Published in final edited form as:
Proc Int Workshop Autom Softw Test. 2010 ; 2010: 16–23. doi:10.1145/1808266.1808269.

Automatic Detection of Previously-Unseen Application States for Deployment Environment Testing and Analysis

Christian Murphy, **Moses Vaughan**, **Waseem Ilahi**, and **Gail Kaiser**
Dept. of Computer Science, Columbia University, New York NY 10027 {cmurphy, mjv2123, wki2001, kaiser}@cs.columbia.edu

Abstract

For large, complex software systems, it is typically impossible in terms of time and cost to reliably test the application in all possible execution states and configurations before releasing it into production. One proposed way of addressing this problem has been to continue testing and analysis of the application in the field, after it has been deployed. A practical limitation of many such automated approaches is the potentially high performance overhead incurred by the necessary instrumentation. However, it may be possible to reduce this overhead by selecting test cases and performing analysis only in previously-unseen application states, thus reducing the number of redundant tests and analyses that are run. Solutions for fault detection, model checking, security testing, and fault localization in deployed software may all benefit from a technique that ignores application states that have already been tested or explored.

In this paper, we present a solution that ensures that deployment environment tests are only executed in states that the application has not previously encountered. In addition to discussing our implementation, we present the results of an empirical study that demonstrates its effectiveness, and explain how the new approach can be generalized to assist other automated testing and analysis techniques intended for the deployment environment.

Keywords

Software Testing; In Vivo Testing

1. INTRODUCTION

Software products released into the field typically have some number of residual defects that either were not detected or could not have been detected during testing prior to deployment. This may be the result of flaws in the test cases themselves, incorrect assumptions made during the creation of test cases, or the infeasibility of testing the sheer number of possible configurations and application states for a complex system; these defects may also be due to application states that were not considered during lab testing, or corrupted states that could arise due to a security violation.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Reliability, Verification

Automated testing solutions such as “perpetual testing” [17] and “residual testing” [18] suggest continuing the testing of applications into the deployment environment, based on the assumption that, over time, defects will reveal themselves given that multiple instances of the same application may be run globally with different configurations, under different patterns of usage, and in different system states. Other techniques for profiling and analyzing deployed software (such as those surveyed by Elbaum and Hardojo [6]) are based on a similar observation that the best way to understand how software is used is to observe it as it runs in the field.

A limitation of any approach that conducts tests or program analysis in the deployment environment is the potentially high performance overhead. The instrumentation required for these approaches may need to update data structures, write to local files, send information over the Internet, or invoke test cases that slow down normal program execution. This overhead can be prohibitive from the users’ perspective, considering that potentially all functions in a program might be instrumented.

In this regard the following question arises: “*Can these approaches be made more efficient by only running tests and performing analysis in application states that the program has not seen before?* ” That is, it may be possible to reduce the number of redundant tests and analyses that are run by ignoring application states that have already been tested or explored, assuming that the result of the test depends only on the application state. Of course, determining whether the state has already been seen incurs its own cost, but solutions for fault detection, model checking, security testing, and fault localization in deployed software may all benefit from such a technique if it can be implemented efficiently.

In this paper, we describe such a technique and implement a solution for an automated testing approach called “In Vivo Testing” [13], which conducts tests in deployed applications. We demonstrate both theoretically and empirically that it is possible to improve the efficiency of the approach by ensuring that test cases are only selected in states that the application has not previously encountered. Although our implementation and evaluation focus particularly on In Vivo Testing, we also demonstrate that the technique is applicable to a variety of other approaches for testing and analyzing deployed software.

The rest of this paper is organized as follows. Section 2 motivates the work by providing examples from various areas of software testing and analysis that could benefit from such a technique. Section 3 then presents background information on In Vivo Testing, including a theoretical analysis of how the approach can be improved by only running tests in previously-unseen states. Section 4 describes our implementation, and Section 5 evaluates its efficiency. Sections 6 and 7 describe future work and related work, respectively, and Section 8 concludes.

2. MOTIVATION

The ability to quickly determine at runtime whether the current program state has previously been encountered has practical application for many testing and dynamic analysis approaches.

For instance, model checking techniques could benefit from knowing whether a function has already been run in a given state. A function may be executed once in a particular state, and then be revisited later via a different execution path, but be set for execution in the same state as before. If it were known that the function had already been checked in that state, then pruning could occur at that point, reducing the number of paths that later need to be investigated. This would be particularly useful for distributed model checking frameworks

[9], which require knowing which parts of the state space to distribute, based on which ones have already been considered.

