Pex–White Box Test Generation for .NET

Nikolai Tillmann and Jonathan de Halleux

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
{nikolait,jhalleux}@microsoft.com

Abstract. Pex automatically produces a small test suite with high code coverage for a .NET program. To this end, Pex performs a systematic program analysis (using dynamic symbolic execution, similar to path-bounded model-checking) to determine test inputs for Parameterized Unit Tests. Pex learns the program behavior by monitoring execution traces. Pex uses a constraint solver to produce new test inputs which exercise different program behavior. The result is an automatically generated small test suite which often achieves high code coverage. In one case study, we applied Pex to a core component of the .NET runtime which had already been extensively tested over several years. Pex found errors, including a serious issue.

1 Overview

Pex [24] is an automatic white-box test generation tool for .NET. Starting from a method that takes parameters, Pex performs path-bounded model-checking by repeatedly executing the program and solving constraint systems to obtain inputs that will steer the program along different execution paths, following the idea of dynamic symbolic execution [12,6]. Pex uses the theorem prover and constraint solver Z3 [3] to reason about the feasibility of execution paths, and to obtain ground models for constraint systems.

While the concept of dynamic symbolic execution is not new, Pex extends the previous work in several ways:

- Pex can build faithful symbolic representations of constraints that characterize execution paths of *safe* .NET programs. In other words, Pex contains a complete symbolic interpreter for safe programs that run in the .NET virtual machine. (And the constraint solver Z3 comes with decision procedures for most such constraints.)
- Pex can reason about a commonly used set of *unsafe* features of .NET. (*Unsafe* means unverifiable memory accesses involving pointer arithmetic.)
- Pex employs a set of search strategies with the goal to achieve high statement coverage in a short amount of time.

We have integrated Pex into Visual Studio as an add-in. Pex can generate test-cases that can be integrated with various unit testing frameworks, including NUnit [20] and MSTest [22]. Pex is an extensible dynamic program analysis

B. Beckert and R. Hähnle (Eds.): TAP 2008, LNCS 4966, pp. 134–153, 2008.

platform; one recent plug-in is DySy [7], an invariant inference tool based on dynamic symbolic execution. We are working towards making the symbolic execution analysis compositional [1].

We have conducted a case study in which we applied Pex to a core component of the .NET architecture which had already been extensively tested over five years by approximately 40 testers. The component is the basis for other libraries, which are used by thousands of developers and millions of end users. Pex found errors, including a serious issue. Because of proprietary concerns, we cannot identify the .NET component on which this case study was based. We will refer to it as the "core .NET component" in the following.

The rest of the paper is structured as follows: Section 2 contains an introduction to Pex. Section 3 discusses the implementation of Pex in more detail. Section 4 shows a particular application of Pex to unsafe .NET code. Section 5 presents the results of applying Pex to a core .NET component. Section 6 compares Pex with other related technologies, and Section 7 concludes.

2 An Introduction to Pex

2.1 Parameterized Unit Testing

At its core, Pex is a test input generator. A test input generator is only useful in practice

- if we have a program to generate test inputs for, and
- if we have a test oracle that decides whether a program execution was successful for some given test inputs.

For Pex, we have adopted the notion of *parameterized unit tests* [28,29] which meet both requirements. A parameterized unit test is simply a method that takes parameters, performs a sequence of method calls that exercise the code-under-test, and asserts properties of the code's expected behavior.

For example, the following parameterized unit test written in C# creates an array-list with a non-negative initial capacity, adds an element to the list, and then asserts that the added element is indeed present.

```
[PexMethod]
public void AddSpec(
    // data
    int capacity, object element) {
    // assumptions
    PexAssume.IsTrue(capacity >= 0);
    // method sequence
    ArrayList a = new ArrayList(capacity);
    a.Add(element);
    // assertions
    Assert.IsTrue(a[0] == element);
}
```

Here, `AddSpec` is decorated with the *custom attribute* `[PexMethod]`, which Pex uses to distinguish parameterized unit tests from ordinary methods.

2.2 The Testing Problem

Starting from parameterized unit tests as specification, we formulate the testing problem as follows.

> Given a sequential program P with statements S, compute a set of program inputs I such that for all reachable statements s in S there exists an input i in I such that $P(i)$ executes s.

Remarks:

- By *sequential* we mean that the program is single-threaded.
- We consider failing an assertion, or violating an implicit contract of the execution engine (e.g. `NullReferenceException` when `null` is dereferenced) as special statements.

2.3 The Testing Problem in Practice

In general, the reachability of program statements is not decidable. Therefore, in practice we aim for a good approximation, e.g. high coverage of the statements of the program. Instead of statement coverage, other coverage metrics such as arc coverage can be used.

In a system with dynamic class loading such as .NET, it is not always possible to determine the statements of the programs ahead of time. In the worst case, the only way to determine all *reachable statements* is an incremental analysis of all possible behaviors of the program.

The analysis of all possible program behaviors, i.e. all execution paths, may take an infinite amount of time. In practice, we have only a limited amount of time available, so we aim for an analysis that can produce test inputs for most reachable statements fast.

