Execution Generated Test Cases: How to Make Systems Code Crash Itself

Cristian Cadar and Dawson Engler*

Computer Systems Laboratory,
Stanford University,
Stanford, CA 94305, U.S.A

Abstract. This paper presents a technique that uses code to automatically generate its own test cases at run-time by using a combination of symbolic and concrete (i.e., regular) execution. The input values to a program (or software component) provide the standard interface of any testing framework with the program it is testing, and generating input values that will explore all the "interesting" behavior in the tested program remains an important open problem in software testing research. Our approach works by turning the problem on its head: we lazily generate, from within the program itself, the input values to the program (and values derived from input values) as needed. We applied the technique to real code and found numerous corner-case errors ranging from simple memory overflows and infinite loops to subtle issues in the interpretation of language standards.

1 Introduction

Systems code is difficult to test comprehensively. Externally, systems interfaces tend towards the baroque, with many different possible behaviors based on tricky combinations of inputs. Internally, their implementations tend towards heavily entangling nests of conditionals that are difficult to enumerate, much less exhaust with test cases. Both features conspire to make comprehensive, manual testing an enormous undertaking, so enormous that empirically, many systems code test suites consist only of a handful of simple cases or, perhaps even more commonly, none at all.

Random testing can augment manual testing to some degree. A good example is the fuzz [3, 4] tool, which automatically generates random inputs, which is enough to find errors in many applications. Random testing has the charm that it requires no manual work, other than interfacing the generator to the tested code. However, random test generation by itself has several severe drawbacks. First, blind generation of values means that it misses errors triggered by narrow ranges of inputs. A trivial example: if a function only has an error if its 32-bit integer argument is equal to "12345678" then random will most likely have to generate billions of test cases before it hits this specific case. Second, and

* This paper is a shortened version of [1], which was in simultaneous submission with similar but independent work by Patrice Godefroid et al [2]. Our thanks to Patrice for graciously accepting this version as an invited paper.

P. Godefroid (Ed.): SPIN 2005, LNCS 3639, pp. 2–23, 2005.

similarly, random testing has difficulty hitting errors that depend on several different inputs being within specific (even wide) ranges of values. Third, the ability of random testing to effectively generate random noise is also its curse. It is very poor at generating input that has structure, and as a result will miss errors that require some amount of correct structure in input before they can be hit. A clear example would be using random test generation to find bugs in a language parser. It will find cases where the parser cannot handle garbage inputs. However, because of the extreme improbability of random generation constructing inputs that look anything like legal programs it will miss almost all errors cases where the parser mishandles them.

Of course, random can be augmented with some amount of guidance to more intelligently generate inputs, though this comes at the cost of manual intervention. A typical example would be writing a tool to take a manually-written language grammar and use it to randomly generate legal and illegal programs that are fed to the tested program. Another would be having a specification or model of what a function's external behavior is and generate test cases using this model to try to hit "interesting" combinations. However, all such hybrid approaches require manual labor and, more importantly, a willingness of implementors to provide this labor at all. The reluctance of systems builders to write specifications, grammars, models of what their code does, or even assertions is well known. As a result, very few real systems have used such approaches.

This paper's first contribution is the observation that code can be used to *automatically* generate its own potentially highly complex test cases. At a high level, the basic idea is simple. Rather than running the code on manually-constructed concrete input, we instead run it on symbolic input that is initially allowed to be "anything." As the code observes this input, these observations tell us what legal values (or ranges of values) the input could be. Each time the code makes a decision based on an observation we conceptually fork the execution, adding on one branch the constraint that the input satisfies the observation, and on the other that it does not. We can then generate test cases by solving these constraints for concrete values. We call such tests *execution generated testing* (EGT).

This process is most easily seen by example. Consider the following contrived routine `bad_abs` that incorrectly implements absolute value:

```
int bad_abs(int x) {
    if(x < 0)
        return -x;
    if(x == 12345678)
        return -x;
    return x;
}
```

As mentioned before, even such a simple error will probably take billions of random-generated test cases to hit. In contrast, finding it with execution generated testing it is straightforward. Symbolic execution would proceed as follows:

1. Initial state: set `x` to the symbolic value of “anything.” In this case, before any observations at all, it can be any value between `INT_MIN` and `INT_MAX`. Thus we have the constraints $x \geq INT\_MIN \wedge x \leq INT\_MAX$.
2. Begin running the code.
3. At the first conditional (line 1) fork the execution, setting `x` to the symbolic constraint $x < 0$ on the true path, and to $x \geq 0$ on the false path.
4. At the return (line 2) solve the constraints on `x` for a concrete value (such as `x == -1`). This value is later used used as a test input to `bad_abs`.
5. At the second conditional (line 3) fork the execution, setting `x` to the constraints $x \equiv 12345678 \wedge x \geq 0$ on the true path and $x \neq 12345678 \wedge x \geq 0$ on the false path.
6. At the second return (line 4) solve the symbolic constraints $x \equiv 12345678 \wedge x \geq 0$. The value is 12345678 is our second test case.
7. Finally, at line 5, solve x's constraints for a concrete value (e.g., `x = 1`). This value is used as our third, final case.

