic domains. For example, we applied both module and system-level testing to the motion planning software developed for an autonomous helicopter program. We also tested the planning system developed for an unmanned naval vessel, testing strictly at the system-level. We tested an autonomy kit that can be applied to UGVs, as well as a software system that enables teleoperation of such vehicles across long distances.

Many of the robots we tested are based on ROS, an open-source collection of libraries and tools that provides a framework for robot software. We tested a variety of functional components in a variety of systems, including perception, planning, and control modules. The Technology Readiness Level (TRL) [2] of the SUTs generally ranged from 4 ("component validation in a laboratory environment") to 6 ("system/subsystem prototype demonstration in a relevant environment").

Although we cannot disclose the actual testing results for any specific SUT, we can analyze and describe in broad terms the kinds of bugs that were found throughout the course of the ASTAA project. The bug reports included in this analysis cover four years of ASTAA testing and contain over 150 bugs from 11 distinct projects.[3] The reports include varying levels of detail for diagnosing and replicating the bugs. To characterize the kinds of problems that ASTAA uncovered, we reviewed each bug report and assessed the issue across several dimensions. Three of the authors took part in the review, independently evaluating the reported bugs using the same set of definitions. We then compared and merged the individual assessments, resolving any disagreements through discussion and consensus. Note that some bug reports did not include enough information to make a determination in certain dimensions, so categories do not always sum to 100%.

While the available data does not support a comparative study contrasting bugs found by traditional testing techniques with those found by ASTAA, we can make still broad qualitative assessments about the types of robustness bugs that manifest in autonomy systems and how to find them, with a focus on understanding what differentiates—and doesn't—autonomy systems from traditional software. Such an analysis helps assess our design decisions, and provides lessons for practitioners and researchers moving forward.

4.1 Autonomy system-specific features

Some ASTAA design choices were necessarily set in stone, dictated by features of the systems in question. For example, testing at interfaces is necessary for the majority of systems in this domain (Section 3.2). However, other design decisions were informed by shortcomings we encountered in adapting general software robustness testing to autonomy systems over the course of ASTAA development. This section discusses patterns of robustness bugs that corroborate many of ASTAA's autonomy-specific features.

4.1.1 Scaffolding Messages are Necessary. ASTAA's design reflects our hypothesis that many bugs in robotic systems can only be activated and thus discovered with sufficient scaffolding messages.

For such systems to reach an interestingly testable state, they typically need to be initialized via **startup messages**. Additionally, certain bugs only manifest when a system is in a certain mode,

[3] While the ASTAA project tested 17 systems, for some systems we are not permitted to disclose the results of testing, even in aggregate. While we did not perform an analysis of bug reports for these systems, we have no reason to believe the results would be any different.

requiring further messages to bring them to that mode. For example, ASTAA triggered a bug in a robotic vehicle that only manifested when the vehicle was in reverse (the absolute value of the speed was not taken before comparing it to the desired speed limit). Finally, because these systems are temporal, some bugs arise due to a bad value propagating through a system (e.g., bad values accumulating in a filter). In this case, **turnover messages** are needed to keep the system running after the faulty value is injected to render the effect visible.

We were able to classify 133 bugs in our dataset according to the scaffolding messages required to trigger them. While 59 bugs *did not* require any scaffolding messages, 74 bugs did require some combination of startup and/or turnover messages to manifest. Startup messages are were required in more cases than turnover messages: 36 bugs required startup messages alone, while 14 bugs required turnover messages alone. An additional 24 bugs required both startup messages and turnover messages. This corroborates our hypothesis that bugs in autonomy systems often require scaffolding, and testing techniques should operate accordingly.

4.1.2 Deeper Invariant Monitoring is Valuable. Not all tested systems were associated with safety specifications; for such systems, the only bugs found were those violating the traditional crash/hang invariants. **Crash invariants** detect problems that cause a process to terminate with an error code or core dump. **Hang invariants** identify bugs that cause a process to cease communications or become unresponsive. The majority (100, or roughly 69%) of bugs in our dataset were detected with crash/hang invariants.