Another problem arises from the fact that most interesting programs interact with the environment. In other words, the semantics of some program statements may not be known ahead of time. Most static analysis tools make conservative assumptions in such cases and may produce many *false positives*, e.g. test-cases that supposedly may exhibit an error, but in practice do not. For test generation tools it is more appropriate to take into account environment interactions in order to filter out false positives.

In the remainder of this section we describe the foundations on which Pex tries to address the testing problem, and the next section describes Pex' implementation in more detail, including how heuristic search strategies often solve the problem of achieving high coverage fast.

2.4 Symbolic Execution

Pex implements a white box test input generation technique that is based on the concept of *symbolic execution*. Symbolic execution works similar to concrete

execution, only that symbolic variables are used for the program inputs instead of concrete values. When a program variable is updated to a new value during program execution, then this new value may be an expression over the symbolic variables. When the program executes a conditional branch statement where the condition is an expression over the symbolic variables, symbolic execution has to consider two possible continuations, since the condition may evaluate to either `true` or `false`, depending on the program inputs. For each path explored by symbolic execution in this way, a *path condition* is built over symbolic variables. The path condition is the conjunction of the expressions that represent the branch conditions of the program. In this manner all constraints are collected which are needed to deduce what inputs cause an execution path to be taken.

A constraint solver or automatic theorem prover is used to decide the feasibility of individual execution paths, and to obtain concrete test inputs as representatives of individual execution paths.

2.5 Dynamic Symbolic Execution

Pex explores the reachable statements of a parameterized unit test using a technique called *dynamic symbolic execution* [12,6]. This technique consists in executing the program, starting with very simple inputs, while performing a symbolic execution in parallel to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution. Then Pex uses a constraint solver to compute variations of the previous inputs in order to steer future program executions along different execution paths. In this way, all execution paths will be exercised eventually.

Dynamic symbolic execution extends conventional static symbolic execution [16] with additional information that is collected at runtime, which makes the analysis more precise [12,11]. While additional information is collected by monitoring concrete traces, each of these traces is representative of an execution path, i.e. the equivalence class of test inputs that steer the program along this particular execution path. By taking into account more details of structure of the program (e.g. boundaries of basic blocks or functions), even bigger equivalences classes can be analyzed at once [12,1].

Algorithm 2.1 shows the general dynamic symbolic execution algorithm implemented in Pex. The choice of the new program inputs i in each loop iteration decides in which order the different execution paths of the program are enumerated.

Pex uses several heuristics that take into account the structure of the program and the already covered statements when deciding on the next program inputs. While the ultimate goal of Pex is to discover all reachable statements, which is an undecidable problem, in practice Pex attempts to achieve high statement coverage fast. This simplifies the configuration of Pex greatly: the user just has to set a time limit or another rough exploration bound. Other dynamic symbolic execution tools ([12,11,12,6]) perform an exhaustive search of all the execution paths in a fixed order, within bounds on the size and structure of the input given

by the user. In the case of Pex the inputs are often richly structured object graphs for which it is a difficult problem to define practical and useful bounds.

Algorithm 2.1. Dynamic symbolic execution

```
Set J := ∅                                  (intuitively, J is the set of already
loop                                                analyzed program inputs)
    Choose program input i ∉ J              (stop if no such i can be found)
    Output i
    Execute P(i); record path condition C   (in particular, C(i) holds)
    Set J := J ∪ C                          (viewing C as the set {i | C(i)})
end loop
```

2.6 More Reasons for *Dynamic* Symbolic Execution

Symbolic execution was originally proposed [16] as a static program analysis technique, i.e. an analysis that only considered the source code of the analyzed program. This approach works well as long as all decisions about the feasibility of execution paths can be made on basis of the source code alone. It becomes problematic when the program contains statements that cannot be reasoned about easily (e.g. memory accesses through arbitrary pointers, or floating point arithmetic), or when parts of the program are actually unknown (e.g. when the program communicates with the *environment*, for which no source code is available, and whose behavior has not been specified rigorously).

It is not uncommon for .NET programs to use unsafe .NET features, i.e. using pointer arithmetic to access memory for performance reasons, and most .NET programs interact with other unmanaged (i.e. non-.NET) components or the Windows API for legacy reasons.

While static symbolic execution algorithms do not use any information about the environment into which the program is embedded, dynamic symbolic execution can leverage dynamic information that it observes during concrete program executions, i.e. the memory locations which are actually accessed through pointers and the data that is passed around between the analyzed program and the environment.

As a result, Pex can prune the search space. When the program communicates with the environment, Pex builds a model of the environment from the actual data that the environment receives and returns. This model is an under-approximation of the environment, since Pex does not know the conditions under which the environment produces its output. The resulting constraint systems that Pex builds may no longer accurately characterize the program's behavior. In practice this means that for a computed input the program may not take the predicted execution path. Since Pex does not have a precise abstraction of the program's behavior in such cases, Pex may not discover all reachable execution paths, and thus all reachable statements.

In any case, Pex always maintains an under-approximation of the program's behavior, which is appropriate for testing.

