Detecting Latent Cross-Platform API Violations

Jeff Rasley[1] Eleni Gessiou[2] Tony Ohmann[3][2] Yuriy Brun[3] Shriram Krishnamurthi[1] Justin Cappos[2]
[1]Brown University Providence, RI USA 02912
[2]New York University New York, NY USA 10003
[3]University of Massachusetts Amherst, MA USA 01003
{jeffra, sk}@cs.brown.edu, eleni.gessiou@gmail.com, {ohmann, brun}@cs.umass.edu, jcappos@nyu.edu

***Abstract*—Many APIs enable cross-platform system development by abstracting over the details of a platform, allowing application developers to write one implementation that will run on a wide variety of platforms. Unfortunately, subtle differences in the behavior of the underlying platforms make cross-platform behavior difficult to achieve. As a result, applications using these APIs can be plagued by bugs difficult to observe before deployment. These portability bugs can be particularly difficult to diagnose and fix because they arise from the API implementation, the operating system, or hardware, rather than application code.**

This paper describes CheckAPI, a technique for detecting violations of cross-platform portability. CheckAPI compares an application's interactions with the API implementation to its interactions with a partial specification-based API implementation, and does so efficiently enough to be used in real production systems and at runtime. CheckAPI finds latent errors that escape pre-release testing. This paper discusses the subtleties of different kinds of API calls and strategies for effectively producing the partial implementations. Validating CheckAPI on JavaScript, the Seattle project's Repy VM, and POSIX detects dozens of violations that are confirmed bugs in widely-used software.

I. Introduction

Cross-platform APIs are critical for software portability and reuse. Well-known examples include the Portable Operating System Interface (POSIX) and Java, famously advertised as "write-once, run-anywhere" (WORA) [91]. Similarly, JavaScript and the document-object model (DOM) offer a browser- and architecture-independent programming interface to the Web, and Eclipse and Emacs offer cross-platform extensible editors. Cross-platform APIs are at the heart of the software engineering process, providing valuable detail-hiding abstractions. Developers often use them to create higher-level abstractions, including other cross-platform APIs, such as layering a GUI package on top of the JVM, on top of POSIX.

Unfortunately, API implementations often inadvertently breach the promise of a cross-platform API, forcing applications to be altered to handle platform differences [17], [18], [36]. This has led developers to re-coin the catchphrase as "write-once, debug everywhere" [40], [104]. Portability issues arise because API implementations — which we refer to as *libraries* to distinguish from the common *interface*, or API — behave differently across platforms, often in subtle and undocumented ways. Sometimes these behavioral variations are caused by reliance on other purportedly cross-platform APIs that fail to fulfill their promise. Even when a library successfully provides cross-platform coverage, portability can change over time because the underlying systems themselves change. It is unreasonable to expect developers to double-check all the documented — and undocumented — behavior of each library, including operating systems, browsers, etc., that may affect the application's behavior every time new versions of those libraries are released.

The result of cross-platform failures is frustrating to programmers: applications using libraries behave in unpredictable ways. This is often discovered only after deployment, and sometimes the fix cannot even be implemented in the application itself. As this paper shows, even well-known and long-developed APIs (such as POSIX and JavaScript) continue to exhibit serious portability bugs.

In addition to causing difficult-to-reproduce failures, variations in libraries can have security implications. An example of this is the `SO_REUSEADDR` socket option. On Linux kernels, this option allows the reuse of local addresses, except when there is an active listening socket bound to the address [84]. This option is widely used, e.g., to allow applications to reuse addresses in the `TIME_WAIT` state. However, on Windows, this allows multiple sockets to actively listen on the same address, with indeterminate results. For example, if multiple sockets are listening for TCP connections on the same address and port, either could accept a connection. A malicious application can use this option to bind to sockets already in use by other services to deny access to or impersonate those services [97]. These differences in `SO_REUSEADDR` implementations have forced a large number of major software projects to write platform-specific code to prevent application failures and to mitigate potential security risks, including Python [87], Twisted [81], OpenSSH [73], ntpd [102], Mozilla [79], Java [86], Ruby [92], Cygwin [45], Eclipse [15], Mono [70], Go [69], and OpenVPN [74].

Despite these issues, there has been relatively little research on helping the creators of cross-platform libraries and their users identify violations of cross-platform uniformity. We tackle this problem. Our approach finds violations in well-known, widely-used APIs, such as POSIX and JavaScript, with relatively little effort, and is designed to provide incremental benefits for incremental effort. Our approach improves on two alternatives — testing and virtualization — that are useful but have drawbacks we describe next.

Testing: Testing involves generating suites of test inputs, running them on multiple API implementations, and comparing the outputs. Each output inconsistency potentially uncovers a violation of cross-platform behavior. Developers have tried to verify portability by writing tens of thousands of test cases [29], [72] and by building huge test networks [27], [44]. While this approach works in principle and does catch violations before

978-1-5090-0406-5/15/$31.00 ©2015 IEEE

deployment, it can be very expensive and many bugs still arise in practice in deployed software [27], [101]. Further, testing cannot fully anticipate user configurations, and developers often use libraries in ways library developers did not anticipate [55]. Finally, because the APIs are not all deterministic, some latent failures may not manifest during testing.

Virtualization: Operating system virtualization [8] has many practical concerns: performance can suffer tremendously, commercial systems cannot be easily distributed, end-users have to work inside a virtual environment rather than in their traditional workspaces, etc. Furthermore, virtualization does not always succeed: bugs and limitations can cause the virtual machine to behave differently from both the host and guest OSes [109].

Many existing validation techniques are also insufficient for finding cross-platform violations. Static analysis often relies on source code, which is not available for many popular operating systems, libraries, and hardware. Traditional modeling techniques require abstracting away portions of the system to be tested so that one can reason about the correct behavior of the remainder, thereby masking the very parts that need checking. Parameter fuzzing can generate an enormous test suite, but cannot detect problems with the interactions between calls: fuzzing typically relies on crashes to detect errors, but many violations result in inconsistency, not crashes.

We propose **CheckAPI**, an approach to detect cross-platform violations in execution traces of library use. CheckAPI compares a library's trace induced by real API client behavior against a reference implementation, called a Partial Specification-based Implementation (PSI). A PSI is an API *implementation* in that it runs and produces answers. It is *specification-based* in that it focuses on correctness, as opposed to efficiency or utility. (For example, a PSI may implement a file-system in memory and not on disk.) Finally, a PSI can be *partial*, implementing only a part of the API. This enables developers to write PSIs quickly and to benefit immediately. Instead of creating one from scratch, a PSI can often be derived by modifying an existing library implementing the API. Checking library behavior against a PSI rather than an unmodified alternate library is more effective — and sometimes necessary — to handle methods that have state or are nondeterministic, as discussed in Section IV.

By checking actual API client behavior, CheckAPI avoids developers' estimates of application behavior. This is particularly helpful when APIs have state, because the exact sequence of steps is relevant, and it is difficult to identify all relevant sequences with offline testing. Most of all, exploiting the client helps CheckAPI find *latent* errors that escaped the pre-release testing process. Our evaluation shows that many such errors persist beyond testing.

This work was motivated by our interaction with multiple cross-platform projects, most notably with the developers of the Repy VM [16], [37] for the Seattle testbed [80], which provides cross-platform access to computational resources on heterogeneous devices such as servers, tablets, and phones. Repy was built using APIs such as POSIX, and initially had 212 cross-platform tests. However, as dozens of latent violations were encountered, the tests grew to 350, at which point the project contacted us for better ways to manage this problem. As we show, applying CheckAPI to Repy resulted in significant improvement of its cross-platform behavior.