If the set of states in which a function had been executed were known, then checking the set of previously-encountered states could also be used for security testing. We have previously demonstrated [5] that tests in deployed software could be used to check for "security invariants" [2], the violation of which indicate a vulnerability in the software. In that approach, the invariants are test cases written by the tester or developer. However, if the set of acceptable good states (or known bad states) were automatically pre-populated in advance, then it would be easy to determine whether a given function execution should or should not be allowed, given the current state; control could then be sent to a "rescue point" [20] if a known bad state were encountered. Even if the set were not pre-populated, the approach could be used for anomaly detection, i.e., determining whether a particular execution occurs in a state that varies greatly from previous executions.

The application state data collected at runtime could also be sent back to the developers, as it may be useful for the developers of the software to know which functions are being called with what arguments, the number of times the functions are called, the frequency with which they are called in the same state, etc. This information could then be used in regression testing and test case selection [6]. If the sequence of function calls and their corresponding states were also recorded, the data could then be used for fault localization: once an test fails in the field, the developers could investigate the history of function calls in the different application states, culminating with the failed test, and then compare it to executions that did not fail and use techniques such as delta debugging [22] to determine where the defect may have occurred.

Last, such a technique could be used for automatic memoization [14], in that the results (i.e., the output and any side effects) of functions can automatically be cached, thus speeding up the application further. That is, if it can quickly be determined that the function has already been called with the same set of arguments and/or in the same application state, then if the results of the function are already known, there is no need to perform the calculation a second time. Rather, the cached results can be returned, without having to actually execute the function.

3. BACKGROUND

In this work, we implement a solution for detecting previouslyunseen application states and apply it to a testing methodology called In Vivo Testing [13].

3.1 In Vivo Testing

The motivation behind the In Vivo Testing approach is the fact that many (if not all) software products are released into deployment environments with latent defects still residing in them, as well as the observation that these defects may reveal themselves when the application executes in states that were unanticipated and/or untested in the development environment. The approach can be used to detect defects hidden by assumptions of a clean state in the tests, errors that occur in field configurations not tested before deployment, and problems caused by unexpected user actions that put the system in an unanticipated state; these flaws may also be due to corrupted states that could arise due to a security violation. The approach goes beyond passive application monitoring (e.g., [15]) in that it actively tests the application as it runs in the field.

In Vivo Testing is an automated testing approach by which a program continuously conducts tests in the deployment environment, in the context of the running application, as opposed to a controlled or blank-slate environment. Crucial to the approach is the notion that the test must not alter the state of the application from the user's perspective. In a live system in the deployment environment, it is clearly undesirable to have a test alter the system in such a way that it affects the users of the system, causing them to see the results of the test code rather than of their own actions. In the simplest case, In Vivo tests can be thought of as program invariants or assertions that are allowed to have side effects, but the side effects are hidden from the user.

In Vivo Testing works as follows: when an instrumented function is to be executed, a corresponding test is then automatically executed in a separate "sandbox" that allows the test to run without altering the state of the original application process. The application then continues its normal operation as the test runs to completion in the sandbox, and the results of the test are logged.

In the current implementation of the In Vivo Testing framework, called "Invite", creating a sandbox is achieved by forking a new process, which creates a copy-on-write version of the memory space for the child process in which the test is run. To make the sandbox more robust, Invite has been integrated with a virtualization layer called a "pod" (PrOcess Domain) [16], which creates a virtual environment in which the process has its own view of the file system and process ID space and thus does not affect any other processes or any shared files.

Although the cost of creating a sandbox via a simple process fork to run the In Vivo tests can be less than a millisecond per test, in practice there may be thousands or tens of thousands (or more) tests per application run depending on the number of instrumented functions (conceivably all of them) and the number of times they are invoked, adding a substantial amount of time [12]. Moreover, real-world industrial applications may require the use of the "pod" virtualization layer to ensure that the test does not affect the external system state. Given that the time to checkpoint the application and create a "pod" can be over a second [12], this may incur an unacceptable performance cost from the users' point of view.

Note that there is a possibility that many of these tests will be run in the same application state multiple times, thus causing unnecessary overhead. If the test is deterministic and depends only on the current state, and not on any external factors, then it follows that it may be more efficient to avoid subsequent executions of tests in states that have already been encountered, since it would be expected that the test result would not change.