3 Pex Implementation Details

3.1 Instrumentation

Pex monitors the execution of a .NET program through code instrumentation. Pex plugs into the .NET profiling API [21]. It inspects the instructions of a method in the intermediate language [15] which all .NET compilers target. Pex rewrites the instructions just before they are translated into the machine code at runtime. The instrumented code drives a "shadow interpreter" in parallel to the actual program execution. The "shadow interpreter"

- constructs symbolic representations of the executed operations over logical variables instead of the concrete program inputs;
- maintains and evolves a symbolic representation of the entire program's state at any point in time;
- records the conditions over which the program branches.

Pex' "shadow interpreter" models the behavior of all verifiable .NET instructions precisely, and models most unverifiable (involving unsafe memory accesses) instructions as well.

3.2 Symbolic Representation of Values and Program State

A symbolic program state is a predicate over logical variables together with an assignment of expressions over logical variables to locations, just as a concrete program state is an assignment of values to locations. For Pex, the locations of a state consist of static fields, instance fields, method arguments, local variables, and positions on the operand stack.

Pex' expression constructors include primitive constants for all basic .NET data types (integers, floating point numbers, object references), and functions over those basic types representing particular machine instructions, e.g. addition and multiplication. Pex uses tuples to represent .NET value types ("structs") as well as indices of multi-dimensional arrays, and maps to represent instance fields and arrays, similar to the heap encoding of ESC/Java [10]: An instance field of an object is represented by a *field map* which associates object references with field values. (For each declared field in the program, there is one location in the state that holds current field map value.) An array type is represented by a class with two fields: a length field, and a field that holds a mapping from integers (or tuples of integers for multi-dimensional arrays) to the array elements. Constraints over the .NET type system and virtual method dispatch lookups are encoded in expressions as well. Predicates are represented by boolean-valued expressions.

We will illustrate the representation of the state with the following class.

```
class C {
    int X;
    int GetXPlusOne() { return this.X + 1; }
    void SetX(int newX) { this.X = newX; }
}
```

Symbolically executing the method `c.GetXPlusOne()` with the receiver object given by the object reference `c` will yield the expression `add(select(X_Map,c), 1)` where the `select` function represents the selection of `c`'s `X`-field value from the current field map `X_Map`. After symbolically executing `c.SetX(42)`, the final state will assign the expression `update(X_Map,c,42)` to the location that holds the current field map of `X`. `X_Map` denotes the value of the field map of `X` before the execution of the method.

Pex implements various techniques to reduce the enormous overhead of the symbolic state representation. Before building a new expression, Pex always applies a set of reduction rules which compute a normal form. A simple example of a reduction rule is constant folding, e.g. $1 + 1$ is reduced to 2. All logical connectives are transformed into a binary decision diagram (BDD) representation with if-then-else expressions [5]. All expressions are hash-consed, i.e. only one instance is ever allocated in memory for all structurally equivalent expressions. Map updates, which are used extensively to represent the evolving heap of a program, are compactly stored in tries, indexed over unique expression indices.

Based on the already accumulated path condition, expressions are further simplified. For example, if the path condition already established that $x > 0$, then $x < 0$ simplifies to `false`.

3.3 Symbolic Pointers

Pex represents pointers as expressions as well. Pex distinguishes the following pointer constructors.

- Pointer to nowhere. Represents an invalid pointer, or just `null`.
- Pointer to value. Represent a pointer to an immutable value, e.g. a pointer to the first character of a string.
- Pointer to static field.
- Pointer to instance field map. Represents a pointer to the mapping of an instance field that associates object references with field values.
- Pointer to method argument or local variable.
- Pointer to element. Given a pointer to a mapping and an index expression, represents a pointer to the indexed value.

While the pointers in safe, managed .NET programs are guaranteed to be either `null` or pointing to a valid memory location, unsafe .NET code that is sufficiently trusted to bypass .NET's byte code verifier does not come with such a guarantee. Thus, when the user enables Pex' strict pointer checking mode, Pex builds a verification condition whenever the program is about to perform an indirect memory access through a pointer. In particular, given a pointer to an element of an array, the condition states that the index must be within the bounds of the array.

In practice, the verification conditions can verify most uses of unsafe pointers. For example, the following code shows a common use of pointers, with the intention of simply avoiding the overhead of repeated array-bounds checking.

```
public unsafe bool BuggyContainsZero(byte[] a) {
    if (a == null || a.Length == null) return false;
    fixed (byte* p = a)
        for (int i = 0; i <= a.Length; i++)
            if (p[i] == 0) return true;
    return true;
}
```

This code contains an error: The loop condition should be `i < a.Length` instead of `i <= a.Length`. This error might not be detected with conventional testing, since reading beyond the bounds of an array with a pointer often does not trigger an exception (the allocated memory is usually advanced to another block of allocated memory).

While the problem of buffer overflows has been well studied, e.g. in the context of C programs that are compiled to machine code directly, we are not aware of a thorough checker in the context of managed execution environments, in particular .NET.