We can then test the code on the three generated values for `x`. Of course, this sketch leaves many open questions — when to generate concrete values, how to handle system calls, how to tell what is correct, etc. The rest of the paper discusses these issues in more detail.

There are a couple of ways to look at the approach. From one point of view, implementation code has a “grammar” of the legal inputs it accepts and acts on, or rejects. EGT is an automatic method to extract this grammar (and the concrete sentences it accepts and rejects) from the implementation rather than from a hand-written specification. From another viewpoint, it can be seen as a way to turn code “inside out” so that instead of consuming inputs becomes a generator of them. Finally, and perhaps only half-vacuously, it can be viewed as a crude analogue of the Heisenberg effect in the sense that unlike observations perturbing experiments from a set of potential states into a variety of concrete ones, observations in this case perturb a set of possible inputs into a set of increasingly concrete ones. The more precise the observation the more definitively it perturbs the input. The most precise observation, an equality comparison, fixes the input to a specific concrete value. The least precise, an inequality, simply disallows a single value but leaves all others as possibilities.

This paper has three main contributions:

1. A simple conceptual approach to automatically generate test cases by running code on symbolic inputs.
2. A working prototype EGT system.
3. Experimental results showing that the approach is effective on real code.

The paper is organized as follows. Section 2 gives an overview of the method. Section 3 discusses concrete implementation issues. The next four sections give four case studies of applying the approach to systems code. Finally, Section 7 discusses related work and Section 8 concludes.

2 Overview

This section gives an overview of EGT. The next section discusses some of the implementation details.

In order to generate test cases, EGT runs the code on symbolic rather than real input. Whenever code reads from its environment (via network packets, command line options, files, etc) we want to instead return a symbolic variable that has no constraints on its actual value. As the program executes and uses or observes this value (e.g., through comparisons), we add constraints based on these observations. Then, to determine how to reach a given program path, we solve these constraints and generate input that satisfies them.

At a high-level, the EGT system has three core activities:

1. Instrumentation to track symbolic constraints. Our prototype EGT system instruments the tested code using a source-to-source transformation. This instrumentation inserts checks around every assignment, expression and branch in the tested program and calls into our runtime system. It also inserts code to fork a new process at each decision point at which the associated boolean condition could return both `true` and `false`.
2. Constraint solving. We model our constraints using formulas of quantifier-free first-order logics as represented by CVCL, a state-of-the-art decision procedure solver [5, 6]. CVCL has been used in applications ranging hardware verification to program analysis to mathematical theorem proving.
   We use CVCL in two ways. First, after every branch point we call it to determine if the current set of constraints is satisfiable. If not, we stop following the code path, otherwise we continue. CVCL is sound: if it states that no solution exists, it is correct. Second, at the end of a code path that uses symbolic input, we use CVCL to generate concrete values to use as test input.
3. Modeling. External functions that return or consume input can either be modeled so that they work with symbolic variables, or not modeled, in which case any value they take must be made concrete. In general, one can leave most things unmodeled, with the downside that testing coverage will be reduced. Models are not that hard to write. A four-line model for the Unix `recv` system call is given in Section 6. In addition, models can be used to speed up the test generation. This optimization is discussed in Section 3.2.

The mechanical act of instrumenting code is pretty easy, and there are a lot of constraint solvers to pick from and use as black boxes. Thus, the main challenge for the approach is how to run code symbolically. The next subsection talks about this in more detail.

2.1 Symbolic Execution

The basic idea behind our approach is that when we perform logical or arithmetic operations, we generate constraints for these, and when we perform control flow decisions, we fork execution and go down both paths. This section sketches how we can symbolically execute code. For ease of exposition, we initially assume that all the variables in a program are symbolic; Section 3.1 shows how we can intermix symbolic and concrete execution in order to efficiently process real code.

Assignment: `v = e` . We symbolically do an assignment of an expression `e` to a variable `v` by generating the constraint that $v \equiv e$. For example, `v = x + y` generates the constraint that $v \equiv x + y$; other arithmetic and logical operators are similar.

