Finding Errors in .NET with Feedback-Directed Random Testing:

February 18, 2008

Technical Report
MSR-TR-2008-29

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Finding Errors in .NET with Feedback-Directed Random Testing

Carlos Pacheco
MIT CSAIL
32 Vassar Street, 32-G714
Cambridge, MA 02139
cpacheco@mit.edu

Shuvendu K. Lahiri Thomas Ball
Microsoft Research
One Microsoft Way
Redmond, WA 98052
{shuvendu, tball}@microsoft.com

Abstract

We present a case study in which a team of test engineers at Microsoft applied a feedback-directed random testing tool to a critical component of the .NET architecture. Due to its complexity and high reliability requirements, the component had already been tested by 40 test engineers over five years, using manual testing and many automated testing techniques.

Nevertheless, the feedback-directed random testing tool found errors in the component that eluded previous testing, and did so two orders of magnitude faster than a typical test engineer (including time spent inspecting the results of the tool). The tool also led the test team to discover errors in other testing and analysis tools, and deficiencies in previous best-practice guidelines for manual testing. Finally, we identify challenges that random testing faces for continued effectiveness, including an observed decrease in the technique's error detection rate over time.

1. Introduction

Testing software is expensive. Estimates in the literature put the cost of testing at approximately half of the total development cost of software [Bei90]. At Microsoft, for example, there is approximately one tester for every developer. In addition to being expensive, testing software can be tedious and error-prone. A significant portion of a test engineer's work consists in constructing test inputs that run the software under different scenarios. Since it is impossible to exercise software under every possible scenario, test engineers must craft a small number of test inputs that reveal as many defects as possible. As the size and complexity of software increases, it becomes more difficult to cover all possible scenarios, and easier to miss test inputs that could have revealed an error.

Random testing [MFS90, Ham94, FM00, CH00, CS04, PLEB07, GHJ07] helps a test engineer create error-revealing test inputs, by mechanically and randomly sampling a program's input space. The effectiveness of random testing is an unresolved question in the testing community. Some studies [FK96, MAD$^+$03, VPP06, CGP$^+$06] suggest that random testing is not as effective as other test generation techniques such as chaining, bounded exhaustive testing, symbolic execution or model checking. Other studies [HT90, Nta98, GHJ07, PLEB07] suggest the opposite: that random testing's speed, scalability, and unbiased search make it an effective error-detection techniques and able to outperform many of the above techniques. For example, in previous work we described an experiment in which feedback-directed random testing [PLEB07], a variant of random testing, finds errors in many software libraries while model checking finds none, and achieves higher coverage than model checking and symbolic execution.

However, the real assessment of a test generation technique's effectiveness is its performance in the real world. Do previous evaluations of random testing measure the relevant variables? In an industrial setting, testing techniques are used under a very different set of constraints from a research setting. Practicing test engineers have tight deadlines and large amounts of code to test. For an automated test generation tool to succeed in this environment, it must reveal errors important enough to be fixed and it must reveal these errors in a cost-effective way, taking up less human time than manual testing or existing automated techniques. These qualities can be particularly difficult to measure in a research setting.

We present the results of a case study that sheds light on the effectiveness of random testing when used by test engineers in an industrial testing environment, and in comparison with the application of other test generation techniques. Engineers from a test team at Microsoft applied feedback-directed random test generation to a large component of the .NET Framework [dot] used by thousands of developers and millions of end users. The component under question sits low in the .NET framework stack, and many .NET applications depend on it for their execution. For this reason, the component has had approximately 40 testers devoted to testing it over a period of five years. It has undergone manual unit, system, and partition testing, as well as automated testing including fuzz, robustness, stress, and symbolic execution-based testing. Because of proprietary concerns, we cannot identify the .NET component analyzed. We will refer to it as "the .NET component" or "the component" from here on.

The case study provides new evidence grounded in industrial experience to the long-standing question about the effectiveness of random testing as an error-detection technique. The test team's knowledge of the average human effort required to manually find an error in the component under test allowed us to quantify the benefit of feedback-directed random testing compared to manual testing. Since the team has applied many testing techniques to the component, we were also able to learn about the effectiveness of feedback-directed random testing against these techniques.

Our main results are:

- Feedback-directed random test generation found more errors in 15 hours of human effort and 150 hours of CPU time than a test engineer typically finds in one year on code of the quality of the component under test. The technique found non-trivial errors, including errors that arise from highly specific sequences of operations. Moreover, these errors were missed by manual testing and by all previously-applied automated test generation techniques. Based on these results, the tool implementing feedback-directed random testing was added to a list of tools that other test teams at Microsoft are encouraged to use to improve the quality of their testing efforts.

