Charting a Course Through Uncertain Environments: SEA Uses Past Problems to Avoid Future Failures

Preston Moore
New York University
pkm266@nyu.edu

Justin Cappos
New York University
jcappos@nyu.edu

Phyllis Frankl
New York University
pfrankl@nyu.edu

Thomas Wies
New York University
wies@cs.nyu.edu

***Abstract*—A common problem for developers is applications exhibiting new bugs *after* deployment. Many of these bugs can be traced to unexpected network, operating system, and file system differences that cause program executions that were successful in a development environment to fail once deployed. Preventing these bugs is difficult because it is impractical to test an application in every environment. Enter Simulating Environmental Anomalies (SEA), a technique that utilizes evidence of one application's failure in a given environment to generate tests that can be applied to *other* applications, to see whether they suffer from analogous faults. In SEA, models of unusual properties extracted from interactions between an application, A, and its environment guide simulations of another application, B, running in the anomalous environment. This reveals faults B may experience in this environment without the expense of deployment. By accumulating these anomalies, applications can be tested against an increasing set of problematic conditions. We implemented a tool called CrashSimulator, which uses SEA, and evaluated it against Linux applications selected from coreutils and the Debian popularity contest. Our tests found a total of 63 bugs in 31 applications with effects including hangs, crashes, data loss, and remote denial of service conditions.**

I. Introduction

No Battle Plan Survives Contact With the Enemy — Helmuth von Moltke

No matter how well an application is tested before its release, new bugs always seem to emerge after deployment. Oracle estimates that 40% of deployed applications contain critical defects – a situation that is compounded by the fact that deployment increases the cost to fix these flaws by 100 fold [1]. One reason for this outcome is that these applications will operate within a diverse set of deployment *environments*, and variations between these environments tend to reveal previously undiscovered flaws. These flaws emerge from such factors as operating system APIs changing across versions [2], [3], [4], or small variations in file systems exhibiting subtle but critical differences [5], [6], [7]. Even if the network and adapter are identical, network behavior can still diverge from what is expected [8], [9], [10], and these environmental differences greatly exacerbate the chance that an application will function incorrectly when deployed.

These unforeseen bugs complicate the work of 43% of application developers who, according to a recent survey conducted by ClusterHQ [11], spend between 10% and 25% of their time debugging errors that only appear in production. Numerous efforts have been made to reduce this burden. One approach is to hide environmental differences behind standard interfaces. Unfortunately, even specialized "Write-Once, Run Anywhere" environments that attempt to hide these differences, such as the Java Runtime Environment, are not perfect, leading them to be rechristened "Write-Once, Debug Everywhere" [12]. A more direct approach would be to identify and fix deficiencies before deployment, but history has shown that, even if enormous effort is put forward, it may be insufficient to uncover these bugs. Microsoft employs thousands of engineers with nearly a 1:1 ratio of testers to developers [13]. Yet, a recent Windows Update released in response to the Spectre Intel CPU vulnerability resulted in machines with certain hardware configurations being rendered unbootable [14].

What is needed is a methodical way to record, preserve, and test against specific features of any environment proven to have caused incorrect behavior in applications. We achieve this by cataloging these features, which we call *anomalies*, and offering a systematic and reproducible strategy for future application tests, without requiring per application effort.

In this paper, we document the development and implementation of a new approach to finding and preserving anomalies that we call *Simulating Environmental Anomalies* (SEA). This technique is founded upon the key insight that problematic environmental properties can often be detected in the function calls, system calls, or other interactions an application makes within an environment. When employing SEA, an application under test is exposed to the anomalies unique to a given environment in such a way that its responses will indicate potential for failures upon deployment. In this way, developers are given an easy and inexpensive way to learn from the mistakes of others, and thus save money and programming hours that otherwise would be spent to find and fix environmental bugs.

We found SEA capable of finding bugs, both known and unknown, in Linux applications ranked highly on Debian's popularity contest by implementing it in a proof of concept tool called *CrashSimulator*[1] and evaluating its performance [15].

[1]Our approach is loosely inspired by flight simulators, which test pilot aptitude under a variety of rare, adverse scenarios (water landings, engine failures, etc.) before the pilots are certified to work in practice.

These applications were chosen because they are commonly used, well tested, and stable, giving CrashSimulator a chance to find new environmental bugs where it should be the hardest. Findings from these evaluations included bugs attributed to unanticipated file system configurations, file types, and network delays, and resulted in a variety of failures, including hangs, crashes, and filesystem damage. In total, the SEA technique was able to identify 65 bugs with much less time and effort than would be required to set up real environments and execute the same applications within them – illustrating SEA's usefulness for developers in a real-world setting.

The main contributions in this work can be summarized as follows:

- It provides evidence that previously unanticipated flaws can be created by the interaction between an application and its environment.
- It introduces *Simulating Environmental Anomalies* (SEA) as an easy-to-use method for simulating environments so an application's behavior in those environments can be assessed before deployment— without the time and resource costs of testing in each environment.
- It allows developers to build a corpus of extracted anomalies and thus increase their capability to test applications against problematic environmental aspects without per application effort.
- It demonstrates a new tool, *CrashSimulator*, which implements SEA in order to find previously-undiscovered environmental bugs in widely deployed and highly tested code.
- It introduces a new technique called *process set cloning* that can generate copies of a running application, so that users can test debugging hypotheses without damaging the original.
- It proves the effectiveness of *CrashSimulator* by showing it can find real bugs in real applications when used by developers both involved and not involved with the project, including developers with limited Linux experience.

II. What is an Environment?

It is important to have a clear understanding of what constitutes an application's environment in order to see how it can contribute to the presence of bugs. An application's environment consists of all of the components an application depends upon that its developers do not control. In practice, this is everything other than the code and data packaged within the application itself. Typically testers focus on explicit inputs to the application and overlook the implicit inputs coming from these uncontrolled components.

Anything external to the application can be configured in unexpected ways. For example, library search rules can result in default system libraries being loaded instead of the versions deployed alongside the application. These external resources can be thought of as providing implicit inputs to the program that affect its flow of execution. An investigation of bug reports has shown that environmental bugs in the following categories have been found in major applications.

- **Operating Systems.** Differences in the way operating systems implement system calls can influence the behavior of applications. For example, on Linux it is possible to remove an open file, yet this is not allowed on Windows systems [16]. An application written without this difference in mind could fail if it relies on one implementation or the other.
- **File Systems.** The exact file system used will also have a substantial impact on the behavior of a system, independent of the operating system. The popular Ext4 file system on Linux is case sensitive, so that "a" and "A" are different files, while in OS X's HFS+ file system those file names would refer to the same file. File systems can have varying limits or behaviors for other items as well, including file name length (popularized due to the 8.3 limitations of the FAT file system), maximum file length, number of directory entries, or depth of directories supported, all of which can lead to errors when programs do not account for these variations [5], [6]. The layout and contents of a filesystem can also impact an application's execution. Unexpected file types can result in application failures, and multi-disk layouts impose additional requirements on common operations, such as moving a file from one location to another. We explore the latter two situations in our evaluation (Sec. IV-A).
- **Network.** Both local and remote network nodes can have specific characteristics that could influence the behavior of an application. For example, POSIX operating systems support the notion of limiting the kernel buffer set aside for a socket. However, many other popular operating systems (Windows, Linux, and Mac) implement this quite differently. If a UDP datagram larger than the specified buffer size is received by a Linux system, it will be dropped. Windows, however, will receive these datagrams, but it will influence data retrieval. Any system calls that retrieve data from the buffer in which datagrams are stored will only return a number of bytes less than or equal to the buffer size, requiring multiple calls to retrieve all the data. [17].
- **Processor.** The processor used can also influence the behavior of an application. This is frequently evidenced through the different floating point behaviors a processor may exhibit [18]. In addition, bugs are fairly common in processors and will cause variances, as will differences in interpreting how to execute complex instructions [19].