The complication is that `v` may have been involved in previous constraints. We must distinguish the newly assigned value of `v` from its use in any already generated constraints. For example, assume we have two assignments: (1) `x = y` and then (2) `y = 3`. The first assignment will generate the constraint that $x \equiv y$. The second will generate the constraint $y \equiv 3$. At this point, the constraints imply $x \equiv 3$, which is obviously nonsensical. This new value for `y` after its assignment `y = 3` has nothing to do with any prior constraints involving `y` and should have no impact on them. Thus, an assignment `v = e` must have two parts. First, generate a new location for `v` and only then generate the constraint that $v \equiv y$. [1]

If-statements. We symbolically execute an if-statement as follows: (1) fork execution at the conditional, (2) on the true path add the constraint that the conditional expression `e` is true ($e \equiv true$) and continue, (3) on the false path add the constraint that `e` is false ($e \equiv false$) and continue. For example:

```
 concrete |       symbolic
if(e)     | if(fork() == child)
   s1;    |     add_constraint(e == true);
          |     s1;
else      | else
   s2;    |     add_constraint(e == false);
          |     s2;
```

Loops. We transform loops into if-statements with goto's so they are handled as above. One danger is that iterating on a symbolic loop variable can continue forever, forking a new execution on each evaluation of the loop condition. The usual practical hack is to only iterate a fixed number of times or for a fixed amount of time (we do the latter). Neither solution is perfect. However, in our context almost any solution is preferable to manual test generation.

Function calls: `f(x)`. There are three differences between a symbolic function call and an imperative, call-by-value call. First, control can return multiple times into the caller, once for each fork-branching that occurs. Second, constraints placed on `x` in the body of `f` must propagate up to the caller. For example, the concrete code:

[1] Alternatively, ignoring aliasing, we could have equivalently gone through all existing constraints involving `v` and relabeled them to use a new, fresh name.

```
int foo(int x) {
    if(x == 3)
        return 1;
    else
        return 2;
}
```

will generate a symbolic execution that returns twice into the caller, since the branch will cause a forked execution. On the true branch we want to propagate the constraint that $x \equiv 3$ back to the caller and on the false that $x \neq 3$. The final difference is that at the exit point from a function, we create a temporary symbolic variable and return that as the function's expression value. Figure 1 gives a symbolic translation of `bad_abs` based on the above rules.

```
// initial constraints: x >= INT_MIN /\ x <= INT_MAX
int symbolic_bad_abs(int x) {
    ret = new symbol;   // holds the return expression.

    if(fork() == child) // fork execution at each branch point.
        add_constraint(x < 0); add_constraint(ret = −x);
        // first return, final constraints:
        //     x >= INT_MIN /\ x <= INT_MAX /\ x < 0 /\ ret = −x
        return ret;
    else
       add_constraint(x >= 0);

    if(fork() == child) // fork execution
        add_constraint(x = 12345678); add_constraint(ret = −x);
        // second return, final constraints: x >= INT_MIN /\ x <= INT_MAX
        //                    /\ x >= 0 /\ x = 12345678 /\ ret = −x
        return ret;
    else
        add_constraint(x != 12345678);

     add_constraint(ret = x);
     // last return final constraints: x >= INT_MIN /\ x <= INT_MAX
     //                          /\ x >= 0 /\ x != 12345678 /\ ret = x
     return ret;
}
```

Fig. 1. A symbolic translation of `bad_abs`

2.2 What is Correctness?

EGT, like all testing approaches, needs to have some notion of what “bad” behavior is so that it can flag it. We use three approaches to do so.

First, and unsurprisingly, check for program independent properties, such as segmentation faults, storage leaks, memory overflows, division by zero, deadlocks, uses of freed memory, etc.

Second, do cross-checking. If a piece of code implements an important interface, then there are likely to be several implementations of it. These implementations can be cross-checked against each other by running the test cases generated from one implementation (or both) on both implementations and flagging differences. One important usage model: after modifying a new version of a system, cross-check it against the old version to make sure any change was intended. This approach works especially well for complex interfaces.

Third, specification-by-example. While writing specifications to state what exactly code must do in general is hard, it is often much easier to take the specific test cases our tool generates and specify what the right answers are just for these cases. For example, for the `bad_abs` routine, the EGT system generates the three concrete values: -3, 12345677, 12345678. Thus, for testing we would just do:

```
assert(bad_abs(−3) == 3);
assert(bad_abs(12345677) == 12345677);
assert(bad_abs(12345678) == 12345678);
```

3 Implementation Issues

This section discusses implementation aspects of our EGT tool.

3.1 Mixed Symbolic and Concrete Execution

Ignoring memory and solver-limitations, we can run any code entirely symbolically until it interacts with the outside, concrete world. For example, if it calls external code, or sends a packet on a real network to a machine running concrete code, or prints output to be read by a real person. At this point you must either make the inputs to the external code concrete (e.g, you must send data rather than a symbolic constraint in a network packet), or, alternatively, make a model of the world to pull it into the simulation.