However, in systems for which we did have a safety specification, the majority of bugs found are violations of that specification. For one such system, 64% of the bugs are violations of the system's safety specification, found via simple or complex invariants, and are not detectable via a hang or crash. **Simple invariants** are univariate inequalities, such as bounds checks, checks for simple missed deadlines, and output validity checks (e.g., array shape or size). **Complex invariants** are multivariate temporal or algebraic relations of two or more quantities, (e.g., "when motors are on, fans should never be off for more than 10 seconds.") Complex invariants cover safety bounds, as well as checks for inconsistent modes, erroneous (but in bounds) output values, or ignored inputs.

In many cases where invariants were not explicit, our testers inferred invariants from documentation. Our database contains 11 bugs that violated inferred simple invariants, and 8 bugs that violated inferred complex invariants, the majority of which were indeed fixed when reported.

4.2 Autonomy systems as software systems

Although autonomy systems behave differently than traditional software, and need additional testing infrastructure such as startup messages and control loop closure, we still found that they share important properties with traditional software for the purposes of robustness testing. This section discusses bug patterns corroborating these observations.

4.2.1 Autonomy Bugs are Low Dimensionality. ASTAA is built to accommodate long sequences of messages and find complex sequence failures. However, perhaps surprisingly, many bugs in autonomy systems are low in dimensionality. In traditional systems, this refers to the number of function parameters that must be manipulated to trigger a defect; many bugs can be found by manipulating only one or two such parameters. Because ASTAA constructs tests by manipulating messages, defining "dimension" is less straightforward. We define *mutation dimensionality* across both injection *interfaces* and manipulation *instances*. Injection interfaces refer to the number of fields or message types that must be manipulated to activate a bug:

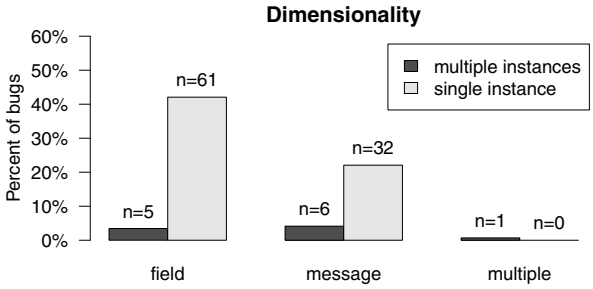

Figure 3: Most bugs found exhibit low interface and instance dimensionality.

field The bug is activated by injection on a single field in a single message type.

message The bug is activated by injection on multiple fields in a single message type.

multiple The bug is activated by injection on multiple message types (implying injection on multiple fields).

Manipulation instances refer to the number of times a value must be injected into a field or message to activate a bug:

single The bug requires only a single manipulation of the injection interfaces.

multiple The bug requires multiple manipulations of an equivalent injection interface. (e.g., multiple mutations of the same single field in a single message type)

Note that approximately 5% of the bugs were triggered without manipulation at any interface, typically corresponding to configuration errors (e.g., using the wrong format in a library call due to version changes). We were surprised by the number of such bugs, but speculate that ASTAA's edge case testing may exercise otherwise unused code outside the usual functional/regression testing envelope that would normally detect such bugs after Application Programming Interface (API) changes.

Figure 3 shows results for the bugs we could classify. The bugs exhibit low mutation dimensionality in both interfaces and instances. 42% of the bugs can be activated by sending a single faulty value to a single field. Another 22% of the bugs were related to multiple fields in the same message, yet still only required a single bad message. Only 12 bugs, or 8% of those we examined, required multiple instances of faulty data. Only one bug needed multiple instances across multiple message types to activate.

We conclude that, for this definition, the bugs we found in autonomy systems are low in dimensionality. That said, these bugs

are specifically low in "mutation dimensionality". Practically, this means that the distance between a fault activating input pattern and a pattern that appears in normal operation is low, and therefore this input pattern is likely to appear during the operation of the system [28].

However, a traditional software testing definition of bug dimensionality is the "number of conditions required to trigger a failure" [27]. In this sense, many of the bugs in autonomy systems are high dimensionality. Test case minimization often finds that a bug requires many constraints on message fields [45]; The bugs ASTAA found often contained as many 10 to 14 constrained fields.

This result demonstrates why interception is more effective that traditional injection in this context. Because interception testing mutates existing messages, the likelihood of discovering a bug relates to its mutation dimensionality, which is generally low. Traditional injection testing would have to construct these messages from scratch, and is unlikely to randomly generate a given 14-field-constrained message.