The main contributions of this work are:

- The CheckAPI approach for detecting cross-platform API violations at runtime. CheckAPI is open-source and available for use to check API libraries: `https://checkapi.poly.edu`.
- The taxonomy of API calls on which CheckAPI relies, and how CheckAPI can exploit and support them to evaluate library conformance.
- The process of creating PSIs, showing how even relatively little effort can yield significant payoffs.
- PSIs for JavaScript, the Repy VM, and POSIX for finding cross-platform violations for applications using these libraries.
- An evaluation of CheckAPI's effectiveness and efficiency. CheckAPI identified six real errors not detected by the test suites of three production systems: POSIX, JavaScript, and the Repy VM. Our design and optimizations make CheckAPI efficient enough to be used at runtime, for example, checking JavaScript for violations while browsing the Web.

Section II discusses a taxonomy of API calls and Sections III and IV describe how CheckAPI works for these different kinds of calls. Section V describes the effort needed to create three PSIs, and Section VI evaluates CheckAPI. Section VII describes how CheckAPI can be extended to multi-threaded execution, Section VIII places our work in the context of related research, and Section IX summarizes our contributions.

II. Taxonomy of API Method Calls

CheckAPI detects cross-platform API violations by comparing behavior of a library and a PSI. This comparison requires a definition of API behavior. We define behavior of an API in terms of execution traces: sequences of the API method calls and their results. To detect conformance violations, CheckAPI compares the calls and results of a library to those of a PSI. However, for many calls, a direct comparison is insufficient because, for example, the calls may also alter internal state. To understand how CheckAPI compares traces, we first define three types of API calls and how they differ. Sections III and IV will describe how CheckAPI handles each type.

CheckAPI records library execution traces that consist of a list of *actions*: an API call with its arguments and its result. Figure 1 shows a sample trace from a JavaScript application. A call may have side effects that are not immediately observable but that can be observed via subsequent actions.

A *trace* is composed of an initial state and a sequence of actions. The sequence of actions can be captured by recording the actions of all method calls issued by an application. (We assume for now the library is invoked by a single-threaded application. Section VII discusses multi-threaded applications.)

Method *calls* can be classified into three categories:

- A **stateless** method call returns the same value (for the same arguments) independent of its location in the trace. Stateless calls' actions are completely defined by their

```
String.charAt('unload',         0)       →       'u'
String.toLowerCase('SCRIPT')        →        'script'
Array.indexOf(['h',    'e',    'y'],    'y')   →   2
RegExp.test('^-?\\d+(?:px)?$', 'i', '215px') → true
```

Fig. 1. A sample JavaScript application execution trace with array, string, and regular expression calls.

arguments and are independent of the initial state and prior actions. An example of a stateless method call is any call to `String.length()` in JavaScript.

- A **stateful-deterministic** method call returns the same value (for the same arguments) whenever it is executed within a given system state but may return a different value when executed within different states. Given the initial state and a sequence of actions executed before a stateful-deterministic call, there is exactly one possible result for the call. For example, a `read()` call at the start of a trace on an open, blocking file descriptor that refers to a local file is a stateful-deterministic call (assuming the call does not return `EIO` due to an error).
- A **nondeterministic** method call's results can differ depending on information not in the trace. This is common for calls that involve resources like the network. For example, a call to `gettimeofday()` is nondeterministic. This call returns data that cannot be predicted by the application and is likely to vary across executions. However, while results can vary, usually not all results are acceptable.

Some methods can exhibit complex behavior that fits into multiple of the above classifications. Some uses of the method may be stateless, others stateful-deterministic, and still others nondeterministic. For example, the POSIX `connect()` system call will respond to a negative (invalid) socket descriptor with `EBADF` regardless of previous calls. Thus, given a negative socket descriptor, the `connect()` call is stateless. If the `connect()` call is performed on a socket descriptor that is being listened on, the call will always return `EOPNOTSUPP`. In this situation, the `connect()` call acts in a stateful-deterministic manner, because it is dependent on prior actions. The `connect()` call can also be nondeterministic: when given valid arguments, it can succeed or fail to connect based on aspects of the environment not reflected in the initial systems state and prior actions.

Because CheckAPI handles each type of call differently, each call must be classified as stateless, stateful-deterministic, or nondeterministic. CheckAPI requires this is done while implementing the PSI, as Section V will describe. It is usually easy to classify these calls by noting the system state on which the implementation of that call depends. Observing the runtime behavior of calls, and the return values of multiple instances of the same call, can also help classify the call.

Next, Section III will describe CheckAPI with respect to the stateless calls, and Section IV will describe how CheckAPI handles the stateful deterministic and nondeterministic calls.

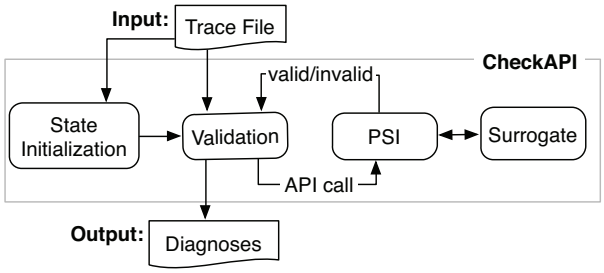


Fig. 2. CheckAPI architecture.

III. Stateless Deterministic Methods

Figure 2 shows the high-level CheckAPI architecture. CheckAPI captures application execution traces (Section III-A), and then uses a PSI (Section V) to detect violations (Section III-B) in the trace. This section describes CheckAPI with respect to just stateless deterministic actions, which are straightforward to handle by a PSI that uses a reference implementation, an alternate existing implementation, or some combination thereof.

A. Trace Capture

At runtime, CheckAPI captures execution traces. Each trace is a sequence of actions (API calls, with their arguments and return values) the applications makes (e.g., Figure 1). How the trace is captured is specific to the system. For instance:

- POSIX has several popular trace capture mechanisms, such as `ktrace` and `strace`. These and similar mechanisms capture OS system calls via `ptrace`, which guarantees that the calls are properly serialized.
- For JavaScript, we wrote a custom script to interpose on relevant API calls. This script modifies the prototype object for built-in objects such as `String`, `Array`, `Number`, `Boolean`, `Date`, `Object`, `Function` and `RegExp`. The script calls the underlying method and records the action. It can be loaded on a webpage with a tag in the HTML or automatically on all pages via a browser extension. We implemented such an extension for Chrome. This allows the user to browse normally, while checking for violations at runtime.
- Seattle's security layer functionality [16] includes a mechanism to collect traces from applications running inside the Repy VM.

B. Detecting Violations

Given a PSI and a trace, a validation component detects if an action results in a violation. Validation involves executing the trace in the PSI by simulating each recorded action, in order. For stateless (and also stateful-deterministic) calls, a difference between the action's call result and the PSI's call result implies a violation.

Many stateless calls are repeated more than once in a program's execution. For the top-10 Alexa sites [3], repeated stateless calls represent about half of the actions. To optimize verification, information about stateless calls can be memoized in the trace-gathering routine. If the action for a stateless call is not identical across multiple invocations, assuming the call

has been correctly identified as stateless, it is immediately a violation. API call entries are about 250 bytes each and are kept in a fixed-size cache with an LRU eviction policy. This technique significantly speeds up violation detection without reducing accuracy, as we show in Section VI.

IV. Stateful and Nondeterministic Methods

This section describes how CheckAPI handles stateful systems and nondeterminism. Whereas reusing an existing alternate implementation usually suffices for stateless deterministic methods, handling these features usually requires constructing a PSI (Section IV-A) with additional facilities to handle nondeterminism (Section IV-C) beyond what an existing reference implementation PSI offers. For example, a simple reference implementation PSI might erroneously report a violation for nondeterministic method calls, such as `gettimeofday()`, when the PSI and library results do not match exactly; in such cases, it is better to check for conformance to a pattern rather than for a precise match. In addition, many deterministic calls, especially those related to system resources, require that the initial state is recorded in the trace (Section IV-B).