3.2 Analysis

An approach designed to increase the efficiency of In Vivo Testing (or any approach, for that matter) based on running tests only in previously-unseen states is heavily dependent on the assumption that a given function will, in fact, run in the same state multiple times. In the best case, if the function *always* runs in the same program state, then the In Vivo test will only be run once (i.e., the very first time), and the performance overhead will approach the hypothetical minimum of never running any tests, give or take a little bit of overhead from the instrumentation. In the worst case, if the function *never* runs in the same state, then In Vivo tests will run for every invocation of the function, which will incur worse performance overhead than the "standard" In Vivo approach, since not only are test functions being run, but there is extra overhead from determining whether the state had been seen before. It follows, then, that there must be some percentage of previously-seen states such that the new approach will, in fact, be more efficient.

The theoretical analysis is rather straightforward. We define the *Distinct State Percentage* (*DSP*) as the number of distinct states in which a function is called, divided by the total number of times the function is called. We define the *Repeat State Percentage* (*RSP*) as 1 - *DSP*. For example, if a function is called in states A, B, A, C, B, C, A, and A, then the *DSP* is 3/8 (since it was called 8 times and had three distinct states) and the *RSP* is 1-*DSP* = 5/8 (since five of the times, the function was called in a state it had already seen).

We also define the following:

- t_s = the time it takes to create the sandbox in which the In Vivo test will be run
- t_d = the time it takes to determine whether the function had already been run in the current state
- t_u = the time it takes to update the data structure storing previously-seen states
- N = the number of times the function is called

Given these definitions, we can simply calculate the overhead from “standard” In Vivo Testing (i.e., running tests on every function invocation) as:

$$T_{invivo} = N^* t_s.$$

We can also calculate the overhead from the suggested new approach (i.e., only running tests in previously-unseen states). The time taken when tests are run in previously-unseen states is:

$$T_{unseen} = N^* DSP^* \left(t_d + t_u + t_s\right).$$

The time taken when tests are not run because the state had already been seen is simply:

$$T_{seen} = N^* RSP^* \left(t_d\right) = N^* (1 - DSP)^* t_d.$$

The total overhead for the suggested new approach is their sum, T_{unseen} + T_{seen}.

To achieve the benefits of running tests only in states that have not previously been seen, we seek a low *DSP* such that the overhead for running tests only in previously-unseen states is less than that of running tests in every state, i.e.:

$$T_{unseen} + T_{seen} \le T_{invivo}.$$

Replacing with the formulas above and solving for *DSP*, we get:

$$\begin{aligned}
& N^* DSP^* \left(t_d + t_u + t_s\right) + N^* (1 - DSP)^* t_d \le N^* t_s \\
& \Rightarrow DSP^* \left(t_d + t_u + t_s\right) + (1 - DSP)^* t_d \le t_s \\
& \Rightarrow DSP^* \left(t_d + t_u + t_s\right) + t_d - DSP^* t_d \le t_s \\
& \Rightarrow DSP^* \left(t_u + t_s\right) + t_d \le t_s \\
& \Rightarrow DSP^* \left(t_u + t_s\right) \le t_s - t_d \\
& \Rightarrow DSP \le \left(t_s - t_d\right) / \left(t_u + t_s\right)
\end{aligned}$$

If we can construct a solution such that the time to do a lookup (t_d) or update (t_u) is much less than the time to create a sandbox (t_s), we can see that the right side of the inequality comes close to 1. This means, then, that even for a *DSP* of almost 100%, i.e., even if almost all of the states in which the function runs are distinct, then it still is better to incur the overhead of checking the state and only run tests in states that have not previously been encountered.

4. IMPLEMENTATION

In developing a new, more efficient implementation of the Invite testing framework, a number of questions immediately arise:

- How do we define a “state”?
- How do we represent the state?
- How can we quickly determine whether the state has already been seen?
- In practice, is any performance gain from running tests only in previously-unseen states outweighed by the overhead of the instrumentation required to track the states?

The following subsections discuss our new prototype, and the implementation decisions that were made in answering these questions.

4.1 Determining Function Dependencies

For our purposes, we define the “state” of the application at any given point during its execution as “the values of all variables that are in scope at that point”. We acknowledge that this definition is somewhat limiting in that it does not include the process heap or stack, or the program counter, but we expect that these would be too complex to represent in a format that can be represented and compared efficiently enough to meet our goals. We also do not include external elements such as the state of other processes, the underlying virtual machine and/or operating system, etc., for similar reasons. If any In Vivo test relies on these, then this feature of Invite can simply be disabled for the given test, so that it executes regardless of the external system state.