4.2.2 Sanitization and wrappers are highly effective. Testing traditional software has shown that, when possible, developers should sanitize and validate system input, including configuration files and network traffic. While this can incur some small overhead, it substantially impacts system robustness [11].

Sanitization wrappers are univariate expressions that remove exceptional values (e.g., `Inf`, `NaN`, empty objects, etc.), check bounds, enforce limits, etc. This may include sanitizing *non-application specific values*, such as array shape/size, non-negative values, and invalid enums. **Consistency** wrappers are typically expressed as multivariate relationships that enforce consistency between values, such as index vs. array length, $width * height$ vs. array size, etc.

The effectiveness of wrappers is visible in the ASTAA testing results, where 58 bugs were preventable using sanitization wrappers and 40 bugs were preventable using consistency wrappers. Only 14 of the identified bugs would not have been preventable using wrappers. In these autonomy systems, the majority of the problems could be avoided via sanitization and consistency checking to protect against invalid inputs, as in traditional systems.

5 RECOMMENDATIONS

There are several additional recurring lessons that the ASTAA team observed over the course of testing real systems that can help developers build more robust autonomous systems and researchers build better tools to support them in doing so. While ASTAA is not a code analysis tool, in some cases we were able to trace bugs to code, or system developers provided insight when a bug was reported. While such snippets are by necessity anecdotal, the lessons, along with other best practices for distributed embedded systems, have been previously observed [23, 25]. Our experience suggests that they remain underappreciated, and under-supported in practice.

Protect your robots from data assumptions. Developers often need to make assumptions about the data fed to their systems, and problems arise when those assumptions are flawed. For example, developers often trust that configuration files are valid, which can lead to difficult-to-diagnose startup and consistency problems (e.g., initialization values for sin_x and cos_x that are both 0).

Data assumptions can also cause problems in dynamic data or messages. For example, fields can be semantically redundant but inconsistent within a message (e.g., the 'size' of 2d array may be inconsistent with its 'height' and 'width'). Message timestamps are often incorrectly assumed to be monotonic, safe from overflow [25, 32]. We see the impact of flawed assumptions both in the degree to which autonomy bugs can be mitigated with sanitization (Section 4.2.2) and broadly over our experience.

Floats and `NaN`s are useful but dangerous. Floating point numbers are a frequent source of unexpected software errors [16]. They are a source of particular concern in autonomy systems, because they are often used in CPS-specific contexts. For example, wrapping angles to between $-\pi$ and π radians is a common function, but it must be carefully implemented. If the wrapping is computed within a subtractive loop, the lack of float precision can lead to an infinite loop. In one of the systems we tested, such a bug led to a hang when given a sufficiently large input angle. In general, floats should not be used as iterators.

Furthermore, `NaN` values propagate dangerously [16] by causing problems with control flow structures (all comparisons with a `NaN` return `False`) and data flow (any operation with a `NaN` becomes a `NaN`). For example, one robot ASTAA tested compared the current speed to a desired limit. When the current speed was `NaN`, this comparison always returned `False`, and so the system was never determined to be above the limit; it sped up indefinitely. Defensive checks (`if` guards with error handling branches on the `False` condition) can protect against `NaN`s and prevent dangerous actuation in autonomy systems.

Plan for the system to fail. No code is ever bug free, but the best way to eliminate bugs is to find them when activated. Nodes should have means to report errors to the rest of the system, and should not fail or automatically restart silently. Finally, logging that is consistent, parsable, and replayable is invaluable for error detection, diagnosis, and debugging. This suggestion comes both from our experience, and from the developers whose systems we tested (and to whom we reported bugs). Many system developers were enamored of ASTAA's monitoring and logging capabilities.

6 THREATS AND LIMITATIONS

There are several threats to the validity of our results. First, we categorized bugs manually, and thus human biases may influence our results. We mitigate this risk by aggregating results across three expert human raters, who reached consensus conservatively. Implementation errors in ASTAA are a related threat, as spurious bug reports could influence our results. Overall, however, the bugs ASTAA finds are identified by well-established monitoring techniques from the robustness testing literature or via safety invariants defined or verified by the SUT creators/maintainers, external to the ASTAA development team. ASTAA has been successfully applied to a number of real-world systems, and its reported bugs validated by those independent systems' developers, suggesting the risk of relevant implementation error is low. Additionally, although ASTAA must remain closed-source, we note that it is the result of active development over several years by a team of experienced developers.