A. Partial Specification-Based Implementations (PSIs)

A PSI consisting of a complete reference implementation is useful for detecting cross-platform violations of stateless calls. However, for stateful and nondeterministic calls that, for example, modify the filesystem, send and receive network messages, or rely on randomness, simply running a second implementation in parallel will not ensure that the PSI returns the expected result for an API call. Instead, the PSI needs to have careful control over the state of the environment in which it executes. This is particularly important for managing nondeterminism and concurrency, where small deviations in the order of execution can dramatically affect the result. Of course, off-the-shelf library implementations that could theoretically be used as a PSI do not handle such details. In contrast, implementing a custom PSI (potentially starting from the source of an off-the-shelf implementation) yields full visibility into the library's execution environment and state, and provides exactly the required control. PSIs may also employ optimizations to perform faster than off-the-shelf implementations.

A PSI consists of an executable description of the correct, expected behavior of the API or a subset of the API. Since a PSI is executable, it is possible to replay a trace from an application directly in the PSI. The PSI can be written in any language. The behavior of the PSI dictates how all implementations of the API must behave on all platforms.

For example, our POSIX PSI models file system calls, including the `open()` system call. When the PSI executes the `open()` call, it does not open an actual file, only one in memory. To continue the `open()` example, when receiving an open action for a file that doesn't exist, the PSI checks the input flags to determine if `O_CREAT` is set (indicating to create the file if it does not exist). If `O_CREAT` is not set, the PSI returns the `ENOENT` error code, indicating that the named file does not exist in the file system. Otherwise, the PSI simulates the creation of the file, adding its descriptor to the PSI's data structure. Section V discusses more examples of building PSIs.[1]

CheckAPI enables developers to benefit from violation detection without requiring building a complete PSI that handles all API calls. PSIs are partial, meaning that they can implement a subset of the API, and can immediately be used to detect violations over the implemented calls. This greatly lowers the bar to entry and allows developers to benefit from CheckAPI without committing much effort to building a PSI. Once the benefits become clear, the developers may put in more effort into extending the PSI to more calls, checking for more potential violations. As Sections V and VI will show, the PSIs we have built are partial and were used to discover significant errors that resulted in cross-platform violations and vulnerabilities in mature APIs.

B. Initial State

Some API calls rely on state that is external to the application, such as file system state: a call may depend on the (non)existence of certain files and their contents. This state presents a challenge because we need to know the initial state of such external resources (i.e., the state before the application began executing). Each of the APIs that we evaluated with CheckAPI required a different strategy.

The JavaScript API is stateless, so this challenge did not apply. In the Repy API, filesystem state is the only external resource we need to consider. Because Repy runs in a sandboxed environment that is constrained to access files in a single directory on the system, the Repy PSI adds all of the files in this directory to the initial state. We applied CheckAPI to POSIX offline, by collecting traces during execution and then verifying their conformance with the PSI later. For this reason, we used a more sophisticated technique to infer the initial state based on the input trace. We read the trace in reverse and updated resource state to have each call in the trace execute as expected. For example, if the trace contained a call that successfully deleted a file, then we update the PSI state to make sure that the file existed prior to that call. Similarly, a call to read data from a file causes the PSI state to update that file with the expected data, and so on.

C. Nondeterministic APIs

Upto this point, a PSI will return the single, correct result for stateless and deterministic-stateful API calls. However, nondeterminism is common in APIs. Nondeterministic behavior comes from state changes outside the APIs control and view, such as many networking-based calls, calls that acquire the system time, or calls using randomness.

Since the possible behavior of a nondeterministic call does have some deterministic constraints (e.g., behavior of

[1]Of course, it is possible for a PSI to contain a bug. Indeed, a PSI may disagree with all actual libraries that implement an API. In such a case, each of these libraries would be flagged as violating the API, and the problem would have to be revealed only through manual analysis — though if multiple libraries exist, then checking for majority outputs can be helpful in narrowing the cause of the divergence.

closed sockets), a PSI can process these scenarios just like a deterministic method. However, when the PSI needs to decide what a nondeterministic value should be, given the constraints it has computed, it communicates with a *nondeterminism surrogate* to derive an allowed value. The job of the surrogate is to act as the resources and sources of nondeterminism on which the calls rely, including the network, the file system, a pseudo-random number generator, data and time resources, etc. To find the appropriate value to return to the PSI, our experience has shown that record-and-replay mechanisms are sufficient: it is adequate to seed the surrogate with the result of the execution trace action being validated and repeat that result during checking. The PSI developer can specify a pattern — such as a regular expression — of the expected values and what post-processing needs to be performed before returning the result. As an example, the `getmessage()` call in the Repy API is used to receive datagram messages; when successful, it returns the total number of bytes received. This value should be between 0 and 65507, so the regular expression for this can be expressed as "`^.{0,65507}$`".

If the surrogate returns a value allowed by the pattern, and results in the PSI returning the same result as is in the trace, the action is consistent with expectation and is not a violation. If the PSI returns a different value or the returned value does not match the pattern, the action is a violation.

The gain control over the execution environment, the PSI provides the necessary facilities to handle concurrency. This is the subject of our current and immediate future work, details of which we describe in Section VII.

The surrogate can also be used to reduce the state needed to verify stateful-deterministic calls. For example, when verifying a `read()` call on a file, it is essential to know the file contents. Storing the entire file in memory can be prohibitively expensive. Instead, a hash of the file or file blocks can be kept (similar to prior work [107]). The PSI can use the hashes to check the blocks or complete files returned by the surrogate. This allows the detection of errors in stored or retrieved data, without requiring the file's contents to be in memory.

V. PSI Implementations

With the goal of discovering cross-platform violations in real APIs, we developed PSIs for three programming interfaces: JavaScript, Repy, and POSIX. These used different techniques, which we discuss below.

A. JavaScript

To check for the compliance of JavaScript implementations, the ECMA standards body for JavaScript [28] has created a conformance suite called test262 [29]. In selecting an implementation from which to build a PSI, we ran test262 on a set of popular browsers (Chrome, Firefox, and IE9) and found that Internet Explorer 9, at the time, had the most accurate implementation. Due to IE9's closed environment we were not able to isolate its JavaScript implementation to create a PSI from, so we instead used the second-most compliant JavaScript implementation, JaegerMonkey (from Firefox). We were able to use JaegerMonkey as a PSI for JavaScript because we limited our JavaScript evaluation scope to only stateless calls.

We wrote a custom script and browser extension for Chrome to interpose on stateless JavaScript calls and create execution traces (recall Section III-A). The browser extension then sends the traces to a web server, which runs the CheckAPI detection mechanism. Using the trace, CheckAPI replays actions in the PSI. Since each trace's action includes the method and its arguments, the detection mechanism simply re-executes the method call with its arguments and compares the results with those listed in the trace. CheckAPI declares a conformance failure if the result from the PSI does not match the result from the trace.

B. Repy PSI

We derived the PSI for the Repy API by modifying the API implementation. The Repy library is implemented in Python and is used to run experiments on the Seattle peer-to-peer testbed [80]. Non-platform-specific code from the implementation can be re-used in the PSI. All platform-specific code (e.g., file and network IO calls) must be re-written to interface with new in-memory data-structures inside the PSI. For example, platform-specific file IO calls are replaced with calls to in-memory data-structures that keep track of file system metadata. Calls that involve nondeterminism (e.g., network receive calls) must be changed to request a value from the surrogate (discussed in Section IV-C).