- As a result of applying feedback-directed random testing to the component, the test team found and fixed errors in other automated testing tools, performed further manual testing on areas of the code that the technique showed to be insufficiently tested, and implemented new best practices for future manual testing efforts. In other words, the technique was used beyond bug finding, as an assessment tool for the test team's existing testing methodologies, and spurred more testing activities.
- After a highly productive error detection period, feedback-directed random testing plateaued and eventually stopped finding new errors, despite using different random seeds. This observation mirrors the results of a recent, unrelated study of random testing [GHJ07]. We provide a tentative explanation of the plateau effect for the case of feedback-directed random testing applied to the .NET component, and propose research directions to address the effect.

The rest of the paper is organized as follows. Sections 2 and 3 give an overview of the .NET component under test and feedback-directed random test generation. Section 4 describes the process that the test team used in applying feedback-directed random test generation to the component. Section 5 discusses the results, including the number and type of errors revealed, the reason why other techniques missed these errors, and the challenges that feedback-directed random testing faces in finding more errors over time. Section 6 surveys related work, and Section 7 concludes.

2. Overview of .NET Component

The software used in this study is a core component of the .NET Framework [dot]. It implements part of the functionality that allows managed code (code written in a high-level programming language like C#) to execute under a virtual machine environment. The component is required for any .NET application to execute. It is more than 100KLOC in size, written in C# and C++, and it exports its functionality in an API available to programmers both inside and outside Microsoft. Many applications written at Microsoft use the component, including the BCL (a library for I/O, graphics, database operations, XML manipulation, etc.), ASP.NET (web services and web forms), Windows Forms, SQL, and Exchange.

The software component has undergone approximately 200 man years of testing, not counting developer testing (most developers write unit tests). The test team has large computational resources for testing, including a cluster of several hundred machines.

The test team has tested the component using many techniques and tools. In addition to manual (unit and system) testing, the team has developed tools for performance, robustness, stress, and fuzz [FM00] testing. They have created tools that automatically test code for typical corner cases and values, such as uses of `null`, empty containers or arrays, etc. Additionally, thousands of developers and testers inside Microsoft have used pre-release versions of the component in their own projects and reported errors. To facilitate testing, the developers of the component make heavy use of assertions.

At this point, the component is mature and highly reliable. A dedicated test engineer working with existing methodologies and tools finds on average about 20 new errors *per year*. During the earlier years of its development cycle, this figure was much higher. One of the goals of this study was to determine if feedback-directed random testing could find errors not found by previous testing techniques, on software of the maturity level of the component.

3. Feedback-directed Random Testing

Feedback-directed random testing [PLEB07] addresses the automated generation of *unit tests* for object-oriented software. A unit test consists of a sequence of constructor and method calls and code that checks for expected behavior of the sequence. Feedback-directed random testing generates a set of test cases exhibiting error-revealing behaviors in the software under test. This section summarizes the technique presented in our previous work [PLEB07] by describing the RANDOOP unit test generator.

RANDOOP (**Rand**om Tester for **O**bject-**O**riented **P**rograms) implements feedback-directed random testing for .NET (another version of the tool exists for Java [PE07]). The tool is fully automatic. Figure 1 shows the architecture of RANDOOP. It takes as input the location of an assembly, a time limit after which test generation stops, and optionally a set of configuration files that let the user specify (via regular expressions) subsets of classes and methods in the assembly that should be tested or avoided.

RANDOOP outputs unit tests that can be compiled and executed to produce error-revealing behavior in a method under test. The error-revealing behaviors that RANDOOP checks for are assertion violations, access violations, and unexpected program termination. Figure 2 shows an example error-revealing unit test case generated by RANDOOP. The test case shows a test input that leads to an assertion violation (realized as an `AssertionViolationException`). Lines 13—16 comprise the test input, and lines 17—30 comprise the test oracle. Exit codes signal different execution outcomes (if the code under test is non-deterministic, the outcome may differ in different executions). The exit codes can be used by tools to post-process RANDOOP-generated test cases.