Note that a given function may not rely on *all* variables that are in scope at that point in the program’s execution. Thus, in determining whether a function has already been executed in the current program state, we only need to consider the variables on which that function depends, i.e., that are read during the function’s execution.

To determine which variables a function uses during its execution, we developed a simple pre-processor to parse the source code. For a given function, the pre-processor returns a list of all the global variables (i.e., those declared outside the function definition) that the function uses, and also determines which of the parameters the function depends on, since it may not actually use all of them. Alternative approaches would have been to use data dependence analysis or data flow analysis, but simply parsing the source seemed to be the easiest solution, given that we only need to identify the global variables that are read in the function, and do not need to enumerate all possible values or determine how the variables came to get their respective values at that particular point. Also, although this approach does not detect aliases (i.e., two variables that refer to the same piece of data), we are not concerned with modifications to variables, only with listing the variables that are read during the function’s execution.

Figure 1 shows a simple function that can be scanned using the pre-processor. On line 1, the parameters p1, p2, and p3 are specified; at this point, they are not yet added to the dependency list, since we do not know for certain that the function will actually use them (though it is admittedly rare that they would not be used). On line 2, the local variable k is declared, but because it is assigned a constant value, there is no dependency on this line, either. On line 3, the local variable t is declared; this statement uses the global variable a, which we assume to be declared elsewhere. Because a is on the right side of the assignment, i.e., its value is read, we can add a to the dependency list. In line 5, the conditional compares p1 to p2; thus, because those values are read, they are added to the dependency list. Also on line 5, the return value uses k and t, but we know that these are both local variables, thus there is no extra dependency. Line 6 does not use any new variables so nothing is added this line. Once we reach the end of the function on line 7, we know that the function f depends on the parameters p1 and p2 (line 1), as well as the global variable a (line 3); as it turns out, the function does *not* use the parameter p3.

Given this list of dependencies, we can then claim that, at the point when f is called, only the values of p1, p2, and a will affect its outcome; if those three variables are the same for additional executions, the output of f will not change, nor will the result of the corresponding In Vivo test.

Now consider the code in Figure 2, in which the function f1 is the same as f from the previous example, except that it calls the function g on line 5. To determine the dependencies of f1, when scanning line 5, we then need to determine the dependencies of g. We can see on line 10 that g uses the parameter p and the global variable b; those are the only dependencies of g. When that dependency list is returned to f1, the parameter p is replaced with the argument p3. Thus, the overall dependency list for function f1 becomes: parameters p1 and p2, and global variable a, for the reasons described in the previous example; global variable b, inherited from function g; and parameter p3, which was passed as an argument to g.

Note that this approach works for other data types as well, including arrays and values referred to by pointers.

In situations in which the pre-processor does not have access to the source code, e.g., if the code makes a system call or uses some external library, then it is impossible to know for certain what the dependencies are, and thus this approach cannot be used. In these cases, the In Vivo tests will be run regardless of the current application state.

4.2 Representing States

Once we know the variables on which a function depends, we then need a way of representing the state so that it can be compared to other states to determine whether it has previously been encountered. We can at this point consider the state as a map of a set of variables to their corresponding values. Comparing the sets of values can be time consuming (at least O(n), assuming we know the ordering of the elements to compare) if done element-wise; we require a fast way of comparing the sets, ideally with no false positives (thinking two sets are the same, when actually they are not) or false negatives (thinking two sets are not the same, when actually they are).

In the best case, if we assume that the elements of the sets are numerical, then we can attempt to devise a hashing function such that every set has a distinct value. This would allow us to effectively represent different program states with a single number.

A hashing function that meets this criteria is a Cantor pairing function [19], which assigns one distinct natural number to a pair of natural numbers. Note that this function has one key characteristic that is crucial in our state formalization, in that it is simple yet effective as the implementation is simply:

$$f(k1, k2) = (1/2)(k1+k2)(k1+k2+1) + k2.$$

Using this mathematical tool, we can now take a set of values in the function's dependent state and create a single distinct value, which is critical in determining whether the state had previously been seen. The method for achieving this is to recursively apply the Cantor function to the values, i.e., f(a, f(b, f(c, …))). This can be done for array elements, as well.