Figure 3 shows the library and PSI implementations of the Repy `removefile()` call. The library uses Python's `os` library to directly interact with Python's underlying platform-specific code (seen on lines 10, 11, 14, and 18). In contrast, the PSI uses in-memory data-structures (seen on lines 6, 14, 18) to represent these same interactions. Each code block seen in the library and PSI for `removefile()` can be directly mapped to one another in the figure to see how the library is translated into its corresponding PSI. E.g.: line 14 of the library calls Python's `os.path.isfile()` to see if the file exists. If the file does exist it is later removed on line 18 with the `os.remove()` call. The PSI, however, references a dictionary representation of the file system.

The full Repy PSI was written over the course of a few days by an undergraduate student who had no prior knowledge of the internal implementation of Repy. The time it took to implement this PSI was largely reduced because of our ability to re-use non-platform-specific code from the original Repy libraries. The Repy libraries included a collection of unit tests that were used to test basic functional correctness and API conformance. Since the PSI is an executable implementation we are able to re-run these same unit tests. The original Repy implementation consisted of 4,435 lines of code (measured by removing white-space and comments), where as the Repy PSI consisted of 869 lines, 762 of which differ from the original implementation. We suspect there to be further similarities between the PSI and the implementation but were unable to easily quantify them.

```
1  def removefile(filename):
2    # raise a RepyArgumentError if the filename isn't valid
3    _assert_is_allowed_filename(filename)
4
5    # Check if the file is in use
6    if filename in OPEN_FILES:
7      raise FileInUseError("Cannot remove file. It's in use")
8
9    # Get the absolute file name
10   relative_path = os.path.join(REPY_CURRENT_DIR, filename)
11   absolute_filename = os.path.abspath(relative_path)
12
13   # Check if the file exists
14   if not os.path.isfile(absolute_filename):
15     raise FileNotFoundError("Cannot remove non-existent file")
16
17   # Remove the file
18   os.remove(absolute_filename)
```

(a)

```
1  def psi_removefile(filename):
2    # raise a RepyArgumentError if the filename isn't valid
3    _assert_is_allowed_filename(filename)
4
5    # Check if the file is in use
6    if filename in FILEIO_OPEN_FILES_LIST:
7      raise FileInUseError("Cannot remove file. It's in use")
8
9    # The Repy API provides the illusion of a flat file system,
10   # which requires getting an absolute path to the filename.
11   # The PSI version does not need to do this.
12
13   # Check if the file exists
14   if filename not in FILEIO_FILE_STATE_DICT:
15     raise FileNotFoundError("Cannot remove non-existent file")
16
17   # Remove the file
18   FILEIO_FILE_STATE_DICT.pop(filename)
```

(b)

Fig. 3. (a) The original implementation of the `removefile` API call in Repy. (b) The `removefile` call in the Repy PSI derived from the implementation in (a) with minor modifications (unchanged lines are colored grey).

C. POSIX PSI

The POSIX PSI was implemented in Python and covers network and file system API calls. Note that although the POSIX specification allows some platform-specific behavior [43], we treat these as cross-platform violations because we believe these are issues that should be brought to the attention of the library's user. In most cases, the documentation for the API specified the expected result. In situations in which it was ambiguous or unclear, we used tests to observe the actual behavior of POSIX in OS X.

Implementing the 34 calls for the POSIX PSI was straightforward and took only 4 days. Our implementation focused on calls that are widely used within programs, such as `open()`, and did not include rarely-used calls, such as `mount()`. It would be straightforward to support the majority of the POSIX API; however, modeling calls that perform IPC or signaling calls (e.g., `kill()`) would pose a challenge because the call does not contain information on whether it reaches the other process, but that fact affects the expected behavior.

For some aspects of the nondeterministic calls, such as those involving the network, the POSIX PSI queries the nondeterminism surrogate for guidance. Figure 4 illustrates this using the `send` call. When `send` is called, the PSI first validates the known state of the socket (lines 3–13). If it is in a state in which a `send` call may succeed (the conditions on lines 4–5, 10–11 are false), the PSI checks with the surrogate (line 16) to see whether the surrogate has a valid string or an error that should be possible given the known socket status (lines 32–39). If it receives an incorrect string (e.g., longer than the requested length) or an error that should be impossible, the PSI raises a conformance failure (line 41). Otherwise, the PSI returns the number of bytes successfully sent (line 44).

VI. Evaluation

We evaluated CheckAPI on both its effectiveness at finding cross-platform violations and its runtime performance.

A. Detected Cross-Platform Violations

JavaScript. We constructed a Chrome extension to detect JavaScript API violations at runtime. Our implementation used Firefox's JaegerMonkey implementation to validate the Chrome's V8 engine. Once we uncovered a violation, we modified the HTML to include our interposition code and opened the page in other browsers to further understand the issue. As a test, we also visited the ECMAScript compliance page [28], a page designed to test for violations. While some of these tests are for the computational aspects of JavaScript, CheckAPI is able to test the behavior of a variety of JavaScript standard objects, including `String`, `Array`, `RegExp`, and `Number`. We also used CheckAPI in our day-to-day web browsing.

CheckAPI detected two violations that are not listed in the more than 10K ECMAScript tests. The first bug deals with the way that splitting strings works, resulting in divergent behavior across the JavaScript implementations for IE, Chrome, Firefox, and Safari. In fact, there are even separate deviations for different versions of those implementations [52]. The other issue deals with sorting using the locale information from the host. This varies not only with the locale settings, but depends on the specific JavaScript settings [62]. We found outside sources that are able to confirm that these are true JavaScript API violations [52], [62]. Since CheckAPI uncovers violations that are not in the ECMAScript test suite, this demonstrates the value of testing using a PSI.

Repy. Repy's API consists of 33 calls and is built expressly with portability in mind [37]. Though this API has been in use for over four years across tens of thousands of machines, CheckAPI uncovered four file system [33], [64] and network [49], [76] violations. All of these violations were confirmed and fixed by the Repy developers.

These bugs would have been quite difficult to find or fix using other methods. For example, CheckAPI discovered a

```
1. def send_syscall(fd, message, msg_len, flags):
2.
3.   # Send can success if it is on a connected TCP socket.
4.   if ('IPPROTO_TCP' == fdtable[fd]['protocol'] and
5.       fdtable[fd]['state'] != CONNECTED):
6.     raise SyscallError("send_syscall","ENOTCONN",
7.                        "The descriptor is not connected.")
8.
9.   # Send should be used only with TCP/UDP scokets.
10.  if ('IPPROTO_TCP' != fdtable[fd]['protocol'] and
11.      'IPPROTO_UDP' != fdtable[fd]['protocol']):
12.    raise SyscallError("send_syscall","EOPNOTSUPP",
13.                       "Send not supported on this protocol.")
14.
15.  # Get the return value from the surrogate.
16.  impl_ret, impl_errno = posix_surrogate.pop()
17.
18.  # Check with implementation's behavior:
19.  # was it non-deterministic?
20.  if (impl_errno != None and
21.      impl_errno in send_nondeter_errors):
22.    raise SyscallError("send_syscall", impl_errno,
23.                       "Non-deterministic error")
24.
25.  # Omitted: in case of a UDP socket..
26.
27.  # Get the socket so I can send...
28.  sock = sockettable[fdtable[fd]['sockid']]
29.
30.  # Consult surrogate to see if we should raise
31.  # a nondeterministic error.
32.  remote_close = SocketClosedRemote("Closed remotely!")
33.  block_error = SocketWouldBlockError("Send would block.")
34.
35.  # Register regular expression condition
36.  # for verification of this function.
37.  regex_cond = regex_cond_range(0, msg_len)
38.  num_bytes_sent, error = surrogate_getter(regex_cond,
39.                                [remote_close, block_error])
40.  if error != None:
41.    return error
42.
43.  # Return the characters sent.
44.  return num_bytes_sent
```

Fig. 4. Listing of the send call implementation the POSIX PSI. The regular expression `regex_cond_range` (line 37) matches numerical ranges, while `send_nondeter_errors` (line 21) is a list of all the nondeterministic errors for this call. The listing omits initial checks for deterministic errors and the case of a UDP socket.