RANDOOP creates method sequences incrementally by randomly selecting a method call to apply, and selecting input arguments to the method from among previously constructed sequences. As soon as it is created, a new sequence is executed and checked against a set of error-revealing behaviors. RANDOOP uses the result of the execution to determine if the sequence is error-revealing, new, or illegal:

- *Error-revealing:* the execution exhibits an error-revealing behavior. Sequences that lead to error-revealing behavior are output to the user. Figure 2 shows an example error-revealing method sequence. Sequences classified as error-revealing are not used to create new sequences (such an extension would amount to exploring off an already-corrupted state, which would lead to many false positives).
- *New:* the objects that the sequence constructs are not equivalent to objects constructed by a previously created input. RANDOOP considers two objects `o1` and `o2` to be equivalent if `o1.equals(o2)` returns `true`. The tool maintains a cached set of all the objects created during generation, and checks if a new sequence creates new objects (This heuristic did not improve performance for the .NET component under test, and we did not use it in the case study.) Sequences that create new objects and are not error-revealing are output to the user as *normal behavior* test cases and can be used for regression testing.
- *Illegal:* execution of the sequence leads to an exception suggesting that the input is illegal. For example, a sequence that throws an `ArgumentException` when `null` is used as input to a method

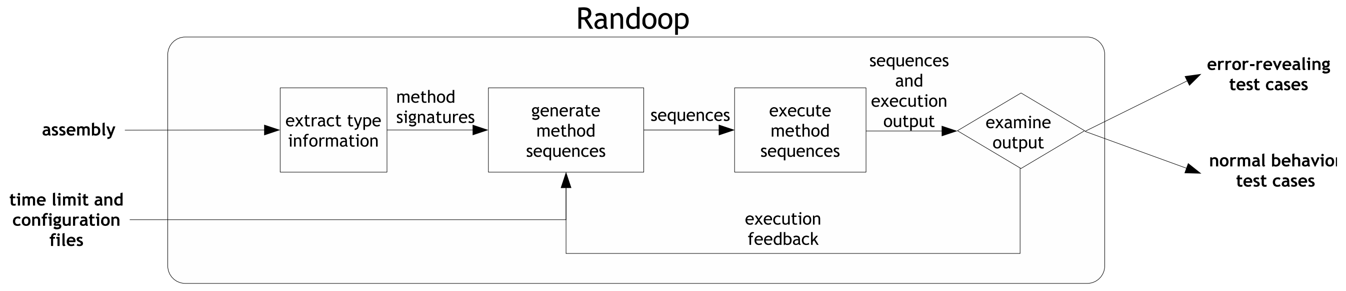
Randoop
assembly
time limit and configuration files
extract type information
method signatures
generate method sequences
sequences
execute method sequences
sequences and execution output
examine output
execution feedback
error-revealing test cases
normal behavior test cases

Figure 1. RANDOOP's architecture. The input to the tool is an assembly, a time limit, and optionally, a set of configuration files. RANDOOP creates method sequences using the public methods and constructors exported by the assembly, executes the sequences, and based on their execution, may output them as error-revealing or regression test cases.

```
// A Randoop-generated unit test for method
//
// ConfManager.LoadConfigFromFile(String,ConfigType).
//
// When executed, the test causes the method
// to raise an assertion violation.
public class RandoopTest4065
{
 public static int Main()
 {
  try
  {
     int v1 = 2;
     String v2 = Convert.ToString(v1);
     ConfigType v3 = ConfigType.User;
     Config v4 = ConfManager.LoadConfigFromFile(v2,v3);
  }
  catch (AssertionViolationException e)
  {
     Console.WriteLine("Test threw an");
     Console.WriteLine("AssertionViolationException.");
     Console.WriteLine("Will exit with code 1.");
     return 1;
  }
  catch (Exception e)
  {
     Console.WriteLine("Unexpected behavior:");
     Console.WriteLine("expected an");
     Console.WriteLine("AssertionViolationException.");
     return 2;
  }
 }
}
```

Figure 2. Example RANDOOP-generated unit test case. The test case reveals an error in method `LoadConfigFromFile` which leads to an assertion violation when the method is executed.

is heuristically classified as illegal. Sequences classified as illegal are discarded and are not used to create new sequences.

RANDOOP outputs error-revealing sequences. Before outputting an error-revealing sequence, RANDOOP attempts to *minimize* it by iteratively omitting method calls that can be removed from the method sequence while preserving its error-revealing behavior.

Robustness. Before starting the case study, we modified RANDOOP to make it more robust. The tool executes method sequences in the same process where it executes its own code, using .NET's reflection infrastructure. This increases the speed of the tool by an order of magnitude compared to compiling and executing each sequence in a separate process. In early runs of RANDOOP on the component under study, some method sequences caused the tool to be forcefully terminated by the operating system, due to execution of sequences containing methods that attempted to perform a low-level OS operation for which RANDOOP had no privileges. Thus, RANDOOP sometimes terminated before the user-specified time limit. To improve RANDOOP's robustness, we wrapped RANDOOP in another program, RandoopWrapper (Figure 3). RandoopWrapper takes the user's input, spawns a RANDOOP process on the given input, and monitors the execution of the RANDOOP process. If RANDOOP crashes before the user-specified time limit is reached, RandoopWrapper spawn a new RANDOOP process, using a new random seed. Method sequences that lead to RANDOOP crashes are also output as potentially error-revealing test cases.