Pex can not only detect the error in strict pointer-checking mode, Pex will even steer the program towards obscure program behaviors by test input generation through dynamic symbolic execution.

However, Pex cannot symbolically reason about all operations that involve pointers. In particular, Pex does not track when the content of a memory is reinterpreted, e.g. a pointer to an array of bytes is cast to a pointer of an integer, and when the memory was obtained from the environment, e.g. through a call to a Windows API.

3.4 Search Strategy

Deciding reachability of program statements is a hard problem. In a system with dynamic class loading and virtual method dispatch the problem does not become easier. As discussed earlier, Pex' approach based on dynamic symbolic execution enumerates feasible execution paths, where information from previously executed paths is used to compute test inputs for the next execution paths. Most earlier approach to dynamic symbolic execution [12,27,26,6] in fact only use information from the last execution path to determine test inputs that will exercise the next path. This restriction forces them to use a fixed "depth-first, backtracking" search order, where the next execution path would always share the longest possible prefix with the previous execution path. As a result, a lot of time may be spent analyzing small parts of the program before moving on. These approaches require well defined bounds on the program inputs to avoid unfolding the same program loop forever, and they may discover "easy" to cover statements only after an exhaustive search. (To avoid getting stuck in the depth-first search, these earlier approaches frequently inject random test inputs to steer the search towards other parts of the program. However, this prevents any deep symbolic analysis.)

Pex uses the information of all previously executed paths: During exploration, Pex maintains a representation of the explored execution tree of the program,

whose paths are the explored execution paths. In each step of the test generation algorithm, Pex picks an outgoing unexplored branch of the tree, i.e. the prefix of a feasible execution path plus an outgoing branch that has not been exercised yet. The next test inputs are the solution (if any) of the constraint system that is built from the conjunction of the path condition of the feasible path prefix, and the condition of the unexercised outgoing branch. If the constraint system has no solution, or it cannot be computed by the constraint solver, the search marks the branch as infeasible and moves on.

In earlier experiments, we tried well-known search strategies to traverse the execution tree, such as breadth-first search. While this strategy does not get stuck in the same way as depth-first search, it does not take into account the structure of the program either.

The program consists of building blocks such as methods and loops, which may get instantiated and unfolded many times along each execution path, giving rise to multiple branch instances in the execution path (and tree). For our ultimate goal, to cover all reachable statements, the number of unfoldings is irrelevant, although a certain number of unfoldings might be required to discover that a statement is reachable. How many and which unfoldings are required is undecidable.

In order to avoid getting stuck in a particular area of the program by a fixed search order, Pex implements a fair choice between all such unexplored branches of the explored execution tree. Pex includes various fair strategies which partition all branches into equivalence classes, and then pick a representative of the least often chosen class. The equivalence classes cluster branches by mapping them

- to the branch statement in the program of which the execution tree branch is an instance (each branch statement may give rise to multiple branch instances in the execution tree, e.g. when loops are unfolded),
- to the stack trace at the time the brach was recorded,
- to the overall branch coverage at the time the branch was recorded,
- to the depth of the branch in the execution tree.

Pex combines all such fair strategies into a meta-strategy that performs a fair choice between the strategies.

Creating complex objects. When an argument of a parameterized unit test is an object that has non-public fields, Pex will still collect constraints over the usage of that field. Later, new test inputs may be computed which assign particular values to those fields. But then Pex may not know how to create an object through the publicly available constructors such that the object's private fields are in the desired state. (Of course, Pex could use .NET's reflection mechanism to set private fields in arbitrary ways, but then Pex might violate the (implicit) class invariant.)

In such cases, Pex selects a constructor of the class (the user may configure which constructor is chosen), and Pex includes this constructor in the exploration of the parameterized unit test. As a result, Pex will first try to find a non-exceptional path through the control-flow of the constructor, and then use the

created object to further explore the parameterized unit test that required the object. In other words, Pex tries to avoid the backward search to find a way to reach a target state; instead, it will perform a forward search that is compatible with dynamic symbolic execution.

In this way, directed object graphs can easily be created, where arguments to constructors can refer to earlier constructed objects. Cyclic object graphs can only result if a constructor updates a field of an argument to point to the constructed object.

As an alternative to employing only existing constructors to configure objects, the user may also provide factory methods, which could invoke a sequence of method calls to construct and configure a new object, possibly creating cyclic references as well.

3.5 Constraint Solving

For each chosen unexplored branch, Pex builds a formula that represents the condition under which this branch may be reached. Pex performs various preprocessing steps to reduce the size of the formula before handing it over to the constraint solver, similar to constraint caching, and independent constraint optimization [6].

Pex employs Z3 as its constraint solver. Pex faithfully encodes all constraints arising in safe .NET programs such that Z3 can decide them with its built-in decision procedures for propositional logic, fixed sized bit-vectors, tuples, arrays, and quantifiers. Arithmetic constraints over floating point numbers are approximated by a translation to rational numbers. Pex also encodes the constraints of the .NET type system and virtual method dispatch lookups as universally quantified formulas.