Portability violation	Affected Applications
Found using CheckAPI	
Inheritance of flags from accept()	Python [49]
SO_REUSEADDR semantics differ when binding to the same port	Python [87], Java [86], NTP [102], Mozilla [79]
File name character set	pervasive
Deleting an open file	Seattle Repy Sandbox
Max file name length	pervasive
File name case sensitivity	MathType, SVN [90], etc.
UDP broadcast setup	Apache [26], Netcat [68], etc.
recvfrom error codes handle connection refused differently	Twisted [96]
get/setsockopt buffer size mismatch	Seattle Repy Sandbox
Semantics of shutdown()	Ruby [31], Cygwin [2], Python [85], FreeBSD [82]
Can be found with a more expressive PSI	
Getaddrinfo behavior	Wine [105], Repy [38]
Setsockopt with closed connections	Java/Tomcat [24]
SO_(RECV/SND)TIMEO result	PHP [103], LibSoup [35]
Cannot be found with CheckAPI	
Non-blocking sockets partial send	Twisted [95]
Ioctl size of next available datagram	pervasive
OS buffer is smaller than datagram	pervasive
Datagrams truncated (or dropped) when return buffer is smaller than datagram	pervasive
Receiving datagrams over 8KB	VirtualBox VM with Windows host
recv() hangs despite close()	Ruby [93], Android [23]

Fig. 5. POSIX portability violations. The top category are found by CheckAPI. The middle could be found with a more expressive PSI. The bottom cannot be found with current techniques.

bug in Repy's network API where Python raised an error that Repy did not expect. This bug took more than two weeks of developer effort to try to find the root cause.[2] The core issue was that whether or not the `O_NONBLOCK` flag is inherited by a socket returned from `accept()` differs across operating systems. The underlying fault was actually in the Python standard libraries [49]. This was the root cause for a host of other problems [46], [47], [48].

POSIX. Since the POSIX specification allows a higher degree of variance than the other APIs we studied, there are more potential violations. To evaluate the POSIX PSI we first looked for POSIX bug reports related to portability violations with the goal of finding around 20 bugs. We found 19 bug reports across many popular applications (Figure 5). All of these violations were reproduced with CheckAPI: we first ran each application to produce traces that exhibit the violation and then ran CheckAPI to find the violation.

In 10 cases (top of Figure 5), CheckAPI correctly detected the violation. CheckAPI detected file system and network issues that impact popular applications like Firefox, Apache, Java, Ruby, Perl, Twisted, Cygwin, and Python. For example, the `SO_REUSEADDR` semantic difference allowed other programs to bind to the same port on Windows systems: Windows requires an additional flag `SO_EXCLUSIVEADDRUSE` to prohibit another process from binding to the same port. This caused security issues for a variety of applications [86], [102].

However, there were also 9 violations that CheckAPI did not detect. One reason is that our PSI covers only 34 POSIX calls. Thus for 3 violations (middle of Figure 5) the issue would be straightforward to detect given a more complete PSI. For example, it is trivial to detect the different semantics of `getaddrinfo()`, but our PSI did not implement that API call. Since the PSI is a general construct and not tailored to any one bug or test case, these issues are to be expected for partial implementations of large APIs. At the same time, this also shows that an extremely partial PSI can still find many errors.

The other 6 violations that CheckAPI does not detect are related to network behavior (bottom of Figure 5). Since the

[2]Unfortunately, we had not constructed the POSIX PSI at this time. The POSIX PSI finds the bug immediately.

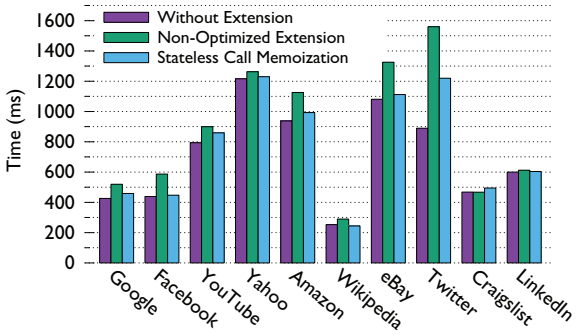


Fig. 6. Page load times comparison with and without CheckAPI. Using stateless call memoization dramatically improves the load time.

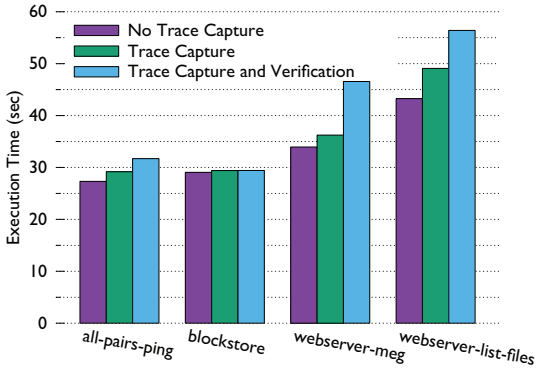


Fig. 7. The benchmark application running time, as perceived by the client.

recipient of a network packet does not know what was sent, it is not possible to detect many issues of this type. For example, a recipient cannot tell if a packet was truncated by the sender or by the local OS: this can only be detected using traces from multiple nodes, which we currently do not support.

B. Performance

JavaScript. The primary user-perceived impact is based on the injection of a custom JavaScript script for trace capture. To measure CheckAPI's page load time impact we tested it with the top 10 most popular Alexa sites [3] and measured it using Google Chrome's built-in developer tools [21]. As Figure 6 shows, using CheckAPI for runtime overhead presents a less than 9% overhead for 9 of the 10 most popular websites.

Stateless Call Memoization: Figure 6 also shows the impact of the stateless call memoization (described in Section III-B). On average, it reduces load time increase from 21.8% to 7.9% and reduces the average trace length from 2027 JavaScript calls per request to 1025. Note here that while the page may have been fully loaded, trace capture mechanism may still be recording data.

Twitter had 8,121 JavaScript calls per request, an increase of 37%. Google had 53 JavaScript calls per request, an increase of 7.6%. In practice, we have used the CheckAPI Google Chrome extension in daily web browsing with minimal perceivable impact even using our unoptimized trace gathering code.

Repy. To understand the performance of Repy with CheckAPI during online verification we evaluated it with four representative applications used on the Seattle testbed. We monitored the client-perceived running time to perform common tasks across four applications: a UDP-based P2P ping application (all-pairs-ping), a block store similar to S3 (blockstore), and a webserver with either small 1K (webserver-list-files) or large 1MB (webserver-meg) requests. We executed these applications under three different scenarios: without trace capture or CheckAPI, with trace capture, and trace capture with CheckAPI. The results from these experiments are seen in Figure 7 and show that on average trace capture (second scenario) adds an average of 10% to the runtime of a Repy program. The addition of CheckAPI-based verification (third scenario) adds 10% to the total execution time of the original application. In general, the verification time and trace gathering overhead are small compared to the actual execution times of the benchmarks.

The version of CheckAPI that was evaluated in these experiments was a prototype implementation that does multi-threaded verification (CheckAPI-MT). This implementation is further discussed in Section VII. Since CheckAPI-MT provides a superset of the features so far described in this paper we are confident that the performance of a single-threaded version of CheckAPI would perform no worse than CheckAPI-MT and would in fact exhibit less verification overhead.