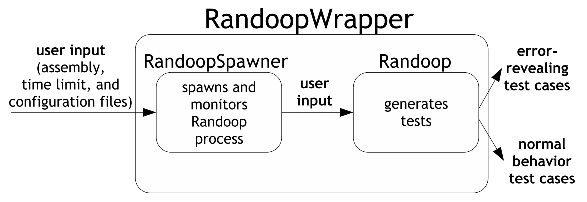
RandoopWrapper
user input (assembly, time limit, and configuration files)
RandoopSpawner
spawns and monitors Randoop process
user input
Randoop
generates tests
error-revealing test cases
normal behavior test cases

Figure 3. RandoopWrapper is a wrapper process around Randoop that makes it more robust to crashes caused by arbitrary code execution. First, RandoopSpawner spawns a Randoop process. If Randoop crashes, RandoopSpawner spawns a new Randoop process with a new random seed. This continues until the user-specified time limit expires.

4. Process

We gave a copy of RANDOOP to the test team, along with instructions on how to run the tool. Since the tool is fully automatic, works directly on assemblies, and it outputs compilable, error-revealing test cases, the test team had no trouble understanding the purpose of the tool. The test team started using RANDOOP with its default settings: one minute generation time limit and no configuration files.

As they discovered errors, the test team created error reports and assigned engineers to fix them. It was not always possible to fix the code immediately, so the test team altered RANDOOP's configuration files to instruct it not to explore methods that led to error-revealing behavior. This prevented RANDOOP from rediscovering the same error in subsequent runs.

As they became more familiar with the tool, they used the tool in more sophisticated ways, creating different configuration files to focus on different parts of the component. An aspect that made the tool easy to adopt was its scalability. The technique does not

total number of tests generated	4,000,000
distinct errors revealed by RANDOOP	30
total CPU time required to reveal the errors	150 hours
total human time spent interacting with RANDOOP	15 hours
average errors revealed by a tester in 1 year of testing	20

Figure 4. Case study statistics.

analyze the code, but simply runs it, which makes it possible to test even code that executes deep into the operating system.

We met with the test team on a regular basis to discuss their experience with the tool, including the time they had spent using it and inspecting its results, as well as its effectiveness compared with their existing manual test suites and testing tools. Based on requests by the test team, we also implemented a number of new configurable options, such as the ability to output all sequences that RANDOOP generated, regardless of their classification (Section 5.1.2 discusses the way that the test team used this option).

5. Results

Figure 4 summarizes the results of the test team's effort. RANDOOP revealed 30 serious, previously unknown errors in 15 hours of human effort (spread among several days) and 150 hours of CPU time. Each error was entered as a bug report; many have since been fixed. The 15 hours of human effort included inspecting the error-revealing tests output by RANDOOP. To place the results numbers in context, recall that for a code base of the component's level of maturity, a test engineer will find approximately 20 errors per year.

The kinds of behaviors that the tool currently checks for (assertion violations, access violations, and unexpected termination) are almost always indicative of errors in the code, so false positives were not as much a problem as *redundant tests:* test that were syntactically distinct but revealed the same error in the implementation. The hours reported include time spent inspecting and discarding redundant tests.

In terms of human effort, a test engineer using RANDOOP *revealed more errors in 15 hours than he would be expected to find in a year using previous testing methodologies and tools.*

5.1 Error characteristics

This section present the observed characteristics of the errors that RANDOOP found, and representatives examples. Each section presents an observation followed by examples.

5.1.1 Errors in well-tested code

RANDOOP revealed errors in code on which previous tests had achieved full block and arc coverage. An example is an error dealing with memory management and native code. The component code base is a combination of memory-managed (garbage collected) code as well as native code with explicit memory allocation. When native code manipulates references from managed objects, it must inform the garbage collector of any changes (new references, references that can be garbage-collected, etc.).

RANDOOP created a test input that caused an internal portion of the component to follow a previously untested path through a method. This path caused the native code to erroneously report a previously used local variable as containing a new reference to a managed object. In this specific path, the address of the reference was an illegal address for a managed object (less than 32k but greater than 0). In a checked build (the version of the component used during testing, which includes assertion-checking code), the component checks for the legality of the addresses, and threw an assertion violation stating that a bad value was given as a reference into the garbage-collected heap.