In this work, we chose to focus on operating system, file system, and network environmental issues. These issues are readily visible in data returned by the system calls an application makes. We took advantage of this in our implementation of SEA as CrashSimulator by intercepting and manipulating system call results. As we will discuss, this allowed us to strike a good balance between a higher level approach, such as hooking library functions, and a lower level approach like directly altering memory values. In our evaluation we take a deeper look at anomalies from these three categories in order

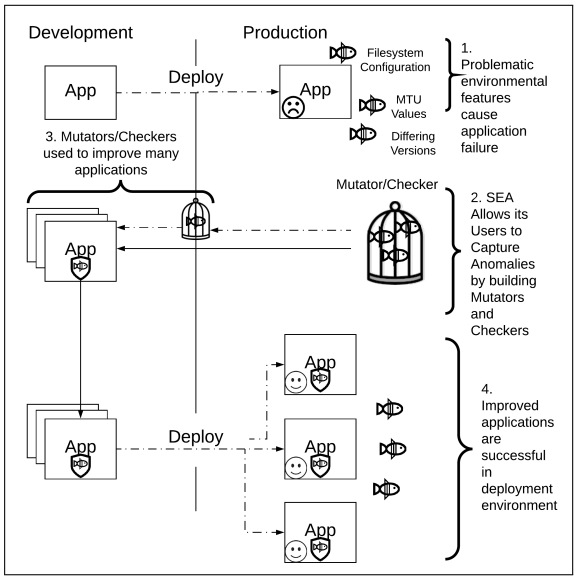

Figure 1: Using SEA allows developers to capture features that make an environment problematic and use them to prevent future applications from falling victim to the bugs of the past.

to assess the technique's effectiveness. We do not consider processor-based environmental differences as bugs related to those are being handled by other work [20].

III. The SEA Technique

The Simulating Environmental Anomalies (SEA) technique offers a methodical way to capture, store, and utilize the insights gleaned from previous failures in a given environment. In this section, we offer a high-level look at how SEA works, as illustrated in Figure 1. This is followed by a more in-depth look at its primary operations, and the details of our concrete implementation, CrashSimulator.

A. SEA in a Half-shell

In brief, here is how the technique works. An application, A, is run in a particular environment and fails. In debugging the failure, we identify and "trap" the particular environmental feature (sec. III-B1), that caused the failure, which we call X. We verify that when X is present, the results of interactions with the environment are different, sometimes in a barely perceptible manner, and other times in a radically contrary manner. We call this difference ΔX. ΔX can be extracted, preserved, and later used in testing other applications thanks to a pair of components referred to as a mutator and a checker. The mutator, $mutX()$, is able to apply ΔX to the appropriate places in an application's interactions in order to simulate the presence of X. The checker, $checkX()$ describes how an application should respond once X has been encountered. To test if another application, B, also has a problem with X, $mutX(B)$ is used to apply ΔX to its interactions (sec. III-B2). $checkX(B)$ is then used to determine whether B has responded correctly to X(sec. III-B3). If the checker accepts B's behavior after X is simulated, we report it has been handled correctly. If the checker doesn't accept, we report that it has not responded correctly. Once created, these pieces act as the persistent medium in which the details of a given anomaly are stored.

Representing an anomaly in terms of mutators and checkers allows it to be easily reused to test other applications without per application effort. While an application's test suite is typically tightly coupled to the programming language and frameworks with which it was written, SEA's approach is agnostic to these features. This means anomalies that were useful in testing one application can be programmatically applied toward testing another. In this way, SEA is able to augment application-specific test suites by both decreasing the number of tests that must be manually constructed to cover environmental concerns and by offering the possibility of catching failures that had not been considered. These anomalies can be accumulated from many applications resulting in the ability to test new applications against an ever increasing set of problematic conditions.

B. Primary Operations

To take a closer look at how SEA functions, we divide the technique into its three primary operations. These are: identifying and trapping anomalies, mutating system call results, and checking application responses.

1) Identifying and Encoding Anomalies: Building a corpus of anomalies is an ongoing process that improves the technique's effectiveness by extending the set of problematic features it can simulate. Anomalies can be sourced in a number of ways, such as examining the failures of other applications in a target deployment environment, or by using other tools that can identify potentially problematic behavior in other domains [17], [21]. Another option is taking this information from public bug trackers, which is ideal if you wish to determine whether or not an application is vulnerable to a widely publicized bug.

The chosen anomalies are examined to determine how they change the results of system calls an application makes as compared to a normal execution. Once teased out, these differences delineate a set of modifications that must be made to an execution in order to simulate the chosen anomaly or anomalies. These details are used to construct both a mutator and a checker. The mutator encodes a description of when in execution an anomaly can be simulated, as well as details of how to conduct that simulation. The checker (or set of checkers) stores a characterization of how the application should respond. Describing anomalies in this fashion allows them to be recorded systematically and cataloged for future use.

As an example, consider an anomalous environment where access to a required file is denied because of the environment's file security configuration. With this anomaly, attempts to access the file, such as the `read()` system call, will fail with an error stating that access to the file is denied. The mutator derived from this issue would be constructed to identify similar accesses as opportunities to insert the anomaly by making these accesses return "access denied". As described in III-B3, an associated checker would be built to examine

the application's behavior after a simulation and assess its correctness. Preserving the details of anomalies like this in the form of mutators and checkers allow them to be easily used to test responses of future applications.

Constructing new checkers and mutators is a creative process not unlike writing a unit test. The writer must understand both the cause of misbehavior in a deployment environment and how it is visible in the results of the system calls an application makes. Once they have this understanding, the actual construction boils down to building state machines that recognize the required patterns of system calls. As a result, effort required by this process depends on the complexity of the anomaly being simulated and the proficiency of the writer with the above concepts.

2) Mutating System Call Results: Simulating an environmental anomaly requires specific interventions at the correct moments during an execution. These situations are identified with the help of the chosen anomaly's mutator. For our work, this means providing the mutator with the system calls an application makes so it can identify sequences indicative of such opportunities. At the appropriate time, the application's system calls are intercepted and the mutator's anomaly description is used to make the modifications necessary to simulate the anomaly. In the simplest case, simulating an anomaly only requires the modification of a single value (e.g. `read()` returning -1 rather than the number of bytes read). We even found use for a "null mutator" that performs no mutation, but gives other tools an opportunity to examine an execution. In more complex cases, large numbers of diverse system calls will need to be interdicted and altered in order to provide a correct simulation. The above file access scenario, for example, requires the modification of a single system call. Simulating something more complex, like an unreliable system clock, requires that all efforts to access the clock be modified to reflect the chosen aberration.