Our evaluation is observational rather than controlled: We do not compare ASTAA to other traditional techniques to assess particular elements of our claims. This risk to our claims is mitigated by the fact that we applied ASTAA to large, real-world, industrial autonomy systems over the course of several years. Note also that we do not make strong statistical claims. Instead, characterization of bugs along several axes of interest produces evidence to corroborate ASTAA's design principles and informs observations about how to test autonomy systems. The size, variety, complexity, and reality of the SUTs we study also mitigates the risk that our evaluation may not generalize, despite having been performed on a particular finite set of systems to which we were given access.

Finally, our evaluation is mostly of systems at TRL 4–6. It is possible that systems at higher TRLs would be free of such robustness bugs. However, the bugs found in mature libraries and more mature research systems indicate that a lot of these problems persist. Additionally our results valuably indicate the kinds of robustness mistakes that should be tested for prior to full deployment.

7 RELATED WORK

For brevity, we restrict our attention to work in robustness testing and autonomy system robustness. We direct the interested reader to Bertolino [4] for a thorough survey of software testing practice history and an overview of the meaning of software test selection, execution, and analysis.

While robustness testing for autonomy systems is relatively novel, testing techniques for traditional software systems have existed for decades. The naive approach of generating arbitrary random input to a function to test it is known as fuzz testing. Fuzz testing efficiency can been improved by constraining the input space or by introducing statefulness [40]. Statefulness has been introduced using model-based testing [10, 39], which attempts to generate tests based on a model of the functional requirements.

Models of requirements make it easier to generate test cases that wouldn't otherwise be created. However, Dalal et al. note that model selection is open-ended and has a large impact on test effectiveness, hindering automation [10]. Further work confirms these tradeoffs [12, 42]. Some tools, such as AutoFuzz [18], mitigate user involvement by automatically learning models from historical data.

To the best of our knowledge, little previous work tackles stress testing robotic or autonomy systems. Researchers have primarily focused on either identifying sources of possible faults during design [22] or during live system execution [44], typically with a focus on hardware failures. A historical analysis of robotic system failures indicates that while attention to hardware is warranted, software remains one of the major sources of error [5].

Chu [8] performed practical robustness testing of a middleware layer for autonomy systems. Among other observations, they noted the difficulty of testing large hierarchical multi-component systems, and the necessity of safety invariants. However, the experience was limited by the necessary simplicity of the fault model: insertions, deletions, and swaps of messages in prerecorded data. They were unable to consider component crashes/aborts as part of their robustness metric.

Comparatively, our testing framework is designed explicitly for autonomy systems, accounting for their various properties. Unlike model-based fuzz testing tools, it does not suffer from mischaracterization of the input, since it does not rely on models, and instead intercepts live data. Our fault model is also much more expansive than that in Chu's work, and the closure of the control loop allows for crashes (aborts) to be measured as part of the system robustness.

In addendum, it should be noted that the focus of ASTAA is adapting the existing Ballista tool to the unique testing domain of robotics and autonomy software. While ASTAA does not take advantage of many of the recent developments in robustness testing, such as combinatorial testing [33] or the testing ecosystem that has sprung up around Android testing [7], it's possible that these techniques could be used in the robotic systems domain by making similar modifications to them as those presented in this paper.

8 CONCLUSION

Robustness in the face of unexpected inputs is particularly important in autonomy systems. However, there are many ways autonomy systems are both similar to and distinct from classic software systems that have important implications for how to robustness test them. To meet these needs, we designed ASTAA, a tool that builds on classical techniques, but incorporates novelties that allow it to address the specific challenges of testing autonomy systems. We used ASTAA to test 17 systems across a variety of domains for robustness weaknesses. Qualitatively and quantitatively analyzing the bugs discovered in a subset of these systems supports many of the design decisions behind ASTAA, including certain principles that it takes from prior robustness testing approaches.