4.3 Tracking Execution States

Given that we have a distinct representation of each execution state, with no false positives or false negatives, we can then select an efficient data structure to determine whether the function has already been called in the given state, by comparing it to those that it has already seen. We started by investigating the use of a hash table, but a hash table is O(n) in the worst case (where n is the number of elements, i.e., the number of states already seen), and we were hoping for something that would give a better guarantee.

We also considered using a Bloom filter, which is O(1), but a Bloom filter allows for false positives, in that we may think we have already seen a state before, even though we have not. This is not desirable for In Vivo Testing, because it might mean that tests are not executed even though the state has not yet been seen. We could, however, allow for false negatives, which would have the result of running tests in previously-seen states; this is not ideal, but at least we do not miss the opportunity to run tests in states that have not previously been encountered.

Our investigation led us to a data structure called a Judy Array[1], which has been proven to demonstrate the properties that we need in a state-management tool. It is **space efficient**, in that it is a dynamically allocated structure that will not take up space when simply declared for later use. The Judy Array also has the property of consuming memory only when it is populated, yet can grow to take advantage of all available memory if desired. These are especially important features since potentially all functions in the program will need an array to represent which states have previously been seen. A Judy Array is also **speed efficient**, and is $O(\log_{256} n)$ for lookup operations [4]. Last, it is **scalable**: this data structure has the potential to use all the available memory on a machine and also claims to be able to hold from zero to billions of elements [4].

Given the selection of a Cantor function for hashing states and a Judy Array for tracking them, we now state the process by which the In Vivo tests of a given function run using the more efficient Invite framework.

1. A pre-processor is used to read the source code and determine which parts of the state (i.e., which variables) the specified function depends on.
2. Another pre-processor creates the necessary instrumentation in the source code so that In Vivo tests become logically attached to the function that they are testing. This generated code makes use of a function that indicates whether a test has already been run in the current state.
3. When the function to be tested is called, the required parts of the current application state are hashed using the Cantor function, which generates a distinct value for that state.

[1]http://judy.sourceforge.net/

4. The code then checks the function's corresponding Judy Array to determine whether the value already exists in the data structure. If so, then the state has already been encountered, and no test is run. If the value does not exist in the array, though, the state has never previously been encountered, so the value is added to the array, and the Invite framework is instructed to run the test.
5. At this point, In Vivo Testing continues as normal.

Figure 3 shows the pseudocode for the instrumentation of a function f, which depends on global variable g and parameters p1 and p2.

When the function f is called (line 21), a check is performed (line 22) to see whether an In Vivo test should be run at this point. The function that performs this check (line 8) uses the Cantor function, recursively if necessary, to generate a distinct value to represent the parts of the state on which the function depends (line 10). If the Judy Array for that function already contains the state (line 12), then there is no need to run the test again (line 13); otherwise, the state is added to the Judy Array (line 15), and the framework is instructed to run the test (line 16).

If it is determined that a test should be run, Invite then forks a new process (line 23), which is a copy of the original, to create a sandbox in which to run the test code, ensuring that any modification to the local process state caused by the In Vivo test will not affect the "real" application, since the test is being executed in a separate process with separate memory. Once the test is invoked (line 25), the application can continue its normal execution, in which it invokes the original "wrapped" function (line 31), while the test runs in the other process. Note that the application and the In Vivo test run in parallel in two processes; the test does not preempt or block normal operation of the application after the fork is performed. When the test is completed, Invite logs whether or not it passed (lines 25-26), and the process in which the test was run is terminated (lines 27-28).

5. EVALUATION

Although the theoretical analysis provided above shows that the new In Vivo approach will be more efficient even when the function runs in many different distinct sets, we know that in practice the variables used in the calculations may not actually be constant, and we do not know for certain whether the time to fork a new process is significantly higher than the time to do a lookup and update in the Judy Array implementation.

To demonstrate that the new approach is, in fact, more efficient, we conducted a simple experiment in which we measured the time it took to run an application with no In Vivo tests at all (the theoretical minimum time), the time to run with the "standard" Invite framework that always executes tests regardless of the state, and the time to run with the new Invite framework, using varying percentages of distinct states. In this study, we used the sandboxes created by simple process forking (rather than creating the more heavyweight virtualization layer) to demonstrate that even a small amount of instrumentation overhead can be mitigated by running tests only in previously-unseen states. The goal is to show that, even when the percentage of distinct states is relatively high, the new approach is still more efficient.