3) Checking an Application's Response: SEA relies on checkers to provide a flexible approach to assess the way an application behaves after it has encountered an anomaly. A checker models the behavior an application should undertake in response to a simulated anomaly. It looks for this behavior by examining an application's system calls before, during, and after simulation. The checker then reports whether the application has handled the anomaly correctly based on whether it observed the behavior for which it was built to look.

As an example, consider the "default checker" from CrashSimulator. It draws a conclusion based on whether or not the application has made an effort to respond to the anomaly. This determination is made based on the assumption that such a response will yield different program paths, and therefore different system calls. If the application does not alter its behavior, it has not correctly handled the anomaly. Alternatively, if the application does deviate, it is likely an action has occurred to handle the simulated condition. This simple yes or no approach is often sufficient to classify application behavior.

We explore this, and other checkers in our Evaluation section (Section VI) IV. Section IV-A1 demonstrates how to find bugs using the null mutator, while Section IV-A2 illustrates how a more complex mutator can simulate specific scenarios to see if unexpected problems emerge.

C. CrashSimulator: A Concrete SEA Implementation

In order to correctly implement SEA, CrashSimulator must provide a framework by which the checkers and mutators constructed by its users can be used to test an application using the anomalies they represent. Building this capability required making a few key design decisions. The first, as mentioned in Section 2, was choosing to operate at the system call level, rather than manipulating calls to library functions, memory accesses, or other points where we could influence an application's interactions. This allowed CrashSimulator to test applications written in any language that can execute Linux system calls – an important advantage as our goal is a tool that can test many applications without per application effort. Working with system calls is also a good fit for simulating the file system, network, and operating system anomalies in which we were interested. An application normally queries these entities using system calls so we simply had to return modified responses in order to simulate an anomaly. Finally, robust tooling in the Linux kernel made the interception and modification of system call results and side effects a fairly simple process, and the well-defined semantics of Linux system calls streamlined implementation.

In order to take advantage of our ability to simulate anomalies using system calls, we needed to ensure that an application would reliably make the same set of system calls, and reach the same simulation opportunities, with every execution. To this end, we employed a modified `rr` debugger [22] to record and replay applications. Our first modification permitted a `strace`-style system call recording to be output alongside `rr's` normal recording format, creating a complete log of an applications system call activity. We chose to make this modification because, while `rr`'s recording contains the system call information we need, it is stored in an opaque format that changes frequently between versions. Storing a copy of this information as a `strace` recording bypasses this shortcoming and offers the advantage of being human readable.

Our second modification parallelized our testing by allowing `rr` to generate copies of a running application when it reaches our target simulation opportunities. We refer to this new technique as *process set cloning* and illustrate it in Figure 2. The `rr` debugger manages, and can copy, the full set of processes underlying an application so that users can test debugging hypotheses without damaging the original. We extended this capability by liberating process set copies from `rr`. This means that given application Y consisting of processes a, b, and c (written as $Y(a, b, c)$), we can generate cloned sets $Y_1(a_1, b_1, c_1)$, $Y_2(a_2, b_2, c_2)$ ... $Y_n(a_n, b_b, c_n)$. Y_1, Y_2 ... Y_n can then be used to test different scenarios. These clones can be run in parallel because they do not actually execute system calls or modify the operating system state. Instead, they are simulated as described below. Additionally, they allow the original Y to continue its execution unhindered.

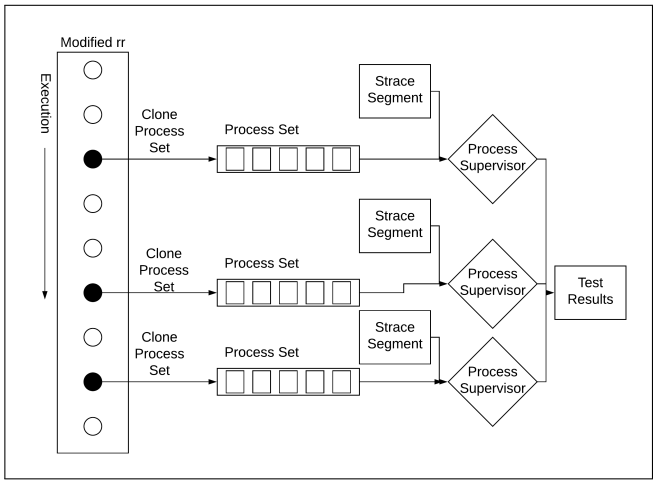

Figure 2: Diagram illustrating CrashSimulator's architecture. During the course of a single rr execution, clone process sets are generated at specific rr events. A CrashSimulator process supervisor attaches to these process sets and uses a strace-style system call listing to feed subsequent system call activity and inject unusual environmental conditions.

Once generated, the cloned process sets are used for testing by the CrashSimulator process supervisor. This supervisor is responsible for attaching to a process set via `ptrace` and servicing any system calls it makes. The data required to service these calls comes from the `strace` log generated by `rr`. Just before it takes over service of system calls, the process supervisor invokes the intended anomaly's mutator on the log in order to alter it to include the responses necessary to simulate an anomaly. Once these alterations are in place, the supervisor is able to impose an anomaly on the application by returning the now-modified responses from the log.

During this process, a corresponding checker monitors the application's behavior, allowing it to report on the correctness of the response. For example, testing could proceed in the following way. At a given simulation opportunity, when executing application Y, cloned sets Y_1 and Y_2 are generated and used to test two scenarios where a file access could fail:

1) For Y_1, access to the file is denied due to a permissions issue
2) For Y_2, access fails because of an I/O error

The simulations for both Y_1 and Y_2 are handled asynchronously and the results are recorded. This approach allows many tests to be run independently of one another, which lends a high degree of speed and parallelism to the testing process. At the same time, the original Y execution continues unhindered to the next simulation opportunity where this process repeats. Keeping the original execution intact, as opposed to destroying it by introducing an error state, avoids the penalty of having to restart a new execution for each test.

The prototype was built on rr version 5.2.0 running on a 32-bit Linux kernel distributed with Ubuntu 16.04 LTS. The modifications to rr described above were carried out in C++, and the CrashSimulator supervisor was implemented in 6260 lines of Python 2.7 code with a 2125 line C extension that allows it to interact with processes using the Ptrace API. This version of CrashSimulator is available at: https://github.com/pkmoore/rrapper.

IV. Evaluation

CrashSimulator was designed as a way to reduce the considerable effort required of developers to test an application against the environments it will encounter once deployed. Therefore, we needed to demonstrate its effectiveness in "the wild." To this end, we carried out two rounds of evaluation for CrashSimulator. We started by exposing a series of real-world applications to a library of collected anomalies in a laboratory environment. These tests, conducted by the research team, were followed by a user study in which undergraduate and graduate computer science students got a chance to use CrashSimulator to identify new environmental bugs in tests on applications of their choosing. We used the results from both these efforts to answer the following questions:

1) Is CrashSimulator able to identify bugs in real world applications? (Subsection IV-A)
2) What sorts of errors does CrashSimulator make? (Subsection IV-B)
3) Can CrashSimulator execute tests efficiently? (Subsection IV-C)

A. Is CrashSimulator able to identify bugs in real world applications?

The most crucial question to ask about the SEA technique and the tool we implemented is: can we use them to identify bugs caused by different types of problematic environmental features? To perform this evaluation we needed both a set of applications to test and a set of anomalies against which to test them. In order to show that CrashSimulator can find bugs in even the most widely deployed and well tested applications, we chose our test candidates from among those deemed "popular" by Debian's Popularity Contest [15], or those used by many Linux distributions, such as the ones provided by the GNU coreutils project.