POSIX. To understand the overhead imposed by CheckAPI with the POSIX PSI, we first wanted to understand the impact trace capture had on applications. We therefore benchmarked the Apache webserver using the benchmarking tool ApacheBench [1] on a 320KB web page (320KB is the the average page size on the web [100]). We used ApacheBench to issue 100 requests with a maximum of 10 concurrent requests. Apache served about $\frac{1}{3}$ as many requests with `strace` as without. We believe this result is largely due to the high number of context switches by `strace`. Increasing the content served by an order of magnitude resulted in a slowdown of only 38%. The time taken by offline verification was less than 10% of the actual execution time.

VII. Future Work: Multi-Threaded Applications

Up to now we have described the CheckAPI approach as applied to a single-threaded application. For broader applicability and to find more violations we have begun work to extend CheckAPI to handle multi-threaded applications. We now discuss our preliminary work on this CheckAPI-MT version.

To support multi-threading, we require that traces record a start action (with no return value) prior to the call's execution and a finish action when the method returns. Tools such as `strace` [89] generate traces for multi-threaded programs that conform to this requirement.

When multiple threads concurrently use an API, the order in which the actions are recorded may differ from the order in which they actually occurred. Guarding API calls with locks can potentially prevent this, but in many cases, decreasing application performance and potentially causing correctness issues or producing inaccurate traces (e.g., due to blocking

Thread1: removefile_start('oldfile')
Thread2: listfiles_start()
Thread1: removefile_finish('oldfile') → SUCCESS
Thread2: listfiles_finish() → ['oldfile']

Fig. 8. An example ambiguous trace used by CheckAPI-MT with only one valid ordering of the two actions: [Thread2: listfiles, Thread1: removefile].

calls). Instead, CheckAPI-MT must handle what we call *ambiguous traces*.

Figure 8 shows an example ambiguous trace. In this trace, two threads execute one Repy call each: Thread1 executes `removefile()`, which deletes the file `oldfile`, and Thread2 executes `listfiles()`, which indicates that `oldfile` is present. As explained above, each of the two calls in the figure has a start and a finish action. If `removefile()` finished before `listfiles()`, then the return value of `listfiles()` would be incorrect; this implies that `removefile()` must have executed after `listfiles()`. The challenge for CheckAPI-MT is therefore to filter out these kinds of implausible traces and focus on the remaining ones.

CheckAPI-MT uses *trace disambiguation* to explore a subset of the space of all totally ordered sequences corresponding to an ambiguous trace to find a valid call order. If no such order is found, the trace is flagged as a violation. To successfully replay many different orderings of a trace, CheckAPI-MT must store versions of the application's global state, this way it can efficiently revert the side-effects of a set of actions.

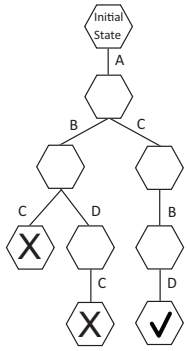
CheckAPI-MT first replays the trace in the PSI: to process the trace in Figure 8, we first replay `removefile()` then `listfiles()`. This ordering causes a conformance failure: `oldfile` was successfully removed, which means `listfile()` should not return it, which causes CheckAPI-MT to backtrack, reverting all side-effects of this action, and to replay these calls in a different order. This process continues until CheckAPI-MT either finds an ordering that does not cause a conformance failure or explores all orderings.

Figure 9 shows a more involved example with four actions, referred to as A, B, C, and D. Figure 9(a) shows the trace and Figure 9(b) shows a corresponding tree of call interleavings that CheckAPI-MT explores (each path in the tree is one interleaving). In this example, CheckAPI-MT explores this tree from left to right: it first traverses the left most path, [`sleep()`, `openfile()`, `listfiles()`] by replaying each action in the PSI as it goes. In this trace, `openfile()` created a file called `newfile` that does not appear in the listing produced by `listfiles()`, so CheckAPI-MT raises a conformance failure. Trace disambiguation then restores the previous PSI state and backtracks up the tree to try another branch. The trace disambiguation process will next check the interleaving [`sleep()`, `openfile()`, `file.close()`, `listfiles()`], which also raises a conformance failure because the `openfile()` action was replayed before the `listfiles()` action. CheckAPI-MT will then proceed to explore the final (right-most) trace in the tree, which is a valid interleaving of the calls in Figure 9(a).

The task of finding a valid trace ordering is similar to finding

A: sleep_start(1.0)
A: sleep_finish(1.0) → SUCCESS
B: openfile_start('newfile', True)
C: listfiles_start()
B: openfile_finish('newfile', True) → fileobj1
D: file.close_start(fileobj1)
D: file.close_finish(fileobj1) → SUCCESS
C: listfiles_finish() → ['oldfile']

(a)



(b)

Fig. 9. An ambiguous trace (a) and its corresponding tree (b) during trace disambiguation. Each action's color represents if Thread 1 or Thread 2 performs it.

valid linearization points [42] for each API call. We essentially have to find a correct ordering where each API call took effect and determine if a call "happens before" another call [58]. This task is made easier for applications that consist of many stateless API calls since these calls do not have any side-effects in the PSI and are thus are not restricted in their call order. We believe that future work in making CheckAPI-MT practical can leverage work in distributed systems and parallel computing.

CheckAPI-MT includes a few heuristics to cull the space of orders it needs to explore. For example, we do not need to consider re-ordering stateless calls or calls that occur in the same thread. In addition, if an action A is reported as finished before another action B starts, then we do not consider the ordering $[B, A]$. In practice, we did not find any cases where these heuristics caused us to miss an ordering, and it greatly reduces the space of interleavings. For example, a naïve approach would consider all 24 possible orders of actions in Figure 9. However, because the `sleep()` call starts and finishes before all other calls start, CheckAPI-MT only considers orderings in which `sleep()` appears in the first position. This reduces the search space by a factor of 4, from 24 orderings to 6.

We have built a prototype checker using these principles and have already found it to be useful (as discussed in Section VI-B). However, there are probably many more opportunities for optimization, which will be necessary for processing long call sequences. Therefore, we consider CheckAPI-MT a work in progress.

VIII. Related Work

Static analysis: The Explode system uses model checking to check properties of file systems [106]. MC enables writing checkers that are compiled into static checks for violations of system rules and security properties [5], [30]. SLAM [6], [7] uses model checking to check if an implementation conforms to rules characterizing correct API usage; it has been very successful in finding bugs in Windows device drivers. By contrast, verification techniques like CheckAPI typically scale

better and are less prone to false-positives, but they provide less strong guarantees of correctness.

Static analysis can integrate static API call verification into an API implementation [88]. This requires formally specifying each API call, whereas a PSI specifies API behavior with a direct implementation. Silakov et al. [83] statically detect portability problems in Linux applications, while the Java API Compliance Checker [51] checks binary- and source-level compatibility of an application against a Java library API. These tools check static properties, while CheckAPI validates an API's runtime behavior.

Portability: System call interposition can monitor executions of binaries to automatically assemble packages of code needed to run applications on other platforms [41]. Inferred behavioral models of software component interactions can automatically generate compatibility test suites [65]. Compliance-validated wrappers can be used for principled migration of APIs across platforms [13]. In C, feature test macros can explicitly specify compatibility with a specific OS. These approaches are complementary to CheckAPI, whose goals is to detect portability issues so that they can be fixed.

Skoll addresses portability bugs arising from configuration problems by sampling the configuration space using distributed and diverse machines [66]. Skoll, together with combinatorial techniques, can systematically discover combinations of configuration options (selected from a large but well-defined space) that cause a given test suite to fail [34], [108].

Recently, several tools have addressed the issue of cross-browser incompatibilities for web applications [19], [20], [67]. These involve computer vision algorithms to detect differences in page rendering, along with dynamic web crawling. Choudhary et al. [83] additionally compare web applications' possible states across browsers. Cross-browser incompatibilities detected by these techniques can arise from differences in JavaScript interpreters, along with other root causes. Integrating CheckAPI for JavaScript into such tools could improve their accuracy and help in fault diagnosis.