In practice, constraint solvers are not as robust as one might hope and so without care overzealous constraint generation will blow them up, sometimes for good theoretic reasons, sometimes for unimpressive practical ones. Further, symbolic-only execution is expensive in both speed and space. Thus, we do a hybrid approach that intermixes concrete and symbolic execution. The basic approach is that before every operation we dynamically check if the values are all concrete. If so, we do the operation concretely. Otherwise, if at least one value is symbolic we do the operation symbolically (using the logic described in Section 2.1.

We use the CIL tool [7] to instrument the code of tested programs. Below, we sketch how to conceptually rewrite source constructs for a C-like language so that they can run on either concrete or symbolic values, mentioning some of the more important practical details.

Our first transformation conceptually changes each variable or expression `v` to have two instances: a concrete one (denoted `v.concrete`) and a symbolic one (denoted `v.symbolic`). If `v` is concrete, `v.concrete` holds its concrete value and `v.symbolic` contains the special token ⟨*invalid*⟩. Conversely, if `v` is symbolic, `v.symbolic` holds its symbolic value and `v.concrete` is set to ⟨*invalid*⟩.

In practice, we track the `v.symbolic` field using a table lookup that takes the address of a the variable `v` (which gives it a unique name) and returns `v`'s associated "shadow" symbolic variable `v.symbolic` (if it is symbolic) or null (if it is concrete). In the latter case, the variable `v` contains the concrete value (`v.concrete`) and can just be used directly. The following examples assume explicit concrete and symbolic fields for clarity.

```
assign_rule(T &v, T e) {
  if(e is concrete)
    // equivalent to v.concrete = e.concrete;
    //               v.symbolic = <invalid>;
    v = (concrete=e.concrete, symbolic=<invalid>);
  else
    // equivalent: v.symbolic = e.symbolic
    v = (concrete=<invalid>, symbolic=new symbolic var T);
    constraint(v.symbolic = e.symbolic);
}
```

Fig. 2. Rewrite rule for assignment `v = e` for any variable `v` and expression `e` of type `T`

The most basic operation is assignment. Figure 2 gives the basic assignment rule. If the right hand variable `e` is a concrete expression or variable, just assign its concrete value to the left-hand side `v` and mark `v`'s symbolic component as `invalid`. If `e` is symbolic, then as explained in the previous section, we must allocate a fresh symbolic variable to be used in any new constraints that are generated. After that, we first set `v.concrete` to be `invalid` and then add the constraint that `v.symbolic` equals `e.symbolic`.

Roughly as simple are basic binary arithmetic operators. Figure 3 gives the rewrite rule for binary addition; other binary arithmetic operators are similar. If both `x` and `y` are concrete, we just return an expression whose concrete part is just their addition and symbolic part is invalid. Otherwise we build a symbolic constraint `s` and then return an expression that has `s` as its symbolic component and `invalid` for its concrete.

The rewrite rule for if-statements is a straight-forward combination of the purely symbolic rule for if-statements with the similar type of concrete-symbolic checking that occurs in binary relations. There are two practical issues. First, our current system will happily loop on symbolic values — the parent process of a child doing such looping will terminate it after a timeout period expires. Second, we use the Unix `fork` system call to clone the execution at every symbolic

```
// rule for x + y
T plus_rule(T x, T y) {
    if(x and y are concrete)
        return (concrete=x.concrete + y.concrete, <invalid>);

    s = new symbolic var T;
    if(x is concrete)
        constraint(s = x.concrete + y.symbolic);
    else if y is concrete
        constraint(s = x.symbolic + y.concrete);
    else
        constraint(s = x.symbolic + y.symbolic);
    return (concrete=<invalid>, symbolic=s);
}
```

Fig. 3. Rewrite rule for "`x + y`" where variables `x` and `y` are of type `T`

branch point. Naively this will quickly lead to an exponential number of processes executing. Instead we have the parent process wait for the child to finish before continuing to execute on its branch of the conditional. This means we essentially do depth-first search where there will only be one active process and a chain of its predecessors who are sleeping waiting for the active process to complete.

```
// rule for *p
T deref_rule(T* p) {
   if(*p is concrete)
       return (concrete=*p, symbolic=<invalid>);
   else
       s = new symbolic var T;
       if(p is concrete)
          constraint(s = (*p).symbolic);
       else
          // symbolic dereference of p
          constraint(s = deref(p.symbolic));
       return (concrete=<invalid>, symbolic=s);
}
```

Fig. 4. Rewrite rule for dereference "`*p`" of any pointer `p` of type `T`. The main complication occurs when we dereference a symbolic pointer: in this case we must add a symbolic constraint on the dereferenced value.

Because dereference deals with storage locations, it is one of the least intuitive rewrite rules. Figure 4 gives the rewrite rule for dereferencing `*p`. A concrete dereference works as expected. A dereference of a concrete pointer `p` that points to a symbolic value also works as expected (i.e., just like assignment, except that the rvalue is dereferenced). However, if `p` itself is symbolic, then we cannot actually dereference it to get what it points to but instead must generate a funny constraint that says that the result of doing so equals the symbolic dereference of `p`.