To prove the breadth of CrashSimulator's capabilities we needed to test applications using a diverse set of exemplar anomalies. We already established that unusual filesystem and network situations can cause an application to fail. So we identified a number of candidate anomalies by examining public bug trackers, the source code of major portable applications, and the capabilities of bug finder tools like NetCheck [17] and CheckAPI [21]. From these candidates we chose to three test scenarios: a simple filesystem anomaly that relies on the null mutator; a more complex filesystem anomaly that simulates the presence of unusual file types; and a network anomaly that requires checking and mutating many values across a variety of system calls. Our experiences testing applications with these anomalies are detailed below.

1) The simplest case - A Filesystem Bug Found With the Null Mutator: In our first test we decided to evaluate the tool in its simplest possible configuration – employing the **"null mutator."** This mutator takes no action and simulates no anomalous conditions. It simply allows checkers to evaluate an application's behavior as it carries out a potentially-buggy

Application	Source Replaced	Preserve Xattrs	Preserve Timestamps	Copying Devices
`mv`	Correct	Correct	Correct	Correct
`mmv`	Correct	**Sec. Flaw**	**Time Loss**	Correct
`install`	Correct	**Sec. Flaw**	**Time Loss**	**Fill Disk**
`perl File::Copy`	Correct	**Sec. Flaw**	**Time Loss**	**Fill Disk**
`shutils`	**Corrupt**	**Sec. Flaw**	Correct	Correct
`rust`	Correct	**Sec. Flaw**	**Time Loss**	**Fill Disk**
`boost::copyfile`	**Corrupt**	**Sec. Flaw**	**Time Loss**	**Fill Disk**

Table I: Applications and libraries analyzed to determine whether or not they are able to correctly move a file from one device to another. Incorrect entries are either missing the needed check to ensure correct behavior or their implementation of the behavior was ineffective.

operation. We decided to look at how applications move files around on the filesystem. Though, in many cases, this operation can be handled atomically by the operating system through the `rename()` system call, in situations where the source file and destination file are on different storage devices, the application must perform the operation in a more complex way. This is a process that even well-tested applications frequently get wrong [23], [24], [25].

Method. CrashSimulator was configured to test each of the applications listed in Table I to see which might fail to correctly move a file from one disk to another. The tests were completed using just the Null Mutator and a set of checkers that model the correct steps involved in moving a file from one storage device to another. After examining several libraries and applications, we found that `mv` seemed to handle cases that other tools failed to consider. Therefore, we used its behavior as a template to create a set of checkers that evaluate whether or not the application correctly performs the following steps.

- *Confirm Source Not Replaced During Copy.* An application should make an effort to ensure that the file being copied is not replaced between the time it is initially examined and when it is opened for copying. If these checks are not performed, it means the application will proceed with its operations, making file corruption [24] possible.
- *Preserve Extended File Attributes.* When copying a file, an application should retrieve extended file attributes from the source file and, later, apply them to the destination file. Failure to do so can lead to security problems [26].
- *Preserve Timestamps.* It is important to ensure that time related metadata – such as creation, modification, and access times – are preserved when copying a file, as incorrect timestamps can impede applications like `make`, archival programs, and similar software [27], [28].
- *Copy Devices Correctly.* Files of this variety must be moved by creating a new device of the same type at the destination, instead of exhaustively reading and writing its contents. In our experience, applications that fail to perform this check can end up completely filling disks, exhausting available memory, or blocking forever, which can cause the system to become unresponsive.

Findings. CrashSimulator was able to identify whether an array of popular programs, including the standard libraries for the programming languages Python, Perl, and Rust, can correctly perform complex operations in anomalous situations. As can be seen from the results in Table I, each of the applications tested failed to perform one or more of the steps required to successfully complete a cross-device move. This is an unfortunate situation because a failure to perform any one of these steps can result in negative outcomes for the system as a whole. There was one case where our checkers made a false positive report which prompted efforts to improve it. This is discussed further in Section IV-B. Taken as a whole, our results demonstrate that even well-tested applications can miss one or more of the steps in a complicated operation.

2) A More Complex Case - The Unexpected File Types Mutator: If a more complex mutator is to be employed, then CrashSimulator will simulate anomalies as an application is executed. This simulation introduces problematic scenarios so an application's response can be evaluated. For an effective test of the tool's capabilities to find bugs created in this process, we needed to pick an anomaly that would arise during a common situation, such as when a Linux application retrieves and processes data from a file. Linux supports several special file types, including directories, symbolic links, character devices, block devices, sockets, and First-In-First-Out (FIFO) pipes. These special files use the same system calls as regular files (such as `read()` and `write()`), but they behave in very different ways. For example, `/dev/urandom` is a character device that produces an infinite amount of pseudo-random data when read. If `/dev/urandom` is provided to an application that relies on exhaustively reading the full contents of a file before processing, it will fill memory or disk space, and could crash the system [29]. Correct execution in these situations requires that applications examine the files, so they do not interact inappropriately with a given file type.

Method. Identifying these bugs involves changing an application's execution to induce its response to an unexpected file type. For example, the `sed` application, which modifies the contents of a text file according to a provided command string, could instead be provided a symbolic link, a directory, or a character device. CrashSimulator tests these alternatives by identifying the calls to `stat()`, `fstat()`, or `lstat()` that an application makes to examine the file, and changing the results to simulate one of the special file types. If the application responds to this injected information, then the special file might be handled correctly. On the other hand, if there is no alteration in the behavior of the application, the condition is not being handled correctly.

For each application, CrashSimulator was configured to simulate all of the non-standard file types. The values inserted into results of the application's `stat()`-like system calls are listed in Table II. A value of "–" indicates that this is the file type the application was expecting, as it matches what was provided when the application was initially recorded. A result of "✓" indicates that the application identified it was being provided with an unexpected file type and its execution differs, a signal that it was potentially handling the file type correctly. A result of "X" indicates that the execution never diverged from the trace being replayed, and thus failed to recognize the

presence of an unusual file type.

Findings. The data in Table II show that of the 84 cases we tested, CrashSimulator identified 46 bugs, predicted 25 instances of correct application behavior, and made 13 errors, the majority of which were false negatives. We discuss the causes of these errors in Section IV-B. The frequency of failed executions in our results indicates that many applications assume they will only be used to process regular files. When this assumption does not hold, execution results can be hard to predict. In many cases a denial of service condition occurs in the form of the application "hanging," as it attempts to incorrectly process the file. This is typically the result of an application blocking forever as it waits for a `read()` call to retrieve non-existent data from an empty FIFO, or an application attempting to read in and process an "infinitely large" file. This situation is particularly dangerous as it can eventually fill all available memory or disk space [30].