The erroneous code was in a method for which existing tests achieved 100% block coverage and 100% arc coverage. After fixing the error, the test team added a new regression test and also reviewed (and added test cases) for similar methods. This is an example of an error discovered by RANDOOP that led to testing for more errors of the same kind, reviewing existing code, and adding new tests.

Feedback-directed test generation revealed errors in code in which existing tests achieved 100% code coverage.

5.1.2 Using RANDOOP's output as input to other tools

At the beginning of the study, we expected RANDOOP to be used as an end-to-end tool. However, the test team started using RANDOOP's test inputs (which are stand-alone executable files) as input to other tools, getting more functionality from each generated test. The test team requested that we add an execution to RANDOOP in which it outputs all the test inputs it creates, even if they were not error-revealing. Their goal was to use other tools to execute the inputs under different environments in order to discover new errors.

Among the tools that they used were stress and concurrency testers. An example is a tool that invokes the component's garbage collector after every few instructions, or a tool that runs several RANDOOP-generated test inputs in a stress tool that executes a single tests input multiple times in parallel (with a separate thread executing the same input). This process led the test team to discover more errors. Using the latter tool, the test team discovered a race condition that was due to incorrect locking of a shared resource. The error was revealed only after a specific sequence of actions by a method, involving locking an object, performing an operation, and finally calling another method that reset the state of the thread. The team fixed the error in the method that reset the thread state, implemented a tighter protocol around the specific behavior, and did a review of similar constructs (the review found no other issues).

The test team used RANDOOP*'s generated tests as input to other testing tools, increasing the scope of the exploration and the types of errors revealed beyond those that* RANDOOP *could find.*

5.1.3 Testing the test tools

In addition to finding errors directly in the component, RANDOOP led the test team to discover errors in their existing testing and program analysis tools. An example of this is an error in a static analysis tool that involved a missing string resource. In the component, most user-visible strings (for example, exception messages) are stored in a text file called a resource file. The resource file is included with the product binary at build time, and is accessed when a string is needed during execution. This approach simplifies language localization.

The test team had previously built a simple analysis tool to detect unused or missing resource strings. The tool inspects the component source code and checks that each resource is referenced at least once in the code, and that the resource exists. However, the tool had a bug and it failed to detect some missing strings. RANDOOP generated a test input that caused an infrequently-used exception type to be raised. When the virtual machine looked up the exception message string, it did not find the string and, in the checked build, led to a fatal assertion violation. On a retail build (the version of the component shipped to customers), the missing string produced a meaningless exception message.

After adding back the missing resource, the test team fixed the error in their resource checking tool and did further manual testing on the tool to verify that it worked properly.

In addition to revealing errors in the .NET component, RANDOOP *revealed errors in the test team's testing and program analysis tools.*

5.1.4 Corner cases and further testing

For software of high complexity, it is difficult for a team of testers to partition the input space in a way that ensures that all important cases will be covered. While RANDOOP makes no guarantees about covering all relevant partitions, its randomization strategy led it to create test cases for which no manual tests were written. As a result, feedback-directed random testing discovered many missed corner cases.

The knowledge gained from the discovery of the corner cases led the test team to consider new corner cases, write new tests, and find more errors. In some cases, the discovery of a new error led the test team to augment an exiting testing tool with new checks for similar corner cases, and in other cases the discovery of an error led the test team to adopt new practices for manual testing.

The first example is an error that uncovered a lack of testing for empty arrays. The component has a container class that accepts an array as input to initialize the contents of the container. The initialization code checks the legality of the input data by iterating over the array and checking that each element is a legal element for the container. An empty array is legal. One of access methods expected did not handle the case in which the input array is empty. In this case, the method incorrectly assumed that it was an array of bytes and started reading bytes starting from the base address of the array. In most cases, this would quickly lead to a failure due to malformed data, and in other cases (one created by RANDOOP), the method would fail with an access violation. The test team fixed the error, reviewed other access methods, and as a result fixed other similar issues. As a result of this error, the team updated their "best practices" to include empty arrays as an important input to test.

This area of the component contained a large number of tests for different kinds of initialization arrays. However, the sheer size of the state space made it impossible to test all possible combinations of inputs, and the manual tests were incomplete.