3.6 Pex Architecture

Internally, Pex consists of several libraries:

Microsoft.ExtendedReflection. Extended Reflection (ER) is a library that enables the monitoring of .NET applications at the instruction level. It uses the unmanaged profiling API to instrument the monitored .NET program with callbacks to the managed ER library. The callbacks are used to drive the "shadow interpreter" mentioned in Section 3.1.

Microsoft.Pex.Framework. This library is serves as a front-end for the user to configure Pex. It defines a number of .NET custom attributes, including the `PexMethod` attribute that we used in the earlier example.

Microsoft.Pex. The Pex engine implements the search for test inputs, by repeatedly executing the program while monitoring it, and building constraint systems to obtain new test inputs.

Microsoft.Z3. [3] is the constraint solver that Pex uses.

Pex is built from individual components, that are organized in three layers:

1) A set of components is alive for the entire lifetime of the Pex engine. 2) In addition, a set of components is created and kept alive for the duration of the exploration of a single parameterized unit test. 3) In addition, a set of components is created and kept alive for each execution path that is executed and monitored.

Pex' monitoring library, ER, is a quite general monitoring library that can be used in isolation. In addition to Pex itself, we have built PexCop on top of ER, a dynamic program analysis application which analyzes individual execution traces, looking for common programming errors, e.g. resource leaks.

Pex itself provides an extension mechanism, where a user can hook into any of the three component layers of Pex (engine, exploration, path). For example, DySy [7], an invariant inference tool based on dynamic symbolic execution, uses this extension mechanism to analyze all execution path of a parameterized unit test.

3.7 Limitations

There are certain situations in which Pex cannot analyze the code properly:

Nondeterminism. Pex assumes that the analyzed program is deterministic; this means in particular that all environment interactions should be deterministic. Pex detects non-determinism by comparing the program's actual execution path with the predicted execution path. When non-deterministic behavior is detected, Pex prunes the test inputs that caused it. Pex also gives feedback to the user, showing the program branches where monitored execution paths began to deviate from the prediction. The user can decide to ignore the problems, or the user can change the code to make it more testable.

To alleviate the problem, Pex has a mechanism for substituting methods that have a known non-deterministic behavior with deterministic alternatives. For example, Pex routinely substitutes the `TickCount` property of the `System.Environment` class that measures time with a constant alternative. Substitutions are easy to write by users; they are applied by Pex through name matching.

```
namespace __Substitutions.System {
    public static class Environment {
        public static int get_TickCount___redirect() {
            return 0;
        }
    }
}
```

Concurrency. Today, Pex does not handle multithreaded programs. Pex only monitors the main thread of the program. Other threads may cause non-deterministic behavior of the main thread, which results in feedback to the user just like other non-deterministic program behavior.

Native Code, .NET code that is not instrumented. Pex does not monitor native code, e.g. x86 instructions called through the P/Invoke mechanism of .NET. Also, since instrumentation of managed code comes with a significant performance overhead, Pex instruments code only selectively. In both cases, the effect is the same: constraints are lost. However, even if some methods are implemented in native code or are uninstrumented, Pex will still try to cover the instrumented code as much as possible.

The concept of redirecting method calls to alternative substitution methods is also used sometimes to give managed alternatives to native methods, so that Pex can determine the constraints of native methods by monitoring the managed alternative.

Symbolic Reasoning. Pex uses an automatic constraint solver (Z3) to determine which values are relevant for the test and the program-under-test. However, the abilities of the constraint solver are, and always will be, limited. In particular, Z3 cannot reason about floating point arithmetic, and Pex imposes a configurable memory and time consumption limit on Z3.

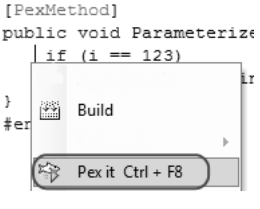
Language. Pex can analyze arbitrary .NET programs, written in any .NET language. Today, the Visual Studio add-in and the test code generation only support C#.

4 Application

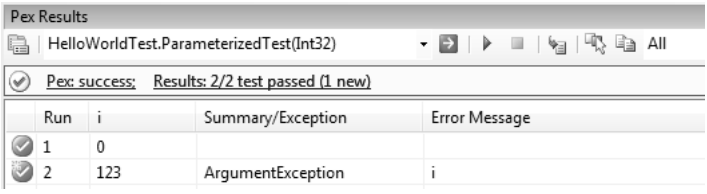
Pex is integrated into Visual Studio as an add-in. The user writes parameterized unit tests as public instance methods decorated the custom attribute `PexMethod`, as shown in the following example.

```
[PexMethod]
public void ParameterizedTest(int i) {
    if (i == 123)
        throw new ArgumentException("i");
}
```

Then, the user simply right-clicks the parameterized unit test, and selects the **Pex It** menu item.



Pex will then launch a process in the background which analyzes the code, executing it multiple times. The results are shown in a **Pex Results** window, that lists the computed parameter values in a table for each parameterized unit test.