In order to confirm the accuracy of CrashSimulator's assessment, we manually exposed the applications listed in Table II to each of the unusual file types to get an idea of how they would respond. This allowed us to identify cases where CrashSimulator made errors and note this information in Table II. Further, we include descriptions of a subset of the applications' behaviors in Table III. These tables serve to document the accuracy of the tool's evaluation of application behavior and illustrate what misbehavior occurs when applications are actually exposed to problematic scenarios.

3) Beyond Filesystem Bugs - Poorly Configured Network Timeouts: CrashSimulator is not limited to identifying filesystem-based bugs. The third anomaly we examined involves an application's behavior when it attempts to communicate over a network with extremely long (on the order of minutes) response times. At a low level, applications retrieve data from a network socket by waiting for data to be available and then reading it. However, this approach needs to be able to handle a situation where communication takes too long and should time out.

Method. CrashSimulator can detect whether an application correctly times out when communications take too long by employing the null mutator and a network timeout checker. The latter can determine if the application makes any effort to configure its network communications with a timeout value. This is done by examining the presence or absence of `setsockopt()`, `poll()` and `select()` calls, as well as the timeout values that may have been passed to them. Applications that do not set the timeout are subject to the operating system-defined protocol timeout value. CrashSimulator is able to take this analysis a step further by employing a Long Network Response Time mutator that manipulates the results of all time-returning calls, simulating an execution where close to the maximum timeout value occurs, without actually spending any time waiting.

An application's failure to time out responsibly is not just an inconvenience. Attackers can take advantage of this flaw to consume resources and potentially cause a denial of service situation. This failure was exploited by the slowloris [31] tool to enhance the ability of a small number of computers to prevent access to vulnerable web servers by opening and maintaining connections for extremely long periods of time. As these servers could only handle a set number of connections due to resource constraints, legitimate traffic was easily crowded out by the attackers. Additionally, similar attacks can be used to indefinitely delay security updates to clients, leaving them vulnerable to compromise [32]. We used CrashSimulator to determine which applications and libraries from a selection based on Debian's ratings [15] could be vulnerable to this sort of attack.

Findings. As Table IV shows, all of these applications were vulnerable to this anomaly, and in some cases, timeouts took hours to resolve. What's more, in the vast majority of cases, the problem occurs because the application makes no effort to specify a timeout value. This means an attacker can transmit one byte of data per timeout period (per Linux's value of 19 minutes for TCP sockets), allowing them to keep the application alive instead of quitting.

4) Bugs Found By Participants: Because the above tests were carried out by CrashSimulator's developers, who necessarily have a high degree of expertise in its operation, we felt it prudent to ensure the tool was useful to outside developers as well. To investigate this angle, we conducted a user study with 12 undergraduate and graduate students with varying backgrounds. Study participants found a total of 11 bugs using CrashSimulator. Of these bugs, nine were found using the "Unusual Filetype" mutator. Five of these bugs have since been reported to the appropriate maintainers, and three of these reports included patches that correct the bug built by the reporting student.

These results are important because they confirm users, other than the original development team, can use the tool to find bugs in real world applications. Participants commented that narrowing the source of a bug down to a particular sequence of system calls was helpful in identifying the area of code responsible for the bug – a feature that decreased the time required to produce a fix. Though observation of study participants showed that familiarity with operating systems concepts made it easier to work with CrashSimulator, those without this background were still able to identify bugs using the built in anomalies.

On a less positive note, the study did reveal some shortcomings of the tool. First, it became clear that the tool does not have a clear mechanism for determining which application behaviors constitute a "bug." For example, an application's developer may have intended that an application processing an "infinitely long" file should run continuously until killed by an outside command. Therefore, that behavior should not be classified as a bug. Second, it demonstrated that simply reporting that an application did or did not change its behavior in the presence of an anomaly may not provide sufficient data to identify a bug. The results indicating the presence of a bug must be clear to the user. Both of these issues are being corrected by improving the tool's outputs. By more clearly describing the nature of a given result, users can have a better

Application	Condition Tested	Regular File (IFREG)	Directory (IFDIR)	Character Device (IFCHR)	Block Device (IFBLK)	Named Pipe (IFIFO)	Symbolic Link (IFLNK)	Socket File (IFSOCK)
`Aspell`	Dictionary File	–	X: *FP*	✓: *FN*	X	X	X	X
`Aspell`	File being checked	–	X: *FP*	✓	X	X	X	X
`gnu-gpg`	secring.gpg	–	X	X	X	X	X	X
`vim`	File being opened	–	✓: *FN*	✓: *FN*	✓: *FN*	✓: *FN*	✓: *FN*	X
`nano`	File being opened	–	✓	✓	✓	X: *FP*	X: *FP*	X: *FP*
`sed`	File being edited	–	✓	X: *FP*	X	X	X	X
`wc`	File being checked	–	✓	X	X	X	X	X
`du`	Directory being checked	✓	–	✓	✓	✓	✓	✓
`install`	File being installed	–	✓	X	X	X	✓	X
`fmt`	File being formatted	–	X	✓	X	X	X	X
`od`	File being dumped	–	✓	✓	X	X	X	X
`ptx`	File being read	–	✓	✓	✓	✓	✓	✓: *FN*
`comm`	Second file being compared	–	✓	✓	X	X	X	X
`pr`	File being read	–	✓	X	X	X	X	X

✓= CrashSimulator predicts application will recognize anomaly
X = CrashSimulator predicts application will fail to recognize anomaly
– = File type expected by the application

Table II: Applications tested for their handling of unexpected file types. A result of "✓" indicates that the application identified the presence of an unusual file and responded in some fashion. A result of "X" indicates that the application failed to recognize the presence of an unusual file and attempted to process it. Cases where CrashSimulator made an error are noted by FP (False Positive) or FN (False Negative)

Application	Directory (IFDIR)	Character Device (IFCHR)	Block Device (IFBLK)	Named Pipe (FIFO)
`wc`	Error: Is a Directory	hangs	slowly process file	Hangs
`install`	Error: Omitting Directory	Fills disk	slowly copies file	Hangs
`fmt`	No output	hangs	garbage output	Hangs
`od`	Error: read error	hangs	No output	Hangs
`ptx`	Error: Is a Directory	fills disk	garbage output	Hangs
`comm`	Error: Is a Directory	hangs	garbage output	Hangs
`pr`	Error: Is a Directory	hangs	garbage output	Hangs

Table III: Responses of a sample of coreutils applications when exposed to anomalous conditions. The character device used was the infinite-length `/dev/urandom`.

Application	**Analysis Result**
`wget`	Overly long timeout supplied to `select()`
`ftp`	No `poll()` or `select()`, no timeout set
`telnet`	`select()` specifies no timeout
`urllib http`	No `poll()` or `select()`, no timeout set
`urllib ftp`	No `poll()` or `select()`, no timeout set
`ftplib`	No `poll()` or `select()`, no timeout set
`httplib`	No `poll()` or `select()`, no timeout set
`requests`	No `poll()` or `select()`, no timeout set
`urllib3`	No `poll()` or `select()`, no timeout set
`python-websocket-client`	No `poll()` or `select()`, no timeout set

Table IV: Applications tested for their handling of extremely slow response times from the host with which they are communicating

idea if, and why, they should be concerned.