The second example is an error that uncovered a lack of testing for I/O streams that have been closed. When an I/O stream is closed, subsequent operations on the stream should fail. A RANDOOP-generated test showed that calling a successive set of state-manipulating methods on a closed stream would lead to one of the operations succeeding. In the specific case, RANDOOP generated a call sequence that would create a specific stream, do some operations and then close the underlying stream. The component has many test cases that test for similar behaviors, i.e. testing that operations on closed streams fail in specific ways. Again, due to the size of the component, some important cases were missing. In a checked build the test case caused an assertion violation, and on a retail build it led to being able to access certain parts of the stream after its closed. The error has been fixed, test cases have been added, and reviews of similar code have been completed.

The errors that RANDOOP *revealed led to further testing activities unrelated to the initial random testing effort, including writing new manual tests and adopting new practices for manual testing.*

5.2 Comparison with Other Test Generation Techniques

The errors that RANDOOP revealed were not revealed using the team's existing methodologies and tools, including a very large collection of manually-written unit and system tests, partition testing, fuzz testing, and program analysis tools like the one described in Section 5.1.3. Conversely, there were many errors revealed by previous efforts not revealed by RANDOOP. In other words, RANDOOP was not subsumed by, and did not subsume, other techniques.

According to the test team, a major disadvantage of RANDOOP in comparison with manual and non-random automated techniques is RANDOOP's lack of a meaningful stopping criterion. After several hours of running RANDOOP without the tool producing a new error-revealing test case, they did not know whether RANDOOP had essentially exhausted its power and was "done" finding all the errors that it would find, or whether more computer resources would lead to new errors. For example, towards the end of the study, the test team ran RANDOOP for many hours on several dedicated machines but the tool did not reveal any new errors. Other techniques have more sensible stopping criteria. When writing manual tests, a test team typically has a code coverage goal; a static analysis tool terminates when the analysis is complete; symbolic execution based techniques terminate when they have attempted to cover all feasible paths, etc.

The experiences of the test team suggests that RANDOOP enjoys two main benefits compared with non-random automated test generation approaches. One is its scalability. For example, concurrently with using RANDOOP, the test team used a new test generator that outputs tests similar to RANDOOP, but uses symbolic execution [Kin76], a technique that instruments the code under test to collect path constraints, and attempts to solve the constraints in order to yield test inputs that exercise specific branches. The symbolic execution tool was not able to find any new errors in the component. One of the reasons is that the tool depends on a constraint solver to generate tests that cover specific code paths, and the solver slowed down the tool and was not always powerful enough to generate test cases. In the amount of time it took the symbolic execution tool to generate a single test, RANDOOP was able to generate many test cases.

The second main benefit is that RANDOOP's (and more generally, random testing's) randomized search strategy protects test engineers against human bias. For example, in Section 5.1.4 we discuss that the test team had not previously considered testing a set of methods using empty arrays, and RANDOOP revealed an error elicited via an empty array. The test team had previously omitted empty arrays from testing because the engineer that crafted the test cases for the method in question did not consider empty arrays an interesting test case at the time.

Human bias is not limited to manual testing; automated tools can suffer from the biases of their creators. For example, the test team has created automated testing tools that test methods that take multiple input parameters, using all possible combinations of a small set of inputs for each parameter slot. RANDOOP revealed errors that were not revealed using the inputs programmed into the tool.

RANDOOP's randomized search revealed errors that manual and non-random automated techniques missed because of human bias or lack of scalability. However, RANDOOP has no clear stopping criterion, which makes it difficult to gauge when the tool has exhausted its effectiveness.

Randoop versus fuzz testing. Previous to RANDOOP, the component had undergone extensive fuzz testing [FM00] on nearly every format and protocol of the component. Like feedback-directed random testing, fuzz testing is also unbiased, but previous fuzzing efforts did not reveal the errors that RANDOOP revealed.

A reason for this is that fuzz testing has been traditionally been done on data-intensive software that take as inputs files, network packets, etc. Fuzzing is less frequently applied to domains that deal with both data and control, such as method sequences in object-oriented libraries. The errors that RANDOOP found turned out to be about both data (the input to methods) and about control (the specific sequence of methods). In order to discover data errors, some amount of control structure was necessary (a sequence of method calls), and in order to discover control errors, some data was necessary (inputs to methods). RANDOOP helped bridge the divide between data and control.

Fuzzing is effective in generating test inputs that explore either data or control. When the structure of inputs includes both data and control, feedback-directed method sequence generation can be more effective.

5.3 The Plateau Effect

The errors revealed by RANDOOP did not emerge at a uniform rate during the testing process. Instead, the rate of error discovery was quick at first and then decreased. During the first two hours of use, RANDOOP revealed 10 distinct errors (5 errors per hour). Then the error-finding rate fell to 2 errors per hour for approximately 10 hours. After that, RANDOOP ceased to output error-revealing test cases.