At an implementation level, CVCL currently does not handle symbolic dereferences so we do not either. Further, in the short term we do not really do the right thing with any pointer dereference that involves a symbolic value (such as a symbolic offset off of a concrete pointer or a symbolic index into a symbolic array). In such cases we will generate a concrete value, which may be illegal.

One happy result of this limitation is that, when combined with the the way the implementation uses a lookup table to map variables to their shadow symbolic values, it makes handling address-of trivial. For example, given the assignment `p = &v` we simply do the assignment, always, no matter if `v` is a symbolic or concrete. A lookup of `p` will return the same symbolic variable (if any) that lookup of `&v` does. Thus any constraints on it are implicitly shared by both. Alternatively, if there is no symbolic, then `p` will point directly at the concrete variable and dereference will work as we want with no help.

Function calls are rewritten similarly to the previous section.

One implementation detail is that to isolate the effects of the constraint solver we run it in its own child Unix process so that (1) we can kill it if it does not terminate and (2) any problems it runs into in terms of memory or exceptions are isolated.

3.2 Creating a Model for Speed

Not all the code in the program under testing should be given the same level of attention. For example, many of our benchmarks make intensive use of the string library, but we don't want to generate test cases that exercise the code in these string routines.

More precisely, imagine a program which uses `strcmp` to compare two of its symbolic strings. Most implementations of `strcmp` would traverse one of the strings, and would compare each character in the first string with the corresponding character in the second string and would return a value when the two characters differ or when the end of a string has been reached. Thus, the routine would return to the caller approximately 2n times, each time with a different set of constraints. However, most applications use a routine such as `strcmp` as a black box, which could return only one of the following three values: 0, when the strings are equal, -1 when the first string is lexicographically smaller than the second one, and 1 otherwise. Returning the same value multiple times does not make any difference for the caller of the black box.

Instead of instrumenting routines such as those in the string library, we could instead provide models for them. A model for `strcmp` would return three times, once for each possible return value. After each fork, the model would add a series of constraints which would make the outcome of that branch symbolically true: for example, on the branch which returns 0, the model would add constraints setting the two strings equal. Of course, certain branches may be invalid; e.g. if the two string have different lengths, `strcmp` could not return 0. In this case, the corresponding branch is simply terminated.

We implemented models for the routines in the string library, and used them in generating tests for our benchmarks. Adding these specifications has two main benefits. On the one hand, it removes useless test cases from the generated test suites (by removing tests which would only improve code coverage in the string routines), and on the other hand it significantly improves performance. For the WsMp3 benchmark that we evaluate in Section 6, the test suites are generated approximately seven times faster.

3.3 Discussion

Currently we do lazy evaluation of constraints, deferring solving them until the last possible moment. We could instead do eager evaluation, where as soon as we use a symbolic value we make up a concrete one. This eliminates the need to execute code symbolically. However, by committing to a concrete value immediately, it precludes the ability to change it later, which will often be necessary to execute both paths of any subsequent branch based on that variable's value (since the concrete value will either satisfy the true or the false branch, but not both). A hybrid approach might be best, where we make up concrete values immediately and then only do full symbolic execution on code paths that this misses.

4 Micro-case Study: Mutt's UTF8 Routine

As the first micro-benchmark to evaluate EGT, we applied it to a routine used by the popular Mutt email client to convert strings from the UTF-8 to the UTF-7 format. As reported by Securiteam, this routine in Mutt versions up to version 1.4 have a buffer overflow vulnerability which may allow a malicious IMAP server to execute arbitrary commands on the client machine [8].

We selected this paper in part because it has been one of the examples in a recent reliability paper [9], which used a carefully hand-crafted input to exploit it.

We extracted the UTF8 to UTF7 conversion routine from Mutt version 1.4, ran the code through our tool, and generated test cases for different lengths of the UTF-8 input string. Running these generated tests immediately found the error.

The paper we took the code from suggested a fix of increasing the memory allocation ratio from `n*2` to `n*7/3`. We applied this change to the code, and reran the EGT generated test cases, which immediately flagged that the code still has an overflow. The fact that the adjusted ratio was still incorrect highlights the need for (and lack of) automated, comprehensive testing.

Table 1 presents our results. For each input size, we report the size of the generated test suite and the time it took to generate it, the cumulative statement coverage achieved up to and including that test suite, and the largest output size that we generated for that input size. These results (and all our later results), were generated on a Intel Pentium 4 Mobile CPU at 1.60GHz, with 512MB RAM.