B. What Sorts of Errors does CrashSimulator Make?

Like other testing tools, CrashSimulator occasionally makes mistakes. Any such mistakes can undermine a developer's confidence in a tool, and thus one of our goals is to minimize them. In this section, we discuss situations where CrashSimulator made either false positive or false negative reports. A false positive report means the tool reports a failure when the application actually handles an anomaly correctly. On the opposite side, false negative reports happen when the tool indicates an application handles an anomaly when, in reality, it does not.

False Positives. The primary source of false positives in CrashSimulator is an application responding to an anomaly with a different sequence of system calls than was expected by the checker. Once identified, CrashSimulator's approach allows these situations to be easily corrected. This is similar to a situation where an application's test suite has a test with an over-constrained "oracle" (e.g. an oracle that ensures a returned value is > 0 when, in reality, 0 is acceptable).

We encountered this situation when testing applications that used GNOME's `glib` file handling functions. When an application makes use of these facilities to move a file across storage devices, the library itself correctly performs a file move operation. When we used CrashSimulator with checkers that expected a call to `read()` and `write()` for a cross-device move, we got reports stating that the application *did not* perform the system calls necessary to correctly move a file. By manually examining a system call trace, we found that, while `glib` correctly performs the requested move operation, it does so using alternative system call sequences. Rather than using a sequence of `read()` and `write()` calls, as our checker expected, `glib` creates a pipe and uses the `splice()` system call to copy the contents out of the source file, through the pipe, and into the destination file.

Fortunately, as soon as issues like this are discovered, CrashSimulator's checkers can be modified to include the alternative sequence. Given the above example about moving files, consider the mapping from high level "operation" to the set of system calls that can implement it in Table V. Each of the steps in the operation map to a small number of system calls. In situations where two system call sequences can correctly implement the same operation, CrashSimulator simply runs two checkers in parallel and accepts the execution if either detects the expected sequence.

False Negatives. False negatives are a fact of life in testing because of missing test cases or under-constrained checks of result correctness. CrashSimulator occasionally registers false negative reports when an application changes its behavior, but does not handle an anomaly correctly. In our evaluation we encountered ten such reports, four occurring with one application, `wc`. Because these reports were generated when using the default checker, they can be addressed by using a more elaborate checker that performs a detailed analysis of the application's post-simulation behavior. Such a checker would

Operation	Potential System calls
Examine source file	stat64(), lstat64(), fstat64()
Examine destination file	stat64(), lstat64(), fstat64()
Open source file	open()
Read contents of source file	read(), splice() with a pipe
List source file's extended file attributes	listxattr(), llistxattr(), flistxattr()
Read contents of source file's extended file attributes	getxattr(), lgetxattr(), fgetxattr()
Open destination file	open(), optionally unlink() the file first
Write contents to destination file	write(), splice() with a pipe
Apply extended file attributes to destination file	setxattr(), lsetxattr(), fsetxattr()
Apply proper timestamps to destination file	utimens(), futimens()
Apply proper permissions to destination file	chmod(), open() with a modeline specified
Close the source file	close()
Close the destination file	close()

Table V: Each step of a successful cross-disk file move operation mapped to the system call or calls that can implement it

compare this behavior to a model of a correct response to that anomaly. Creating this checker requires that a user know what a "correct response" looks like. This "known good" behavior can be found by looking at standards and documentation that describe best practices for handling an anomaly in a given environment, or by examining how applications that correctly deal with the anomaly do so. Consider the case where a `close()` system call fails. Retrying the call may not be the correct action, depending upon the environment in question. SEA can be used to determine if an application has handled the failure correctly by examining post-simulation communications in detail, and taking into account the correctness of retrying the call.

C. Can CrashSimulator execute tests efficiently?

One key attribute of successful testing tools is that they are able to complete their tests in a timely manner. If a tool takes too long, users will be less likely to run it. To address this concern, we evaluated CrashSimulator's performance in order to determine whether or not it was able to complete its test executions in an acceptable time frame.

Method. To answer the question of performance, we examined the completion times for executions of the specified application in both native conditions, and under CrashSimulator configured to test using the "Unusual File Types" anomaly discussed earlier. Table VI shows these results.

Findings. Overall, the performance running an application under CrashSimulator is around two orders of magnitude slower than the original program executed without it. As can be seen in Table VI, around 20% of this slow down is related to the additional overhead of `rr`'s replay. An additional portion is caused by the need to spin up Python's interpreter before testing can begin. In most cases this slowdown is somewhat mitigated by CrashSimulator's ability to process tests asynchronously. In other situations, it will be more efficient than running the program natively, as `rr's` replay does not require actual execution of most system calls. This means that CrashSimulator avoids the system call overheads, such as I/O. Even without these improvements, however, CrashSimulator's performance cost is more than manageable when the value it provides is taken into account. The cumulative runtime required to execute the tests required to produce the results listed in Table II is around 90 seconds.

Application	Native Execution Time	Initial Recording Time	CrashSimulator Replay Time	rr Replay Time
`wc`	0m0.007s	0m0.473s	0m0.668s	0m0.112s
`fmt`	0m0.007s	0m0.321s	0m0.707s	0m0.111s
`od`	0m0.036s	0m0.317s	0m0.689s	0m0.101s
`ptx`	0m0.008s	0m0.352s	0m0.769s	0m0.087s
`comm`	0m0.132s	0m0.371s	0m0.776s	0m0.141s
`pr`	0m0.135s	0m0.888s	0m1.017s	0m0.141s

Table VI: This table contains the times required to execute several coreutils applications under different conditions. Native execution time is the time required to execute the application from the command line while CrashSimulator Replay time is the time required to execute it with the tool, using the null mutator with no checker configured. Initial recording time is the time required to create an initial recording with CrashSimulator. rr replay time is the time required to replay an application using rr without CrashSimulator and is included to demonstrate the additional overhead added by CrashSimulator's other components.

V. Related Work

In constructing SEA and CrashSimulator, we examined a number of previous studies in bug detection, data mining, the influence of environment on application behavior, and protocols for checking program responses to anomalous situations. This section summarizes a few earlier initiatives in each of these areas and discusses their relevance to the development of our tool.

Static analysis. Tools based on static analysis techniques, such as abstract interpretation, model checking, and symbolic execution, have been successfully used to detect bugs resulting from incorrect API usage. Examples of these tools include SLAM [33], [34] and, more recently, CORRAL [35] for conformance checking of Windows device drivers against the Windows kernel API, FindBugs [36] for detecting API usage bugs in Java programs, FiSC [37] for finding bugs in TCP implementations, and the Explode system [38] for detecting crash recovery bugs in file system implementations. Likewise, INFER has found success at Facebook in analyzing units of code as they are committed [39]. Unlike CrashSimulator, these approaches depend on the availability of source or byte code.

Static analysis techniques have also been used to detect portability issues related to what happens when an application encounters different versions of the external components on which it depends [40], [41]. Like CrashSimulator, these techniques address the application's interactions with its environment. However, these studies were focused on proving that the environments behaved as expected. CrashSimulator is only interested in the application's response to anomalies.