As expected, Pex generated 2 tests to cover `ParameterizedTest`. The first tests uses the "default" value 0 for an integer, and Pex records the constraint `i!=123`. The negation of this constraint leads to the second test, where `i==123`, which triggers the branch that throws a `ArgumentNullException`.

In the following example, we show that Pex can analyze unsafe managed .NET code. We wrote the following parameterized unit test, that obtains an unsafe pointer from a (safe) byte array, then passes the pointer to the .NET `UnmanagedMemoryStream`, which is in turn given to the `ResourceReader`.

```
[PexClass]
...
public partial class ResourceReaderTest {
    [PexMethod]
    public unsafe void ReadEntriesFromUnmanagedMemoryStream(
        [PexAssumeNotNull]byte[] data) {
        fixed (byte* p = data)
            using (UnmanagedMemoryStream stream =
                new UnmanagedMemoryStream(p, data.Length)) {
                ResourceReader reader =
                    new ResourceReader(stream);
                readEntries(reader);
            }
    }

    private static void readEntries(ResourceReader reader) {
        int i = 0;
        foreach (DictionaryEntry entry in reader) {
            PexAssert.IsNotNull(entry.Key);
            i++;
        }
    }
}
```

We further decorate the test with the following attributes, to suppress certain exceptions that the documentation deems acceptable, and to enable Pex' strict checking of unsafe memory accesses.

```
[PexInjectExceptionsOnUnverifiableUnsafeMemoryAccess]
[PexAllowedException(typeof(BadImageFormatException))]
[PexAllowedException(typeof(IOException))]
[PexAllowedException(typeof(NotSupportedException))]
```

From the parameterized unit test, Pex generates several test inputs. After around one minute, and executing the parameterized unit tests for 576 times with different inputs, Pex generates test-cases such as the following. (Most of the generated test-cases represent invalid resource file, but some represent legal resource files with one or more entries. The byte array shown here is an illegal resource file.)

```
public void ReadEntriesFromUnmanagedMemoryStream_576() {
    byte[] bs0 = new byte[56];
    bs0[0] = (byte)206;
    bs0[1] = (byte)202;
    bs0[2] = (byte)239;
    bs0[3] = (byte)190;
    bs0[7] = (byte)64;
    bs0[12] = (byte)2;
    bs0[16] = (byte)2;
    bs0[24] = (byte)192;
    bs0[25] = (byte)203;
    bs0[26] = (byte)25;
    bs0[27] = (byte)176;
    bs0[28] = (byte)1;
    bs0[29] = (byte)145;
    bs0[30] = (byte)88;
    bs0[40] = (byte)34;
    bs0[41] = (byte)128;
    bs0[42] = (byte)132;
    bs0[43] = (byte)113;
    bs0[44] = (byte)132;
    bs0[46] = (byte)168;
    bs0[47] = (byte)5;
    bs0[48] = (byte)172;
    bs0[49] = (byte)32;
    this.ReadEntriesFromUnmanagedMemoryStream(bs0);
}
```

Pex deduced the entire file contents from the `ResourceReader` implementation. Note that the first four bytes represent a magic number which the `ResourceReader` expects. The later bytes form resource entries. The following code is part of the resource reader implementation. `ReadInt32` combines four bytes to a 32-bit integer through bitwise operations.

```
// Read ResourceManager header
// Check for magic number
// _store wraps the input stream
int magicNum = _store.ReadInt32();
if (magicNum != ResourceManager.MagicNumber)
    throw new ArgumentException("Resource file not valid!");
```

5 Evaluation

We applied Pex on a core .NET component that had already been extensively tested over several years.

We used a version of the component which contains assertion checks that the developers of the component embedded into the code. These checks are very expensive, and they are removed from the retail version of the component that is normally deployed by the users. These additional consistency checks, realized by conditional branch instructions, greatly increase the number of potential execution paths that must be analyzed. As a result, Pex analysis takes at least an order of magnitude longer than it does when applied on the retail version.

We used the Pex Wizard to generate individual parameterized unit tests for each public method of all public classes. These automatically generated unit tests do not contain any additional assertion validation; they simply pass the arguments through to the method-under-test. Thus, the test oracle only consists of the assertions that are embedded in the product code, and the pattern that certain exceptions should not be thrown by any code, e.g. access violation exceptions that indicate that an unsafe operation has corrupted the memory. In addition, we wrote about ten parameterized unit tests by hand which exercise common call sequences.

For example, for a method `Parse` that creates a data type `DataType` instance by parsing a string, the Wizard generates parameterized unit tests such as the following.

```
[PexMethod]
public void Parse(string s) {
    DataType result = DataType.Parse(s);
    PexValue.AddForValidation("result", result);
}
```

The parameterized unit test calls `DataType.Parse` with a given string and stores the result in a local variable. The call to `PexValue.AddForValidation` logs the result of the call to `Parse`, and it the test suite which Pex creates will include verification code that can be used in future regression testing to ensure that the `Parse` will not change its behavior but always return the same output as when Pex explored it.

We ran Pex on about 10 machines (different configurations, similar to P4, 2GHz, 2GB RAM) for three days; each machine was processing one class at a time.