To make RANDOOP more effective, the test team tried different strategies, such as creating different configuration files that targeted specific portions of the component. These efforts revealed more errors, but did not alter the trend towards diminishing returns for the effort expended. Towards the end of the case study, the test team switched from running RANDOOP on a desktop to running it on parallel in a cluster of several hundred machines, using different combinations of random seeds and configurations. These runs revealed fewer errors than the initial runs on a single desktop.

Groce et al. [GHJ07] also observed this effect using a technique similar to RANDOOP's to generate tests consisting of sequences of file system operations for flight software.

Understanding the plateau effect. Given a software artifact composed of multiple components, a reasonable requirement for a test generation tool is that it be *fair*, meaning that it distribute its computational resources fairly among the different components. When analyzing RANDOOP's output, we discovered that RANDOOP is not fair. RANDOOP selects which method to test next uniformly at random from among the set of all public methods in the assembly under test (the same applies to constructors). This strategy can lead to some classes being explored more than others. A class that defines five constructors will be explored more heavily than a class that defines one constructor. A class that defines a nullary constructor (a constructor that requires no arguments) will be explored more heavily than a class whose constructor requires an object of a type that is difficult to create.

RANDOOP focuses on classes that declare nullary constructors or that define several constructors, at the expense of classes whose constructors require more complex setup. Because RANDOOP is incremental and creates new method sequences from previously created sequences, the initial favoring of a few classes leads to a feedback loop in which classes that were initially easier to create are focused on more in later stages of generation, while classes that are difficult to create become starved.

After an initial period of effectiveness, feedback-directed random test generation yielded diminishing results. Other exploration strategies may extend the technique's period of effectiveness.

Overcoming the plateau effect. We do not yet have a solution to the fairness problem. Below, we sketch a possible approach, with the caveat that we have not yet evaluated its effectiveness. The idea is to use a distribution other than RANDOOP's uniform random distribution. In fact, it may be desirable to use an *adaptive distribution*, as follows. At the start of generation, RANDOOP could maintain a mapping from each constructor and method to a "weight" number indicating the relative probability that the given method or constructor will be selected for testing (members with higher weight have a higher change of being selected). At the beginning, all methods and constructors would have equal weight. At regular intervals, RANDOOP would update the weights: members for which more tests have been created would be given lower weight, and members for which fewer tests have been created would be given higher weight. Thus for example, a class with several constructors or with a unary constructor, for which RANDOOP quickly generates many inputs, would eventually receive a lower weight, which would increase the chances that the tool explores other classes. Other techniques for achieving fairness are possible; the one we have outlined is one possibility.

6. Related Work

Random testing. Researchers have used random testing [Ham94] to reveal errors in many applications, including Unix utilities [MFS90], Windows GUI applications [FM00], Haskell programs [CH00], and object-oriented code programs [CS04, PE05, Ori05]. Research tools for random testing object-oriented code include JCrasher [CS04], Jartege [Ori05], Autotest [CL05], Eclat [PE05], and RANDOOP [PLEB07]; commercial tools include Jtest [Par] and Agitator [Agi]. JCrasher [CS04] creates test inputs by using a parameter graph to find method calls whose return values can serve as input parameters. Jartege [Ori05] and AutoTest [CL05] require a formal specification, and use it to determine whether randomly-generated method calls are error-revealing. Autotest lets the user vary a number of generation parameters and distributions. Eclat [PE05] generates random sequences of method calls and classifies them as normal, illegal, or error revealing based on an operational model derived from an existing test suite. RANDOOP [PLEB07] focuses on generating a set of behaviorally-diverse test inputs, including state matching to prune redundant objects, repetition to generate low-likelihood sequences, oracles based on API contracts that can be extended by the user, and regression oracles that capture the behavior of a program when run on the generated input. Jtest [Par] is a commercial tool that also uses specifications to determine if an input is error-revealing; its input generation algorithm is not published. Agitator [Agi] creates test inputs using a variety of techniques, including random generation and data flow analysis, and proposes to the user program invariants based on execution of the test inputs.

Other approaches combine random generation with more sophisticated techniques in an attempt to achieve higher coverage of the code under test. Ferguson and Korel [FK96] proposed an input generation technique that begins by executing the program under test with a random input, and systematically modifies the input so that it follows a different path. Recent work by Godefroid et al. and Sen et al. [GKS05, SA06] explores DART and CUTE, two related symbolic execution approaches that integrate random input generation.