API protocol mining. Extensive work has been done on mining source code to learn API protocols and use them to detect common usage violations, such as missing method calls (see, e.g., [42], [43], [44], [45], [46]). These techniques primarily target object component interactions, rather than system calls to generate test suites. However, the techniques explored in these works could be used to mine system call

patterns in source code. This will enable researchers to find checkers that identify incorrect responses to environmental anomalies. So far, we have specified these checkers manually for CrashSimulator. Data mining techniques could be adapted in the years ahead to reduce the manual effort required.

Tracing and log mining. Similar to API protocol mining, substantive work has been done using log files to detect anomalies [47], [48], [49], [50] and to aid in program understanding [51], [52], [53]. For example, CheckAPI [21] and NetCheck [17] both use system call traces to diagnose an application's violations of cross-platform portability. CrashSimulator in some ways takes these techniques a step further by using system call traces to test responses to known environmental anomalies. This more active approach can expose bugs that are invisible to passive log mining.

Environmental influence and cross-platform portability. Negotiating the influence of the environment in which an application is deployed has been investigated from several perspectives. One approach, referred to previously, is to build "portable" systems that have cross-platform capabilities. Using wrapper subroutines can enable cross-platform portability of APIs [54] through system call delegation. These wrappers can be automatically generated through techniques like system call interposition [55], a technique that can also be used to detect and prevent security violations [56], [57]. Other works that target complementary classes of portability problems include the detection of configuration-related bugs [58], [59], [60], [61], [62] and cross-browser incompatibilities for web applications [63], [40], [64], [65], [66]. Static analysis tools, such as those mentioned earlier in this section, have also been applied to detection of differences between versions of external components.

Such studies have helped to define the influence of an environment on applications and thus provide a background to our work. Our tool applies these fundamental ideas to testing an application's response to anomalies.

Testing exception handling and conformance. A few researchers have developed testing techniques aimed at checking whether programs respond appropriately to anomalous situations. For example, Fu et al. introduced data flow testing techniques that require tests from the points at which exceptions are thrown to the points at which they are handled in Java code. The purpose is to discover whether programs respond correctly to exceptional situations anticipated by the programmer [67]. Koopman and DeVale developed a system to detect bugs in error handling code related to calls to POSIX functions [68]. Miller et al. cover the kernel as a source of unexpected program inputs and test applications by simulating them [69]. Other approaches to conformance checking of POSIX operations use model based testing [70], [71].

Unlike these approaches, CrashSimulator does not exclusively target error handling code or anomalies that only involve individual system calls. Additionally, it focuses on collecting and simulating problematic features of specific deployment environments. That said, such testing techniques can help identify additional anomalies to add to our repository.

Smart Fuzzing. Smart fuzzers monitor and guide executions to trigger code paths that are unlikely to be reached otherwise [72], [73]. At this point, they make a binary decision about whether the application correctly handled the input or not. Typically, a determining factor will be an actual crash by the application. This is similar to SEA's use of checkers to decide whether to accept or reject a given test execution, based on whether or not it observed a specific system call behavior. In both cases, the tools can demonstrate whether a given code path handles a situation correctly, but cannot prove that the code path is robust against other problematic inputs. In the future, CrashSimulator could use smart fuzzing path exploration techniques to generate tests that reach relevant system calls and apply SEA at that point.

VI. Conclusion

As we have discussed, it is common for an application to fail upon deployment because of unexpected interactions with its environment. Although finding and eliminating faults in an application is a key concern for software developers, it is impractical to test it in every environment it will face. To address this problem, we developed *Simulating Environmental Anomalies* (SEA). SEA is beneficial for developers because it allows the effort spent debugging failures in a given environment to be preserved and reused programmatically to test whether future applications will also fail. As this process is repeated, a corpus of bug-causing aspects, known as "anomalies," along with mutators and checkers that characterize these anomalies, can be accumulated. In doing so, developers have an ever-increasing capability to test applications in situations that proved problematic in the past.

We built a concrete implementation of SEA called CrashSimulator that implements the technique by simulating environmental anomalies extracted from the system calls an application makes. Operating on system calls gives the tool a "universal" way to encode and inject anomalies. Consequently, a set of mutations can be collected from existing applications for use in testing others. Our evaluation of CrashSimulator has shown that this technique is effective at finding bugs in well tested software. In total, 65 new bugs were identified in popular applications. These bugs, if triggered in the wild, could lead to effects ranging from simple program hangs to security vulnerabilities and data loss.

Given that the technique has proven effective, future work to expand its use is warranted. This work includes developing a public repository of anomalies that can be applied to new or existing applications. We are also exploring opportunities to further automate the discovery process and improve the way anomalies are specified using a domain specific language. As this research evolves, we will focus on analyzing how an application attempts to recover from the anomalies. This would allow us to determine whether an application is correctly recovering from an error, or carrying out some incorrect response.

REFERENCES

[1] A. F. Athreya, Jagan, Gongloor, Prabhaker, "Application Quality Management." [Online]. Available: https://www.oracle.com/technetwork/oem/pdf/511991.pdf

[2] "Linux and glibc api changes," http://man7.org/tlpi/api_changes/#Linux-kernel.

[3] "Windows API: Detailed Analysis of Changes in WinAPI," https://abi-laboratory.pro/index.php?view=winapi.

[4] "Musl libc: Functional differences from glibc," https://wiki.musl-libc.org/functional-differences-from-glibc.html.

[5] "Ext4 Disk Layout," https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[6] "Apple File System Guide," https://developer.apple.com/library/content/documentation/FileManagement/Conceptual/APFS_Guide/FAQ/FAQ.html.

[7] "Ntfs /mft structure how to set extended attributes,alternate data stream," https://milestone-of-se.nesuke.com/en/sv-basic/windows-basic/ntfs-filesystem-structure/.

[8] "When using NAT interface on Windows host, guest can't receive UDP datagrams larger than 8 KB," accessed Thursday 7th November, 2019, https://www.virtualbox.org/ticket/12136.

[9] "Nmap network scanning: Os detection," https://nmap.org/book/man-os-detection.html.

[10] "Vmware nat interface failure," https://communities.vmware.com/thread/528349.

[11] ClusterHQ, "Application testing survey reveals legacy testing processes no longer cut it," https://clusterhq.com/2016/11/03/devops-testing-survey/, Nov. 2016.

[12] W. Wong, "Write-Once, Debug Everywhere," May 2002, http://electronicdesign.com/article/embedded-software/write-once-debug-everywhere2255.

[13] A. Page, K. Johnston, and B. Rollison, *How We Test Software at Microsoft®*, ser. Developer Best Practices. Microsoft Press, 2009. [Online]. Available: https://books.google.com/books?id=mFketWnCDhoC

[14] "/usr/lib/i386-linux-gnu," https://support.microsoft.com/en-us/help/4056892/windows-10-update-kb4056892.

[15] "Debian Popularity Contest," http://popcon.debian.org.

[16] "The Open Group Base Specifications Issue 6," 2004, http://pubs.opengroup.org/onlinepubs/009695399/functions/unlink.html.

[17] Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos, "Netcheck: Network diagnoses from blackbox traces," *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI14), USENIX*, 2014.

[18] K. R. Ghazi, V. Lefevre, P. Theveny, and P. Zimmermann, "Why and how to use arbitrary precision," *Computing in Science Engineering*, vol. 12, no. 3, pp. 5–5, May 2010.