Table 1. Test suites generated for utf8_to_utf7

Input Size	Generation Time	Test Suite Size	Statement Coverage	Largest Output
1	16s	10	84.0%	5
2	1m35s	38	94.2%	8
3	7m26s	132	94.2%	11
4	34m12s	458	95.6%	15
5	2h35m	1569	95.6%	19

5 Case Study: Printf

This section applies EGT to three different `printf` implementations. The `printf` routine is a good example of real systems code: a highly complex, tricky interface that necessitates an implementation with thickets of corner cases. Its main source of complexity is the output format string it takes as its first argument. The semantics of this single string absorb the bulk of the 234 lines the ANSI C99 standard devotes to defining `printf`; these semantics define an exceptionally ugly and startling programming language (which even manages to include iteration!).

Thus, `printf` is a best-case scenario for EGT. The standard and code complexity create many opportunities for bugs. Yet the inputs to test this complexity can be readily derived from `printf`'s parsing code, which devolves to fairly simple, easily solved equality checks. Further, the importance of `printf` means there are many different implementations, which we can use to finesse the need for a specification by cross-checking against each other.

We checked the following three `printf` implementations; all of them (intentionally) implemented only a subset of the ANSI C99 standard:

1. The Pintos instructional operating systems `printf`; the implementation intentionally elides floating point. This implementation is a stern test of EGT, since the developer (the co-author of a widely-read C book) had intimate knowledge of the standard.
2. The `gccfast printf`, which implements a version of printf in terms of `fprintf`. [2]
3. A reduced-functionality `printf` implementation for embedded devices. [3]

We used EGT to generate test suites by making the format string the single symbolic argument to `printf`. We set the size of this symbolic string to a fixed

[2] `http://www.opensource.apple.com/darwinsource/WWDC2004/gccfast-1614/`

[3] `http://www.menie.org/georges/embedded/index.html`

Table 2. Test suites generated for `printf`, the first row of each size gives the number of generated tests, the second row the time required to do so

Format Length	Pintos´ printf	Embedded printf	GCCfast printf
2	34 21s	17 2s	30 15s
3	356 4m0s	75 1m48s	273 3m10s
4	3234 40m47s	337 21m6s	2105 87m36s
128	590 123m56s	72 119m38s	908 120m19s

Table 3. Mismatches found in the printf implementations

	Pintos´ printf	Embedded printf	GCCfast printf
Mismatches self tests	426 of 4214	146 of 501	7 of 3316
Mismatches all tests	624 of 8031	6395 of 8031	91 of 8031
Statement Coverage	95% (172 lines)	95% (101 lines)	98% (63 lines)

length and generated test cases from the resultant constraints. We describe our measurements below and then discuss the bugs and differences found.

Measurements. We generated test cases for format strings of length 2, 3, 4, and 128. Table 2 shows the test suite size that we generated for each format length and the time it took to generate the test suite. We allowed a maximum of 30 seconds per CVCL query; there were only two queries killed after spending more than 30 seconds. For format lengths of 128 long, we terminated the test generation after approximately two hours.

Below are a representative fraction of EGT-generated format strings of length 4:

```
" %lle" " %#0f" " %G%." " % +l" " %#he" " %00." " %+jf"
" %-lf" " %#hf" " %+f " %#.E" " %00 " " %.c " " %
" % #c" " %-#." " %c%'" " %c%j" " %# p" " %---" " %+-u"
" %llc" " %0g " " %#+-" " %0 u" " %9s%"
```

Note that while almost all look fairly bizarre, because they are synthesized from actual comparisons in the code, many are legal (and at some level "expected" by the code).

Results. After generating test suites, we checked the output for each `printf` in two ways. First, we took the tests each implementation generated and cross-checked its output on these tests against the output of `glibc`'s `printf`. Each of of the three implementations attempts to implement a subset of the ANSI C99 standard, while `glibc` intends to fully implement it. Thus, any difference is a potential bug. EGT discovered lots of such differences automatically: 426 in Pintos, 146 in the Embedded `printf` and 7 in `GCCfast`'s `printf` (which was surprising since it only does minimal parsing and then just calls `fprintf`, which then calls `glibc`'s `printf`). Since we had access to the implementor of Pintos we focused on these; we discuss these below.

Second, we took the tests generated by all implementations and cross-checked their output against each other. Since they intentionally implement different subsets of the standard, we expect them to have different behavior. This experiment tests whether EGT can find such differences automatically. It can: 624 in Pintos, 6395 in `Embedded` and 91 in `GCCfast`.

Note that in both experiments, the Pintos and the GCCfast `printf` routines print an error message and abort when they receive a format string that they cannot handle. Since they only intend to handle a subset of the standard, this is correct behavior, and we do not report a mismatch in this case. In contrast, the `Embedded printf` instead fails silently when it receives a format string which it cannot handle. This means that we cannot differentiate between an incorrect output of a handled case and an unhandled case, and thus we report all these cases as mismatches.