The results we present here are for a C implementation of the Sieve of Eratosthenes algorithm, which is given a single number as its parameter and returns a list of all prime numbers less than that number. We chose this program because it only uses one function, but takes a good deal of time to execute, so that we could get meaningful results over many executions. The experiment was conducted on a multi-processor 2.66GHz Linux machine with 2GB RAM.

For inputs, we used data files consisting of 100 random numbers, so that the function would run 100 times. We generated a number of different files with different percentages of distinct values, ranging from 0% distinct (meaning that all values were the same) to 100% distinct (meaning that all values were different).

Figure 4 shows the results of the experiment, using the average running time of 10 executions per data set. As expected, the running times for “always” running the In Vivo tests (as in the standard approach) and the time for “never” running tests are more or less constant; they are not exactly constant because of the different values used in the different data sets. More importantly, we see that “sometimes” running In Vivo tests (based only on previously-unseen states) usually outperforms “always” doing it, even with the additional instrumentation, and even when the percentage of distinct states is as high as around 90%.

The results of this experiment demonstrate that our approach does, in fact, make In Vivo Testing more efficient, assuming the percentage of distinct states in which a function is run is less than 90%. Further analysis will be required, however, to determine how true this assumption is in general.

6. LIMITATIONS AND FUTURE WORK

Although it is more efficient to run tests only in states that have not already been encountered, there is a memory cost associated with tracking all the previously-seen states. Regardless of how space efficient the solution may be, a program with many instrumented functions and many distinct program states could have fairly large memory requirements. Future work could assess the practical implications when it comes to additional memory usage.

Aside from the general issue related to memory cost, the specific prototype implementation we have presented here does have some limitations, based on the assumptions stated above. The use of the Cantor function to create a unique hash value for each state does have a practical upper bound in that we cannot store arbitrarily large values in a single variable in C or Java. We observed that the Cantor function can reach the limit of the “double” datatype depending on the values and the number of variables. A solution that scales to arbitrarily large states may not be able to take advantage of the speed of the O(1) hashing function and $O(\log_{256}n)$ lookup in the Judy Array, or would need to allow for false positives and/or false negatives.

Also, we have made some assumptions regarding the types of variables that can be tracked as part of the state, specifically limiting to primitive datatypes (int, float, char, etc.) but not complex objects such as Objects, structs, and such. This can conceivably be addressed by using type-specific hashing functions, analogous to the Cantor function used for numerical values; however, depending on the uniqueness of the hash codes, this too may introduce false positives, meaning that the system incorrectly believes that the current state has previously been observed. Clearly this would not be desirable, since the result would be that tests or analyses are not performed, even though they should be.

Future work could consider using distributed In Vivo Testing [3] to devise an approach so that tests are only run in *globally*-unseen states. It may also be possible to distribute the test cases in advance [9,11], so that a particular instance of the application is not concerned with *all* previously-seen states, only the ones it is responsible for.

7. RELATED WORK

As mentioned above, Elbaum and Hardojo [6] have surveyed other approaches to automating the testing of software in the field, including the monitoring, analysis, and profiling of deployed software, such as Gamma [15], Skoll [11], and Cooperative Bug Isolation [10]. All of these could benefit from a solution that automatically detects previously-encountered states. Others have investigated the use of application state to drive test case generation, e.g. [7], but in our case we assume that the tests already exist, and that the state is used to determine whether or not the tests should be executed.

Previous investigation of techniques for reducing the overhead of runtime monitoring and testing has included the use of static analysis to remove unnecessary instrumentation [21], or pre-determining when to execute uninstrumented “fast cases” instead of instrumented “slow cases” [10]. However, neither of these approaches has the goal of eliminating test cases at runtime based on previously-encountered states, and our techniques could be combined to reduce performance costs even further.

Much of the work in the representation of application state at runtime has focused on anomaly detection, i.e., determining that the application is in a state that is outside the range of what is expected [1,8]. These works also deal with the issue of “has this state been seen before?”, but the representation of state in those approaches is based on a finite state machine that considers the execution path up to that point, and not the set of variable values. However, future work could investigate how state-based anomaly detection techniques and the approach presented here could be combined, for instance by further simplifying the representation of expected states according to semantic equivalence.

8. CONCLUSION

Various approaches have been suggested for continuing to conduct testing and analysis of applications as they run in the deployment environment. Many such approaches, including fault detection, model checking, security testing, and fault localization, could be more efficient if tests and analyses were only conducted in previously-unseen application states, limiting the redundancy and reducing the performance overhead.