[19] Fog, Agner, "The microarchitecture of Intel, AMD and VIA CPUs An optimization guide for assembly programmers and compiler makers," 2017, http://www.agner.org/optimize/microarchitecture.pdf".

[20] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern, "Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 405–418. [Online]. Available: http://doi.acm.org/10.1145/3173162.3177156

[21] J. Rasley, E. Gessiou, T. Ohmann, Y. Brun, S. Krishnamurthi, and J. Cappos, "Detecting latent cross-platform api violations," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015, pp. 484–495.

[22] "rr Debugger," https://rr-project.org/.

[23] "rename of dirs across devices produces confusing copy error," https://bugs.php.net/bug.php?id=54097.

[24] "Race conditions in shutil.copy, shutil.copy2, and shutil.copyfile," https://bugs.python.org/issue15100.

[25] "Prevent copying a directory into itself," https://github.com/jprichardson/node-fs-extra/issues/83.

[26] "macOS Code Signing In Depth," https://developer.apple.com/library/content/technotes/tn2206/_index.html.

[27] "Nautilus not preserving timestamps," https://bugs.launchpad.net/ubuntu/+source/glib2.0/+bug/215499.

[28] "sudo: timestamp too far in the future," https://bugs.launchpad.net/ubuntu/+source/sudo/+bug/76639.

[29] "Attacks on package managers," https://www2.cs.arizona.edu/stork/packagemanagersecurity/otherattacks.html.

[30] J. Cappos, J. Samuel, S. Baker, and J. Hartman, "A look in the mirror: Attacks on package managers," in *The 15th ACM Conference on Computer and Communications Security (CCS '08)*. New York, NY, USA: ACM, 2008, pp. 565–574.

[31] "Apache HTTP DoS Tool Released," https://isc.sans.edu/diary/Apache+HTTP+DoS+tool+released/6601.

[32] Justin Cappos and Justin Samuel and Scott Baker and John Hartman, "Package Management Security," Department of Computer Science, University of Arizona, Tech. Rep. TR–08–02, 2008.

[33] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with slam."

[34] T. Ball and S. K. Rajamani, "The slam project: debugging system software via static analysis," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '02. New York, NY, USA: ACM, 2002, pp. 1–3. [Online]. Available: http://doi.acm.org/10.1145/503272.503274

[35] A. Lal and S. Qadeer, "Powering the static driver verifier using CORRAL," in *SIGSOFT FSE 2014*. ACM, 2014, pp. 202–212. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635894

[36] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *OOPSLA 2004*. ACM, 2004, pp. 132–136. [Online]. Available: http://doi.acm.org/10.1145/1028664.1028717

[37] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *NSDI*, 2004.

[38] J. Yang, C. Sar, and D. Engler, "Explode: a lightweight, general system for finding serious storage system errors," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 131–146. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298469

[39] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods*, K. Havelund, G. Holzmann, and R. Joshi, Eds. Cham: Springer International Publishing, 2015, pp. 3–11.

[40] D. Silakov and A. Smachev, "Improving portability of linux applications by early detection of interoperability issues," in *Leveraging Applications of Formal Methods, Verification, and Validation*. Springer, 2010, pp. 357–370.

[41] "Java API compliance checker - ISP_RAS," accessed September 5th, 2014 http://ispras.linuxbase.org/index.php/Java_API_Compliance_Checker.

[42] L. Mariani, S. Papagiannakis, and M. Pezze, "Compatibility and regression testing of COTS-component-based software," in *ICSE*, 2007.

[43] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," *Autom. Softw. Eng.*, vol. 18, no. 3-4, pp. 263–292, 2011. [Online]. Available: http://dx.doi.org/10.1007/s10515-011-0084-1

[44] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE Computer Society, 2012, pp. 925–935. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2012.6227127

[45] M. Monperrus and M. Mezini, "Detecting missing method calls as violations of the majority rule," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, pp. 7:1–7:25, 2013. [Online]. Available: http://doi.acm.org/10.1145/2430536.2430541

[46] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *ICSE 2016*. ACM, 2016, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884782

[47] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *DSN*, 2002.

[48] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira, "Multi-resolution abnormal trace detection using varied-length N-grams and automata," in *International Conference on Automatic Computing*, 2005.

[49] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *SOSP*, 2009.

[50] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," *ATC*, 2010.

[51] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *ASPLOS*, 2010.

[52] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models." in *FSE*, 2011.

[53] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring Models of Networked Systems from Logs of their Behavior with CSight," in *ICSE*, 2014.

[54] T. T. Bartolomei, K. Czarnecki, and R. Lämmel, "Compliance testing for wrapper-based API migration," in *ISSTA*, 2012.

[55] P. J. Guo and D. Engler, "Cde: using system call interposition to automatically create portable software packages," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 21–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002181.2002202

[56] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Secur.*, vol. 6, no. 3, pp. 151–180, Aug. 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298081.1298084

[57] A. Acharya and M. Raje, "MAPbox: Using parameterized behavior classes to confine untrusted applications," in *USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2000, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251306.1251307

[58] A. M. Memon, A. A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan, "Skoll: Distributed continuous quality assurance," in *ICSE*, 2004, pp. 459–468.

[59] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 45–54. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007519

[60] S. Fouché, M. B. Cohen, and A. Porter, "Incremental covering array failure characterization in large configuration spaces," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 177–188. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572294

[61] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann, "Toward variability-aware testing," in *International Workshop on Feature-Oriented Software Development*, 2012, pp. 1–8.

[62] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," in *ICSE*, 2014, pp. 907–918.

[63] S. R. Choudhary, H. Versee, and A. Orso, "Webdiff: Automated identification of cross-browser issues in web applications," in *ICSM*, 2010, pp. 1–10.

[64] S. R. Choudhary, "Detecting cross-browser issues in web applications," in *ICSE*, 2011, pp. 1146–1148.

[65] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 561–570. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985870

[66] V. Dallmeier, B. Pohl, M. Burger, M. Mirold, and A. Zeller, "Webmate: Web application test generation in the real world," in *ICST 2014 Workshops Proceedings*. IEEE Computer Society, 2014, pp. 413–418. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2014.65

[67] C. Fu, A. Milanova, B. G. Ryder, and D. Wonnacott, "Robustness testing of java server applications," *IEEE Trans. Software Eng.*, vol. 31, no. 4, pp. 292–311, 2005. [Online]. Available: http://dx.doi.org/10.1109/TSE.2005.51

[68] P. Koopman and J. DeVale, "The exception handling effectiveness of posix operating systems," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 26, no. 9, pp. 837–848, 2000.

[69] Z. Miller, T. Tannenbaum, and B. Liblit, "Enforcing murphy's law for advance identification of run-time failures," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 203–208. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/miller

[70] F. Dadeau, A. Kermadec, and R. Tissot, "Combining scenario- and model-based testing to ensure posix compliance," in *Proceedings of the 1st international conference on Abstract State Machines, B and Z*, ser. ABZ '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 153–166.

[71] E. Farchi, A. Hartman, and S. S. Pinter, "Using a model-based test generator to test for standard conformance," *IBM Systems Journal*, vol. 41, no. 1, pp. 89–110, 2002.

[72] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, March 2011, pp. 427–430.

[73] ——, "A taint based approach for smart fuzzing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 818–825.