Table 3 also shows the statement coverage achieved by these test suites; all printf's achieve more than 95% coverage. Most of the lines that were not covered are unreachable. For example, Pintos' `printf` has a `NOT_REACHED` statement which should never be reached as long as Pintos treats all possible format strings. Similarly, for the Embedded `printf`, we don't reach the lines which redirect the output to a string buffer instead of `stdout`; these lines are used by `sprintf`, and never by `printf`. Some lines however where not reached because our system treats only the format string as symbolic, while the rest of the arguments are concrete. Finally, two of the three `printf` implementations use non-standard implementations for determining whether a character is a digit, which our system does currently not handle correctly. The number of lines reported in Table 3 are real lines of code, that is lines which have at least one instruction.

We reported all mismatches from Pintos to its developer, Ben Pfaff. We got confirmation and fixes of the following bugs.

Incorrect grouping of integers into groups of thousands.

> "Dammit. I thought I fixed that... Its quite obviously incorrect in that case." — Ben Pfaff, unsolicited exclamation, 3/23/05, 3:11pm.

The code mishandled the "'" specifier that says to comma-separate integer digits into groups of three. The exact test case was:

```
// correct: −155,209,728
// pintos : −15,5209,728
printf("%'d", −155209728);
```

Amusingly enough, the bug had been fixed in the developer's tree, but he had forgotten to push this out to the released version (which we were testing).

Incorrect handling of the space and plus flags.

> "That case is so obscure I never would have thought of it." — Ben Pfaff, unsolicited exclamation, 3/23/05, 3:09pm.

The character "`%`" can be followed by a space flag, which means that "a blank should be left before a positive number (or empty string) produced by a signed conversion" (`man printf(3)`). Pinto incorrectly leaves a blank before an unsigned conversion too. We found a similar bug for the plus flag.

This bug and the previous error both occurred in the same routine, `format_integer`, which deals with formating integers. The complexity of the specification of even this one small helper function is representative of the minutia-laden constraints placed on many systems interfaces and their internals.

We now give a more cursory description of the remaining errors.

Incorrect alignment of strings. Pintos incorrectly handles width fields with strings, although this feature works correctly for integers (which got better testing).

Incorrect handling of the t and z flags. When the flag t is used, the unsigned type corresponding to `ptrdiff_t` should be used. This is a detail of the standard which was overseen by the developer. We found a similar bug for the z flag, which specifies that the signed type corresponding to `size_t` should be used.

No support for wide strings and chars. Pintos does not support wide string and wide chars, but fails silently in this case with no error message.

Undefined behavior. We found several bugs which are caused by underspecified features. An example of such a case is "`printf(''%hi'', v)`, whose output is undefined if `v` cannot be represented as a `short`.

6 Case Study: WsMp3

This section applies our technique to the WsMp3 web server designed for transferring MP3 files [10]. We use WsMp3 version 0.0.5 which, uninstrumented contains about 2,000 lines of C code; instrumented about 40,000. This version contains a security vulnerability that allows attackers to execute arbitrary commands on the host machine [11, 12]. Our technique automatically generated test cases that found this security hole. In addition, it found three other memory overflows and an infinite loop caused by bad network input (which could be used for a DoS attack).

We first discuss how we set up test generation, coverage results, and then the most direct method of effectiveness: bugs found.

6.1 Setting Up WsMp3

WsMp3 has the typical web server core: a main loop that listens for connections using `accept`, reads packet from the connection using `recv`, and then does operations based on the packet value. It also has a reasonably rich interaction with the operating system. As a first cut we only made the network packet's returned by `recv` be symbolic, but made the packet size be concrete. We did so by replacing calls to `recv` with calls to a model of it (`recv_model`) that just "returned" a symbolic array of bytes of a specific length:

```
// [model does not generate failures; msg_len is fixed]
ssize_t recv_model(int s, char *buf, size_t len, int flags) {
  make_bytes_symbolic(buf, msg_len);
  return msg_len;
}
```

It "reads in" a message of length `msg_len` by telling the system the address range between `buf` and `buf+msg_len` should be treated as symbolic. We then generated test cases for one byte packet, two bytes, and so forth by changing `msg_len` to the desired length.

After the web server finishes processing a message, we inserted a call into the system to emit concrete values associated with the message's constraints. We then emit these into a test file and run the web server on it.

One subtlety is that after the web server processes a single message we exit it. Recall that at every conditional on a symbolic value (roughly) we fork execution. Thus, the web server will actually create many different children, one for each branch point. Thus, even processing a "single" message will generate many many test messages. In the context of this server, one message has little to do explicitly with another and thus we would not get any more test cases by doing additional ones. However, for a more stateful server, we could of course do more than one message.