In this paper, we have presented an improvement to the implementation of the In Vivo Testing approach, such that tests are only executed in application states that the program has not previously encountered. We have demonstrated this improvement both theoretically and empirically, and discussed how our solution is applicable to various areas of testing and analysis of deployed software, indicating its potential for broad impact in the future.

Acknowledgments

We would like to thank Lori Clarke and Lee Osterweil for their helpful suggestions and discussions, and Huning Dai for his assistance with the implementation. Murphy and Kaiser are members of the Programming Systems Lab, funded in part by NSF grants CNS-0905246, CNS-0717544, CNS-0627473, and CNS-0426623, and NIH grant 1U54CA121852-01A1.

10. REFERENCES

[1]. Baah, GK.; Gray, A.; Harrold, MJ. On-line anomaly detection of deployed software: a statistical machine learning approach. Proc. of the 3rd International Workshop on Software Quality Assurance; 2006. p. 70-77.

[2]. Biskup, J. Security in computing systems challenges, approaches, and solutions. Springer-Verlag; 2009.

[3]. Chu, M.; Murphy, C.; Kaiser, G. Distributed in vivo testing of software applications. Proc. of the First International Conference on Software Testing, Verification and Validation; 2008. p. 509-512.

[4]. Company, H-P. Programming with Judy. 2001.

[5]. Dai, H.; Murphy, C.; Kaiser, G. Configuration fuzzing for software vulnerability detection. Proc. of the Fourth International Workshop on Secure Software Engineering (SecSE); 2010.

[6]. Elbaum, S.; Hardojo, M. An empirical study of profiling strategies for released software and their impact on testing activities. Proc. of the International Symposium on Software Testing and Analysis (ISSTA); 2004. p. 65-75.

[7]. Elbaum S, Rothermel G, Karre S, Fisher M. II. Leveraging user-session data to support web application testing. IEEE Transactions on Software Engineering March;2005 31(3):187–202.

[8]. Hangal, S.; Lam, MS. Tracking down software bugs using automatic anomaly detection. Proc. of the 24th International Conference on Software Engineering (ICSE); 2002. p. 291-301.

[9]. Keetha, N.; Wu, L.; Kaiser, G.; Yang, J. Distributed eXplode: A high-performance model checking engine to scale up state-space coverage. Department of Computer Science, Columbia University; 2008. Technical Report CUCS-051-08

[10]. Liblit, B.; Aiken, A.; Zheng, AX.; Jordan, MI. Bug isolation via remote program sampling. Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI); 2003. p. 141-154.

[11]. Memon, A.; Porter, A.; Yilmaz, C.; Nagarajan, A.; Schmidt, D.; Natarajan, B. Skoll: distributed continuous quality assurance. Proc. of the 26th International Conference on Software Engineering (ICSE); May. 2004 p. 459-468.

[12]. Murphy, C.; Kaiser, G. Metamorphic runtime checking of non-testable programs. Dept. of Computer Science, Columbia University; 2009. Technical Report CUCS-042-09

[13]. Murphy, C.; Kaiser, G.; Chu, M.; Vo, I. Quality assurance of software applications using the in vivo testing approach. Proc. of the Second IEEE International Conference on Software Testing, Verification and Validation; 2009. p. 111-120.

[14]. Norvig P. Techniques for automatic memoization with applications to context-free parsing. Computational Linguistics March;1991 17(1):91–98.

[15]. Orso, A.; Liang, D.; Harrold, MJ. Gamma system: Continuous evolution of software after deployment. Proc. of the International Symposium on Software Testing and Analysis (ISSTA); 2002. p. 65-69.

[16]. Osman, S.; Subhraveti, D.; Su, G.; Nieh, J. The design and implementation of Zap: A system for migrating computing environments. Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI); 2002. p. 361-376.

[17]. Osterweil, L. Perpetually testing software. Proc. of the The Ninth International Software Quality Week; May. 1996

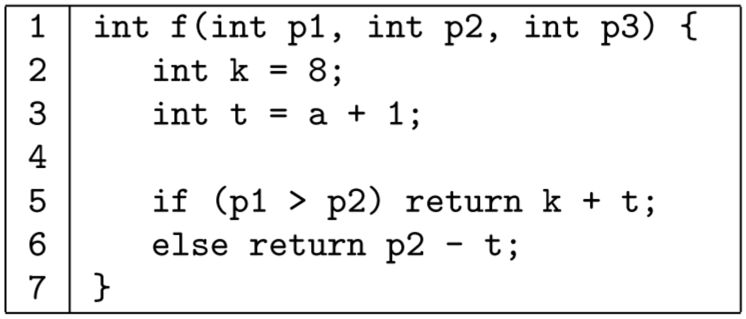
[18]. Pavlopoulou, C.; Young, M. Residual test coverage monitoring. Proc. of the 21st International Conference on Software Engineering (ICSE); May. 1999 p. 277-284.