POSIX conformance testing: The Ballista system generates parameter values for POSIX functions aimed at detecting implementation problems. In fifteen popular OS implementations, Ballista revealed many bugs through crashes and abnormal task termination [57]. By contrast, CheckAPI detects violations that are the result of more subtle differences in OS behavior.

Model-based test and oracle generation [50], [78] can aid conformance, and has been applied to POSIX file systems [25] and to other POSIX aspects [32]. This work generates tests to validate library behavior and is complementary to CheckAPI, which finds violations at runtime. Executing test suites across multiple configurations of a product or system, known as variability execution [54], [71], may reveal differences between the configurations. By contrast, CheckAPI focuses on executing multiple implementations that share no or little code.

Reliability via redundancy: Developing and running multiple implementations of an application in parallel, comparing them at checkpoints, can achieve high reliability under the assumption that the implementations fail independently [56]. Such reliability can be made optimal, in terms of the number of necessary implementations [14]. CheckAPI's use of a PSI differs because a PSI represents an idealized execution of the system, avoiding OS behavior to the greatest extent possible. For example, the file system metadata in a CheckAPI PSI is likely to be entirely in memory. Thus, while n-version testing would have all implementations heavily dependent on the same underlying OS behavior, a PSI is independent of the OS implementation and can discover violations caused by the OS.

Runtime verification: Runtime verification (RV) techniques [4], [11], [59], [61], [63], [77], [94], [98] can detect violations of properties on specific executions but do not show that the software satisfies the specification for every possible input or on every possible execution. Many of these techniques find general violations of properties, such as atomicity [77], [98]. Other RV techniques enable checking program-specific requirements usually specified with formal languages, such as automata or logic formalisms [12], [39].

Many RV approaches instrument code to capture relevant events or application state and insert executable assertions [9], [10], [22], [53], [60], [75]. However, inserting pre- and post-conditions obscures the fact that the specification can be treated as a parallel construct to the implementation [9], [10]. Instead, an architecture can be used for runtime verification of .NET components by running the model and the implementation side-by-side, comparing results at method boundaries [9], [10]. CheckAPI does not require application instrumentation, assuming a tracing mechanism exists in the API [16], [89].

Like CheckAPI, several other (runtime and static) checking techniques allow the use of languages more familiar to programmers. The WiDS checker allows using a scripting language to specify properties of a distributed system [61]. Contracts written in a C-like language can specify components for use in TinyOS applications [4]. CheckAPI allows programmers to choose the language for PSI construction or simply to use an existing implementation. Lastly, work on deterministic replay [99] enables record-replay techniques on multi-core systems and could help improve CheckAPI-MT performance.

IX. Contributions

Latent cross-platform violations are difficult to identify and costly. We have presented CheckAPI, a technique for effectively identifying cross-platform violations at runtime. CheckAPI is efficient and finds bugs in popular and mature projects, including JavaScript, Repy, and POSIX, suggesting promise for improving cross-platform library quality and facilitating software reuse.

X. Acknowledgments

Yanyan Zhuang, Savvas Savvides, Yang Li, Jonathan Jacky, Phyllis Frankl, and Ivan Beschastnikh provided useful feedback and implementation help. This work is supported by the National Science Foundation under grants CNS-1205415, CNS-1405904, CCF-1453474, CNS-1513055, CNS-1513457, and DGE-1058262.

REFERENCES

[1] ab — Apache HTTP server benchmarking tool. Accessed May 1st, 2012 http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] Re: listen socket / poll block. Accessed April 30th, 2012 http://cygwin.com/ml/cygwin/2011-04/msg00235.html.

[3] Alexa top 500 global sites. Accessed April 30th, 2012 http://www.alexa.com/topsites.

[4] W. Archer, P. Levis, and J. Regehr. Interface contracts for tinyos. In *IPSN*, pages 158–165, 2007.

[5] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

[6] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM, July 2011.

[7] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[9] M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In *Workshop on Specification and Verification of Component-Based Systems*, 2001.

[10] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Journal of Systems and Software*, 65(3):199–208, 2003.

[11] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors. *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418. Springer, 2010.

[12] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.

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

[14] Y. Brun, G. Edwards, J. young Bang, and N. Medvidovic. Smart redundancy for distributed computation. In *ICDCS*, pages 665–676, June 2011.

[15] Cannot bind to default port after using connection and restarting. Accessed May 2nd, 2012 https://bugs.eclipse.org/bugs/show_bug.cgi?id=249736.

[16] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *CCS*. ACM, 2010.

[17] P. Chanezon. Write Once, Run Anywhere: the devil is in the details, October 2006. http://wordpress.chanezon.com/?p=7.

[18] B. Charny. Write once, run anywhere not working for phones, July 2005. http://mcall.com.com/Write-once,-run-anywhere-not-working-for-phones/2100-1037_3-5788766.html.

[19] S. R. Choudhary. Detecting cross-browser issues in web applications. In *ICSE*, pages 1146–1148, 2011.

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

[21] Chrome developer tools: Network panel. Accessed April 1st, 2012 http://code.google.com/chrome/devtools/docs/network.html.

[22] J. A. Clause and A. Orso. Camouflage: automated anonymization of field data. In *ICSE*, pages 21–30, 2011.

[23] Closing a socket from another thread doesn't generate IOException. Accessed April 30th, 2012 http://code.google.com/p/android/issues/detail?id=7933.

[24] Confusing error "java.net.SocketException: Invalid argument" for socket disconnection. Accessed April 30th, 2012 http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6378870.

[25] 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*, pages 153–166. Springer-Verlag, 2008.

[26] Datagram (UDP) broadcasts fail with "permission denied" on platforms that require SO_BROADCAST to be set first. Accessed April 30th, 2012 https://issues.apache.org/bugzilla/show_bug.cgi?format=multiple&id=46389.

[27] Developers battle with over 100 different versions of android. Accessed April 26th, 2012 http://www.pcpro.co.uk/news/enterprise/361948/developers-battle-with-over-100-different-versions-of-android.

[28] Ecmascript language specification. Accessed March 28th, 2012 http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[29] Ecmascript test262. Accessed March 28th, 2012 http://test262.ecmascript.org/.

[30] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, page 1, 2000.

[31] 'errno::ENOTCONN: Socket is not connected' in test/net/imap. Accessed April 30th, 2012 http://bugs.ruby-lang.org/issues/465.

[32] E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1):89–110, 2002.

[33] file names not case-sensitive? Accessed April 30th, 2012 http://forums.macrumors.com/showthread.php?t=1037142.

[34] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *ISSTA*, pages 177–188, 2009.

[35] Found an interesting bug in libsoup on Windows. Accessed April 30th, 2012 http://mail.gnome.org/archives/libsoup-list/2008-June/msg00001.html.

[36] J. Fruhlinger. LWUIT: Write once, run anywhere (mobile) (hopefully), August 2008. http://www.javaworld.com/community/node/1113.

[37] Future repy library reference. Accessed April 24th, 2012 https://seattle.cs.washington.edu/wiki/FutureRepyAPI.

[38] getmyip fallback to using TCP for Windows Mobile. Accessed May 1st, 2012 https://seattle.cs.washington.edu/changeset/1337/seattle/trunk/repy/emulcomm.py.

[39] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *ASE*, pages 412–416, 2001.

[40] J. Gosling. Java: Write-Once, Debug Everywhere?, May 2008. http://www.uberpulse.com/us/2008/05/java_write_once_debug_everywhere.php.

[41] P. J. Guo and D. Engler. CDE: Using system call interposition to automatically create portable software packages. In *USENIX*, 2011.