Evaluations of random testing. People tend to believe that random testing is a poor testing methodology. Glenford Myers sums up the feeling in his *Art of Software Testing* book: "In terms of likelihood of detecting the most errors, a randomly selected collection of test cases has little chance of being an optimal, or close to optimal, subset." [MS04]. However, this intuition is not backed by experimental evidence, and in fact theoretical studies have shown that random testing can be as effective as more systematic techniques such as partition testing [HT90, Nta98].

The literature contains relatively few empirical evaluations of random testing. As mentioned in the introduction, previous evaluations have reported that random testing performs poorly compared with other techniques. For example, Ferguson and Korel describe an experiment where randomly-generated inputs achieve less code coverage than their chaining technique [FK96]. Marinov et al. [MAD$^+$03] describe an experiment where randomly-generated test inputs kill fewer mutants than inputs generated using Korat. Visser et al. [VPP06] describe the results of experiments where random testing achieves less coverage than model checking and symbolic execution.

In theoretical studies, Hamlet and Taylor [HT90] and Ntafos [Nta98] conclude that random testing can be as effective as partition testing. Groce et al. [GHJ07] describe a case study in which model checking would not scale to testing code for complex flight systems, while random testing scales and finds many errors in the code. In previous work, we describe an experiment in which feedback-directed random testing [PLEB07], a variant of random testing, finds errors in many component libraries while model checking finds none. The paper shows that when augmented with execution-feedback heuristics, random testing outperforms the same benchmarks used in [VPP06].

We know of few industrial case studies of random testing's use by an actual product group to test a real product. Recent work by Groce et al. [GHJ07] describes a realistic application of random testing to a file system used in flight software. Their experience was similar to ours: random testing found a large number of errors in a relatively short amount of time. Like RANDOOP, their tool eventually reached a plateau. To find more errors, Groce et al. suggest moving towards formal verification of the software; this is a different, and complementary approach to our attack on the plateau via techniques to increase fairness.

7. Conclusion

The goal of this case study was to determine the effectiveness of a random testing technique when used by practicing test engineers in an industrial setting. Our focus on a component that had already undergone large amounts of testing allowed us to reach clearer conclusions about the relative effectiveness of feedback-directed random testing, by setting the bar high for the technique, and by allowing us to compare with the many previous techniques applied to the component.

The technique proved highly effective in testing even an extremely well-tested component, and led to a highly productive period of error discovery. In addition, the technique revealed errors in the test team's existing testing and analysis tools, holes in their manual testing, and even led them to improve their best practices. These results provide evidence of the promise of feedback-directed random testing in improving the quality of software, and the productivity of the engineers that test it.

Random testing can greatly increase the effectiveness of the test engineer by creating test cases that reveal not only errors, but alert a test engineer to potential biases in the crafting of manual tests or automated tools. This can have a positive impact on the quality of testing beyond the errors found directly by random testing. RANDOOP proved effective in uncovering these biases. Since the study was conducted, other test teams at Microsoft have used RANDOOP to find errors in software components and improve the quality of their testing efforts.

References

[Agi] Agitar. Agitator tool, http://www.agitar.com.

[Bei90] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[CGP$^+$06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[CH00] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP '00, Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, Montreal, Canada, September 18–20, 2000.

[CL05] I. Ciupa and A. Leitner. Automatic testing based on design by contract. In *Proceedings of Net.ObjectDays 2005 (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World)*, pages 545–557, September 19-22 2005.

[CS04] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, September 2004.

[dot] Microsoft .NET Framework. `http://www.microsoft.com/net/`.

[FK96] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.

[FM00] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows System Symposium*, pages 59–68, Seattle, WA, USA, August 2000.

[GHJ07] Alex D. Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, IL, USA, June 13–15, 2005.

[Ham94] Dick Hamlet. Random testing. In *Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[HT90] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software*

Engineering, 16(12):1402–1411, December 1990.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[MAD$^+$03] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT/LCS/TR-921, MIT Laboratory for Computer Science, September 2003.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[Nta98] Simeon Ntafos. On random and partition testing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 42–48, New York, NY, USA, 1998. ACM Press.

[Ori05] Catherine Oriat. Jartege: A tool for random generation of unit tests for Java classes. In *QoSA/SOQUA*, pages 242–256, September 2005.

[Par] Parasoft. JTest manuals version 6.0. Online manual.

[PE05] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 25–29, 2005.

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM, October 2007.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[SA06] Koushik Sen and Gul Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006. (Tool Paper).

[VPP06] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for Java containers using state matching. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 37–48, New York, NY, USA, 2006. ACM Press.