In total, the analysis involved more than 10,000 public methods with more than 100,000 blocks and more than 100,000 arcs. When executing the code as part of the analysis, Pex created a sand-box with security permissions “Internet”, i.e. permissions that correspond to the default policy permission set suitable for content from unknown origin, which means in particular that most operations involving the environment, e.g. file accesses, were blocked, Starting from the public methods, Pex achieved about 43% block coverage and 36% arc coverage. We do not know how many blocks and arcs are actually reachable.

Table 1. Automatically achieved coverage on selected classes of the core .NET component

Class	Blocks	Block Coverage	Arcs	Arc Coverage
A (mostly stateless methods)	>300	95%	>400	90%
B (mostly stateless methods)	>100	97%	>200	94%
C (stateful)	>200	76%	>300	65%
D (parsing code)	>500	81%	>800	73%
E (numerical algorithms)	>400	71%	>600	67%
F (numerical algorithms)	>100	82%	>200	79%
G (numerical algorithms)	>100	98%	>100	97%
H (numerical algorithms)	>200	71%	>200	61%
I (numerical algorithms)	>200	97%	>300	96%

Because of the restricted security permissions, and the fact that Pex was only testing one method at a time, the overall coverage numbers clearly can be improved. However, Pex did very well on many classes which do not require many method calls to access their functionality. Table 1 shows a selection of classes of the core .NET component on which Pex fully automatically achieved high block and arc coverage. Only lower bounds for the block and arc numbers are given for proprietary reasons.

One category of errors that Pex found contains test cases that trigger rather benign exceptions, e.g. `NullReferenceException` and `IndexOutOfRangeException`. Another more interesting category of 17 unique errors involves the violation of assertions which the developers wrote in the code, and the exhaustion of memory, and other serious issues.

Most of the errors that Pex found required very carefully chosen argument values (e.g. a string of length 100 filled with particular characters), and it is unlikely that a random test input generator would find them. While some of the errors could be found by assertion-targetting techniques, e.g. [18], the branch conditions that guarded the errors were usually quite complex (involving bitvector arithmetic, indirect memory accesses) and were spread over multiple methods, and incorporated values obtained from the environment (here, the Windows API). It requires a dynamic analysis (to obtain the values from the environment) with a precise symbolic abstraction of the program's behavior to find these errors.

6 Related Work

Pex performs path-bounded model-checking of .NET programs. Pex is related to other program model checkers, in particular JPF [2] and XRT [14] which also operate on managed programs (Java and .NET). Both JPF and XRT have extensions for symbolic execution. However, both can only perform *static* symbolic execution, and they cannot deal with stateful environment interactions. Also, in the case of JPF, only some aspects of the program execution can be encoded

symbolically (linear integer arithmetic constraints), while others must always be explored explicitly (constraints over indirect memory accesses).

The idea of symbolic execution was pioneered by [16]. Later work on dynamic test generation, e.g. [17,18], mainly discussed the generation of test inputs to determine whether a particular execution path or branch was feasible. While Pex' search strategies try to exercise individual execution paths in a particular (heuristically chosen) sequence, the strategies are complete and will eventually exercise all execution paths. This is important in an environment such as .NET where the program can load new code dynamically, and not all branches and assertions are known ahead of time.

Dynamic symbolic execution was first suggested in DART [12]. Their tool instruments C programs at the source code level, and it tracks linear integer arithmetic constraints. CUTE [27] follows the approach of DART, but it can track and reason about not only linear integer arithmetic, but also pointer aliasing constraints. jCUTE [26] is an implementation of CUTE for Java, a managed environment without pointers. EXE [6] is another implementation of C source code based dynamic symbolic execution, and EXE implements a number of further improvements, including constraint caching, independent constraint optimization, bitvector arithmetic, and tracking indirect memory accesses symbolically. Each of these approaches is specialized for a particular source language, and they only include certain operations in the symbolic analysis. Also, their search order is not prioritized to achieve high coverage quickly, which forces the user to precisely define bounds on the size of the program inputs and to perform an exhaustive search. Pex is language independent, and it can symbolically reason about pointer arithmetic as well as constraints from object oriented programs. Pex search strategies aim at achieving high coverage fast without much user annotations.

Another language agnostic tool is SAGE [13], which is used internally at Microsoft. It virtualizes a Windows process on the x86 instruction level, and it tracks integer constraints as bitvectors. While operating at the instruction level makes it a very general tool, this generality also comes with a high instrumentation overhead which is significantly smaller for Pex.

Several improvements have been proposed recently to improve the scalability of dynamic symbolic execution, by making it compositional [11,19], and demand-driven [19,8]. We are working on related improvements in Pex [1] with encouraging early results.

Randoop [23] is a tool that generates new test-cases by composing previously found test-case fragments, supplying random input data. Randoop was also used internally in Microsoft to test core .NET components. While Pex and Randoop found some of the same errors, the error findings were generally different in that Randoop found errors that needed two or more method calls, while most of the errors that Pex found involved just a single method calls, but with very carefully chosen argument values.