Much of the discussion in this paper is meant to serve as a practical guide for researching, developing, and testing autonomy systems, backed by an analysis of information gained over the course of testing many such systems across a variety of domains. However, one broader lesson to take away from our work is that it is not only possible to robustness test autonomy systems, but doing so can find easy-to-activate bugs that seriously threaten real systems. In one case, ASTAA discovered a critical flaw that could lead the angle of robotic manipulator to be set to an unbounded negative angle. This would allow the manipulator to collide with—and damage—the robot base. Developers initially ignored our bug report, dismissing the input case as implausible. They fixed the bug a few weeks later, after they accidentally triggered it on their actual robot, leading the manipulator to seriously damage an expensive end effector. ASTAA provides a scalable approach for testing such systems and finding real, important faults before they happen in the field, where they could damage far more than just an end effector.

ACKNOWLEDGMENTS

This material is based upon work supported by the Test Resource Management Center (TRMC) Test and 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-11-C-0025, "Stress Testing for Autonomy Architectures (ASTAA)". Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the TRMC T&E/S&T Program and/or the U.S. Army PEO STRI.

REFERENCES

[1] European Space Agency. 2016. Schiaparelli Landing Investigation Makes Progress. (23 Nov. 2016). Retrieved January 17, 2017 from http://www.esa.int/Our_Activities/Space_Science/ExoMars/Schiaparelli_landing_investigation_makes_progress

[2] Assistant Secretary of Defense for Research and Engineering (ASD(R&E)). 2011. *Technology Readiness Assessment (TRA) Guidance.* Technical Report. U.S. Department of Defense. http://www.acq.osd.mil/chieftechnologist/publications/docs/TRA2011.pdf

[3] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr". In *Proceedings of the 9th International Conference on Information Security (ISC '06)*. 343–358. https://doi.org/10.1007/11836810_25

[4] Antonia Bertolino. 2003. Software testing research and practice. In *International Workshop on Abstract State Machines*. Springer, 1–21.

[5] J. Carlson and R. R. Murphy. 2005. How UGVs physically fail in the field. *IEEE Transactions on Robotics* 21, 3 (June 2005), 423–437.

[6] Paul Caspi and Alain Girault. 1995. Execution of distributed reactive systems. In *European Conference on Parallel Processing*. Springer, 13–26.

[7] S. R. Choudhary, A. Gorla, and A. Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 429–440. https://doi.org/10.1109/ASE.2015.89

[8] Hoang-Nam Chu. 2011. *Test and evaluation of the robustness of the functional layer of an autonomous robot.* Ph.D. Dissertation. Institut National Polytechnique de Toulouse - INPT. https://tel.archives-ouvertes.fr/tel-00627225

[9] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience* 34, 11 (2004), 1025–1050.

[10] Siddhartha R Dalal, Ashish Jain, Nachimuthu Karunanithi, JM Leaton, Christopher M Lott, Gardner C Patton, and Bruce M Horowitz. 1999. Model-based testing in practice. In *International Conference on Software Engineering (ICSE '99)*. 285–294.

[11] John DeVale and Philip J. Koopman, Jr. 2002. Robust Software - No More Excuses. In *International Conference on Dependable Systems and Networks (DSN '02)*. IEEE, 145–154.

[12] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. 2007. A survey on model-based testing approaches: a systematic review. In *1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with ASE 2007*. 31–36.

[13] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. 411–420. https://doi.org/10.1145/302405.302672

[14] Christof Fetzer and Zhen Xiao. 2002. An automated approach to increasing the robustness of C libraries. In *International Conference on Dependable Systems and Networks (DSN '02)*. IEEE, 155–164.

[15] Anup K Ghosh and Matthew Schmid. 1999. An approach to testing COTS software for robustness to operating system exceptions and errors. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*. 166–174.

[16] David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)* 23, 1 (1991), 5–48.

[17] Alwyn Goodloe and Lee Pike. 2010. *Monitoring Distributed Real-Time Systems: A Survey and Future Directions.* Technical Report NASA/CR-2010-216724. NASA Langley Research Center.

[18] Serge Gorbunov and Arnold Rosenbloom. 2010. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS* 10, 8 (2010), 239.

[19] Aaron Kane. 2015. *Runtime Monitoring for Safety-Critical Embedded Systems.* Ph.D. Dissertation. Carnegie Mellon University.

[20] Aaron Kane, Omar Chowdhury, Anupam Datta, and Philip Koopman. 2015. A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System. In *Proceedings of the 6th International Conference on Runtime Verification (RV '15)*, Ezio Bartocci and Rupak Majumdar (Eds.). 102–117.