[19]. Royden, HL. Real Analysis. Prentice Hall; 1988.

[20]. Sidiroglou, S.; Laadan, O.; Viennot, N.; Pérez, C-R.; Keromytis, AD.; Nieh, J. Assure: Automatic software self-healing using rescue points. Proc. of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems; 2009. p. 37-48.

[21]. Yong SH, Horwitz S. Using static analysis to reduce dynamic analysis overhead. Formal Methods in System Design November;2005 27(3):313–334.

[22]. Zeller, A. Isolating cause-effect chains from computer programs. Proc. of the 10th ACM SIGSOFT symposium on foundations of software engineering; 2002. p. 1-10.



```
int f(int p1, int p2, int p3) {
   int k = 8;
   int t = a + 1;

   if (p1 > p2) return k + t;
   else return p2 - t;
}
```

Figure 1.
Sample function. The Invite pre-processor scans the function looking for variables, to determine the function's dependencies.

```
int f1(int p1, int p2, int p3) {
   int k = 8;
   int t = a + 1;

   if (p1 > p2) return g(p3);
   else return p2 - t;
}

int g(int p) {
   int m = p * b;
   return m * m - b;
}
```

Figure 2.
Example of two functions, one of which inherits the set of dependencies from the other.

```
/* original function */
int __f(int p1, int p2) { ...  }

/* In Vivo test function */
boolean __INVtest_f(int p1, int p2) { ...  }

/* Determines whether the state has already been seen */
boolean __should_run_INVtest_f(int g, int p1, int p2) {
    /* use Cantor function to get distinct value for state */
    double value = Cantor(g, Cantor(p1, p2));
    /* determine whether value is already in Judy Array for this function */
    boolean alreadySeen = JudyArray_f.contains(value);
    if (alreadySeen) return false;
    else {
        JudyArray_f.add(value);
        return true; // indicates that test should be run
    }
}

/* wrapper function */
int f(int p1, int p2) {
    if (__should_run_INVtest_f(g, p1, p2)) {
        create_sandbox_and_fork();
        if (is_test_process()) {
            if (__INVtest_f(p1, p2) == false) fail();
            else succeed();
            destroy_sandbox();
            exit();
        }
    }
    return __f(p1, p2);
}
```

Figure 3.

Pseudo-code for wrapper of instrumented function, in which In Vivo tests are only executed in previously-unseen states

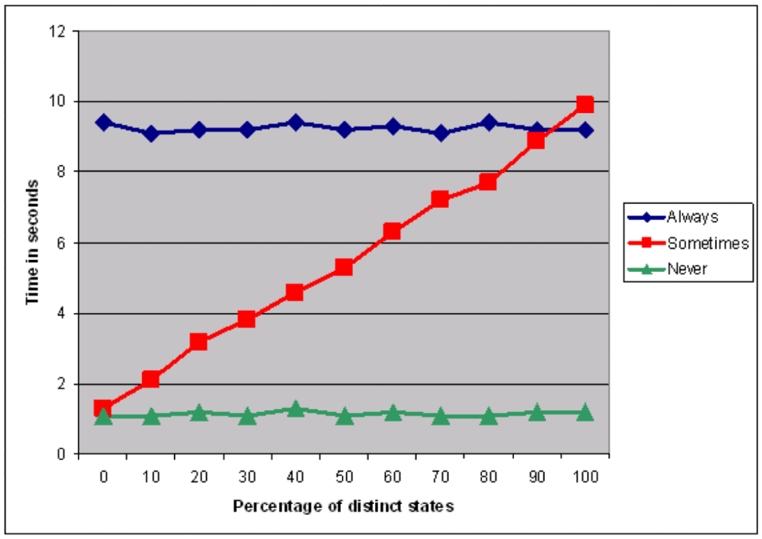


Figure 4.
Graph indicating performance caused by different variations in the percentage of distinct states.