The commercial tool AgitarOne from Agitar [4] generates test-cases for Java by analyzing the source code, using information about program invariants

obtained in a way similar to [9]. Similar to idea of parameterized unit testing [25], work building on Agitar proposes a concept called theories [25] to write and explore general test-cases.

7 Conclusion

Pex [24] is an automatic white-box test generation tool for .NET that explores the code-under test by dynamic symbolic execution. Pex analyzes safe, managed code, and it can validate unsafe memory accesses on individual execution paths. We applied Pex on a extremely well tested core .NET component, and found errors, including a serious issue. The automatically achieved results are encouraging. However, the combined coverage of the test-cases that Pex generated fully automatically clearly show that there is room for future research, e.g. leveraging information about the structure of the program to construct method call sequences automatically.

Acknowledgements

We would like to thank Wolfram Schulte for his support, our interns Thorsten Schuett, Christoph Csallner and Saswat Anand for their work to improve Pex, Nikolaj Bjorner and Leonardo de Moura for Z3, the developers and testers of the core .NET component for their support and advice, as well as Patrice Godefroid, and the anonymous reviewers for their comments.

References

1. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. Technical Report MSR-TR-2007-138, Microsoft Research, Redmond, WA (October 2007)
2. Anand, S., Pasareanu, C.S., Visser, W.: Jpf-se: A symbolic execution extension to java pathfinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
3. Bjorner, N., de Moura, L.: Z3: An efficient SMT solver (2007), `http://research.microsoft.com/projects/Z3`
4. Boshernitsan, M., Doong, R., Savoia, A.: From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In: ISSTA 2006: Proceedings of the 2006 international symposium on Software testing and analysis, pp. 169–180. ACM Press, New York (2006)
5. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: DAC 1990: Proceedings of the 27th ACM/IEEE conference on Design automation, pp. 40–45. ACM Press, New York (1990)
6. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: CCS 2006: Proceedings of the 13th ACM conference on Computer and communications security, pp. 322–335. ACM Press, New York (2006)

7. Csallner, C., Tillmann, N., Smaragdakis, Y.: Dysy: Dynamic symbolic execution for invariant inference. Technical Report MSR-TR-2007-151, Microsoft Research, Redmond, WA (November 2007)
8. Engler, D., Dunbar, D.: Under-constrained execution: making automatic code destruction easy and scalable. In: ISSTA 2007: Proceedings of the 2007 international symposium on Software testing and analysis, pp. 1–4. ACM, New York (2007)
9. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming (2007)
10. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proc. the ACM SIGPLAN 2002 Conference on Programming language design and implementation, pp. 234–245. ACM Press, New York (2002)
11. Godefroid, P.: Compositional dynamic test generation. In: POPL 2007: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 47–54. ACM Press, New York (2007)
12. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. SIGPLAN Notices 40(6), 213–223 (2005)
13. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. Technical Report MSR-TR-2007-58, Microsoft Research, Redmond, WA (May 2007)
14. Grieskamp, W., Tillmann, N., Schulte, W.: XRT - Exploring Runtime for .NET - Architecture and Applications. In: SoftMC 2005: Workshop on Software Model Checking, July 2005. Electronic Notes in Theoretical Computer Science (2005)
15. E. International. Standard ECMA-335, Common Language Infrastructure (CLI) (June 2006)
16. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
17. Korel, B.: A dynamic approach of test data generation. In: IEEE Conference On Software Maintenance, November 1990, pp. 311–317 (1990)
18. Korel, B., Al-Yami, A.M.: Assertion-oriented automated test data generation. In: Proc. the 18th international conference on Software engineering, pp. 71–80. IEEE Computer Society, Los Alamitos (1996)
19. Majumdar, R., Sen, K.: Latest: Lazy dynamic test input generation. Technical Report UCB/EECS-2007-36, EECS Department, University of California, Berkeley (Mar 2007)
20. Two, M.C., Poole, C., Cansdale, J., Feldman, G., Newkirk, J.W., Vorontsov, A.A., Craig, P.A.: NUnit, `http://www.nunit.org/`
21. Microsoft. Net framework general reference - profiling (unmanaged api reference), `http://msdn2.microsoft.com/en-us/library/ms404386.aspx`
22. Microsoft. Visual Studio Team System, Team Edition for Testers, `http://msdn2.microsoft.com/en-us/vsts2008/products/bb933754.aspx`
23. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE 2007, Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, USA, May 23–25 (2007)
24. Pex development team. Pex (2007), `http://research.microsoft.com/Pex`
25. Saff, D., Boshernitsan, M., Ernst, M.D.: Theories in practice: Easy-to-write specifications that catch bugs. Technical Report MIT-CSAIL-TR-2008-002, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, January 14 (2008)

26. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
27. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 263–272. ACM Press, New York (2005)
28. Tillmann, N., Schulte, W.: Parameterized unit tests. In: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 253–262. ACM, New York (2005)
29. Tillmann, N., Schulte, W.: Unit tests reloaded: Parameterized unit testing with symbolic execution. IEEE Software 23(4), 38–47 (2006)