[42] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[43] *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*, 2001.

[44] Internet explorer performance lab: reliably measuring browser performance. Accessed January 23rd, 2015 http://blogs.msdn.com/b/b8/archive/2012/02/16/internet-explorer-performance-lab-reliably-measuring-browser-performance.aspx.

[45] Issue with socket SO_REUSEADDR when a client is connected. Accessed May 2nd, 2012 http://comments.gmane.org/gmane.os.cygwin/1246085.

[46] Strange behavior for socket.timeout. Accessed April 29th, 2012 http://bugs.python.org/issue10473.

[47] OSX broken poll testing doesn't work. Accessed April 29th, 2012 http://bugs.python.org/issue5154.

[48] Socket timeout can cause file-like readline() method to lose data. Accessed April 29th, 2012 http://bugs.python.org/issue7322.

[49] On mac / bsd sockets returned by accept inherit the parent's fd flags. Accessed April 29th, 2012 http://bugs.python.org/issue7995.

[50] J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.

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

[52] JavaScript split bugs: Fixed! Accessed April 30th, 2012 http://blog.stevenlevithan.com/archives/cross-browser-split.

[53] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. *SIGPLAN Not.*, 45:241–255, October 2010.

[54] 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*, pages 1–8, 2012.

[55] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella. Reproducing field failures for programs with complex grammar-based input. In *ISSTA*, pages 163–172, 2014.

[56] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Software Eng.*, 12(1):96–109, 1986.

[57] P. Koopman and J. DeVale. The exception handling effectiveness of posix operating systems. *IEEE TSE*, 26(9):837–848, 2000.

[58] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[59] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[60] B. Liblit. *Cooperative Bug Isolation (Winning Thesis of the 2005 ACM Doctoral Dissertation Competition)*, volume 4440. Springer, 2007.

[61] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS checker: Combating bugs in distributed systems. In *NSDI*, pages 19–19, 2007.

[62] localecompare implementation differs. Accessed April 29th, 2012 http://code.google.com/p/v8/issues/detail?id=459.

[63] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48. ACM, 2006.

[64] Mac and Windows OS file/folder naming rules. Accessed April 30th, 2012 http://www.portfoliofaq.com/pfaq/FAQ00352.htm.

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

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

[67] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 561–570. ACM, 2011.

[68] ncat broadcast support? Accessed May 3rd, 2012 http://seclists.org/nmap-dev/2010/q2/440.

[69] net.listentcp succeeds twice on windows. Accessed May 2nd, 2012 http://code.google.com/p/go/issues/detail?id=2307.

[70] New — SO_REUSEADDR should also set SO_REUSEPORT on FreeBSD UDP sockets. Accessed May 2nd, 2012 http://lists.ximian.com/pipermail/mono-bugs/2007-June/058136.html.

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

[72] On android compatibility. Accessed April 26th, 2012 http://android-developers.blogspot.com/2010/05/on-android-compatibility.html.

[73] Openssh 5.1. Accessed May 2nd, 2012 http://www.openssh.com/txt/release-5.1.

[74] Openvpn change log — version 2.1.4. Accessed May 2nd, 2012 http://openvpn.net/index.php/open-source/documentation/change-log/71-21-change-log.html.

[75] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 65–69. ACM, 2002.

[76] Ticket 426: z_testconcurrentopenconns.py fails on Mac 10.4 (Tiger). Accessed January 23rd, 2015 https://seattle.cs.washington.edu/ticket/426.

[77] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 25–36. ACM, 2009.

[78] A. Pretschner. Model-based testing. In *ICSE*, pages 722–723, 2005.

[79] Pr_sockopt_reuseaddr functions differently on windows from on other oses. Accessed April 30th, 2012 https://bugzilla.mozilla.org/show_bug.cgi?id=489488.

[80] Seattle. https://seattle.cs.washington.edu/.

[81] Security: SO_EXCLUSIVEADDRUSE should be enabled when binding to ports on Windows. Accessed May 2nd, 2012 http://twistedmatrix.com/trac/ticket/4195.

[82] shutdown() of non-connected socket should fail with ENOTCONN. Accessed April 30th, 2012 http://comments.gmane.org/gmane.os.freebsd.bugs/31006.

[83] 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*, pages 357–370. Springer, 2010.

[84] socket - linux socket interface. Accessed May 3rd, 2012 http://www.kernel.org/doc/man-pages/online/pages/man7/socket.7.html.

[85] socket.shutdown documentation: on some platforms, closing one half closes the other half. Accessed April 30th, 2012 http://grokbase.com/t/python/docs/121cvag9n1/issue6774-socket-shutdown-documentation-on-some-platforms-closing-one-half-closes-the-other-half.

[86] SO_REUSEADDR broken on Windows. Accessed April 30th, 2012 http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4476378.

[87] SO_REUSEADDR doesn't have the same semantics on Windows as on Unix. Accessed April 30th, 2012 http://bugs.python.org/issue2550.

[88] D. Spinellis and P. Louridas. A framework for the static verification of api calls. *Journal of Systems and Software*, 80(7):1156–1168, 2007.

[89] strace(1) - Linux man page. Accessed March 28th, 2012 http://linux.die.net/man/1/strace.

[90] Subversion problem with case sensitivity. Accessed April 29th, 2012 http://stackoverflow.com/questions/713220/subversion-problem-with-case-sensitivity.

[91] Sun Java J2EE – Compatibility & Java Verification. http://java.sun.com/j2ee/verified/.

[92] TCPServer should not use SO_REUSEADDR in Cygwin port. Accessed May 2nd, 2012 http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/6765.

[93] Tcpsocket#readline doesn't raise if the socket is #close'd in another thread. Accessed April 30th, 2012 http://bugs.ruby-lang.org/issues/4390.

[94] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 131–144. ACM, 2007.

[95] [Twisted-Python] ENOBUF and Twisted. Accessed January 23rd, 2015 http://twistedmatrix.com/pipermail/twisted-python/2004-August/008461.html.

[96] UDP error trapping on Cygwin. Accessed April 30th, 2012 http://twistedmatrix.com/pipermail/twisted-python/2007-February/014770.html.

[97] Using SO_REUSEADDR and SO_EXCLUSIVEADDRUSE. Accessed May 3rd, 2012 http://msdn.microsoft.com/en-us/library/windows/desktop/ms740621(v=vs.85).aspx.

[98] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *OSDI*, pages 117–130, 2006.

[99] N. Viennot, S. Nair, and J. Nieh. Transparent mutable replay for multicore debugging and patch validation. In *ASPLOS*, pages 127–138, 2013.

[100] Web metrics: Size and number of resources. Accessed May 1st, 2012 https://developers.google.com/speed/articles/web-metrics.

[101] Why android upgrades take so long. Accessed April 26th, 2012 http://mobile.slashdot.org/story/11/12/09/1839223/why-android-upgrades-take-so-long.

[102] Windows ntpd should secure UDP 123 with SO_EXCLUSIVEADDRUSE. Accessed April 30th, 2012 https://support.ntp.org/bugs/show_bug.cgi?id=1149.

[103] Winsock programmer's FAQ articles: The lame list. Accessed April 30th, 2012 http://tangentsoft.net/wskfaq/articles/lame-list.html.

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

[105] ws2_32: getaddrinfo edge cases broken on OS X. Accessed April 30th, 2012 http://bugs.winehq.org/show_bug.cgi?id=29756.

[106] J. Yang, C. Sar, and D. Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *OSDI*, pages 131–146, Seattle, WA, USA, 2006.

[107] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, Nov. 2006.

[108] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *ISSTA*, pages 45–54, 2004.

[109] Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos. NetCheck: Network diagnoses from blackbox traces. In *NSDI*, pages 115–128, Apr. 2014.