[21] Aaron Kane, Thomas Fuhrman, and Philip Koopman. 2014. Monitor based oracles for cyber-physical system testing: Practical experience report. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 148–155.

[22] Koorosh Khodabandehloo. 1996. Analyses of robot systems using fault and event trees: case studies. *Reliability Engineering & System Safety* 53, 3 (1996), 247–264.

[23] Philip Koopman. 2010. *Better embedded system software.* Drumnadrochit Education.

[24] Philip Koopman, Kobey DeVale, and John DeVale. 2008. *Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project.* Wiley-IEEE Press, Chapter 11, 201–226. https://doi.org/10.1002/9780470370506.ch11

[25] Hermann Kopetz. 2011. *Real-time systems: design principles for distributed embedded applications.* Springer Science & Business Media.

[26] Nathan P Kropp, Philip J Koopman, and Daniel P Siewiorek. 1998. Automated robustness testing of off-the-shelf software components. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*. IEEE, 230–239.

[27] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421.

[28] S. Kumar, T. W. S. Chow, and M. Pecht. 2010. Approach to Fault Identification for Electronic Products Using Mahalanobis Distance. *IEEE Transactions on Instrumentation and Measurement* 59, 8 (Aug 2010), 2055–2064. https://doi.org/10.1109/TIM.2009.2032884

[29] Manuel Mendonca and Nuno Neves. 2007. Robustness testing of the Windows DDK. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 554–564.

[30] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[31] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services.* Technical Report 1268. University of Wisconsin. http://digital.library.wisc.edu/1793/59964

[32] Erik Naggum. 1999. The Long, Painful History of Time. (Oct 1999). http://naggum.no/lugm-time.html Presented at Lisp User Group Meeting 1991.

[33] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv.* 43, 2, Article 11 (Feb. 2011), 29 pages. https://doi.org/10.1145/1883612.1883618

[34] Jiantao Pan, Philip Koopman, Yennun Huang, Robert Gruber, and Mimi Ling Jiang. 2001. Robustness testing and hardening of CORBA ORB implementations. In *International Conference on Dependable Systems and Networks (DSN '01)*. 141–150.

[35] Jiantao Pan, Philip Koopman, and Daniel Siewiorek. 1999. A dimensionality model approach to testing and improving software robustness. In *IEEE Systems Readiness Technology Conference (AUTOTESTCON'99)*. IEEE, 493–501.

[36] David Lorge Parnas, GJK Asmis, and Jan Madey. 1991. Assessment of safety-critical software in nuclear power plants. *Nuclear safety* 32, 2 (1991), 189–198.

[37] Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Rosu. 2008. Hardware Runtime Monitoring for Dependable COTS-based Real-Time Embedded Systems. In *Proceedings of the 29th IEEE Real-Time System Symposium (RTSS'08)*. 481–491.

[38] Jane Radatz, Anne Geraci, and Freny Katki. 1990. IEEE standard glossary of software engineering terminology. *IEEE Std* 610121990, 121990 (1990), 3.

[39] Fares Saad-Khorchef, Antoine Rollet, and Richard Castanet. 2007. A framework and a tool for robustness testing of communicating software. In *Proceedings of the 2007 ACM symposium on Applied computing*. ACM, 1461–1466.

[40] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery.* Pearson Education.

[41] Ossi Taipale, Jussi Kasurinen, Katja Karhu, and Kari Smolander. 2011. Trade-off between automated and manual software testing. *International Journal of System Assurance Engineering and Management* 2, 2 (2011), 114–125.

[42] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability* 22, 5 (2012), 297–312.

[43] Peter Varhol and Gerie Owen. 2013. How Did I Miss That Bug? *Pacific North-West Software Quality Conference, Proceedings of* 31 (2013).

[44] V. Verma, G. Gordon, R. Simmons, and S. Thrun. 2004. Real-time fault diagnosis [robot fault diagnosis]. *IEEE Robotics Automation Magazine* 11, 2 (June 2004), 56–66.

[45] Paul Vernaza, David Guttendorf, Michael Wagner, and Philip Koopman. 2015. Learning product set models of fault triggers in high-dimensional software interfaces. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '15)*. 3506–3511. https://doi.org/10.1109/IROS.2015.7353866

[46] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 511–522.

[47] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why?. In *Software Engineering-ESEC/FSE'99*. Springer, 253–267.