Finally, it was not entirely unheard of for even the symbolic input to cause the code to crash during test generation. We handle segmentation faults by installing a handler for the `SIGSEGV` signal and, if it is invoked, generate a concrete test case for the current constraints and then exit the process.

Since WsMp3 makes intensive use of the standard string library, we used our own `string.h` library described in Section 3.2. In our tests, using this library improves performance by roughly seven-fold.

6.2 Test Generation Measurements

We used EGT testing to generate tests for packets of size 1, 2, 3, 4, 5, 12, and 128. Table 4 gives (1) the number of tests generated for each size, (2) the time it took (user time), and (3) the number of times the CVCL constraint solver failed to generate a concrete test from a set of constraints within 30 seconds.

Given our naive implementation, the test generation time was non-trivial. For packets of size 12 and 128 we stopped it after 14 hours (they were running on a laptop that we wanted to write this paper on). However, note that in some sense high test generation cost is actually not so important. First, test generation happens infrequently. The frequent case, running the generated tests, takes less than a minute. Second, test generation is automatic. The time to manually

Table 4. Test suites generated for WsMp3. We stopped test generation for size 12 and 128 after roughly 14 hours

Packet Size	Unfinished Queries	Execution Time (s)	Test Suite Size
1	0	0s	1
2	0	0s	1
3	0	57s	18
4	0	10m28s	90
5	8	16m13s	97
12	134	14h15m	1173
128	63	14h15m	165

generate tests that would get similar amounts types of path coverage would be enormous. Further, manual generation easily misses cases silently. Finally, as far as we know, there was no test suite for WsMp3. Clearly the EGT alternative is much better.

We compare coverage from EGT to random testing. We use statement coverage generated using `gcc` and `gcov`. We would have preferred a more insightful metric than line coverage, but were not able to find adequate tools. We generated random tests by modifying the `recv` routine to request messages filled with random data of a given size. For each packet size (1, 2, 3, 4, 5, 128, 256, and 512 bytes long), we generate 10, 1000, and 100,000 random tests, and then measured the cumulative statement coverage achieved by all these tests. We recorded a statement coverage of 23.4%, as opposed to 31.2% for EGT.

However, the roughly 8% more lines of code hit by EGT is almost certainly a dramatic underreporting of the number of distinct paths it hits. More importantly, these lines appear out of reach of random testing no matter how many more random tests we do. In addition, note that it takes about two hours and a half to execute all the random test cases, while it takes less than a minute to execute all the EGT test cases.

We manually examined the code to see why EGT missed the other statements. Many of the lines of code that were not hit consisted of debugging and logging code (which was disabled during testing), error reporting code (such as printing an error message and aborting when a call to malloc fails), and code for processing the command-line arguments (which wasn't all reached because we didn't treat the arguments as symbolic inputs).

However, a very large portion of the code was not reached because the request messages that we fabricate do not refer to valid files on the disk, or because we fail to capture several timing constraints. As an example from the first category, when a `GET` request is received, the web server extracts the file name from the request packet, and then it checks if the file exists by using `fopen`. If the file does not exist, WsMp3 sends a corresponding error message to the client. If the file is valid, the file name is passed through various procedures for further processing. Since we don't have any files on our server, and since almost all the files being fabricated by our system would be invalid anyway, the code which process files and file names is never invoked. The right way to solve this problem is to provide models for functions such as `fopen`, `fread`, and `stat`. However, even without these models, we find interesting errors, as the next subsection describes.

6.3 Errors Found

We have identified five errors in the code which parses the request messages received by WsMp3. All were caused by a series of incorrect assumptions that WsMp3 makes about the request being processed. We describe three illustrative bugs below.

```
// [buf holds network message]
char* get_op(char *buf) {
    char* op;
    int i;

    if((op=(char *)malloc(10))==NULL) {
        printf("Not enough memory!\n");
        exit(1);
    }
    // [note: buf is '0' terminated]
   if(buf!=NULL && strlen(buf)>=3) {
        //strncpy(op,buf,3);
        i=0;
        while(buf[i]!=' ') {
           op[i]=buf[i];
           i++;
        }
        op[i]='\0';
   }
   else op=NULL;

   return op;
}
```

Fig. 5. WsMp3 buffer overflow bug: occurs if received message (held in `buf`) has more than 10 characters before the first space

Figure 5 gives the first bug. Here WsMp3 assumes that the first part of the request message (held in `buf`) holds the type of the client request, such as `GET` or `POST`, separated from the rest of the message by a space. After a request is received, WsMp3 copies this action type in an auxiliary buffer by copying all the characters from the original request, until a space is encountered. Unfortunately, it assumes the request is legal rather than potentially malicious and allocates only ten bytes for this buffer. Thus, if it receives an invalid request which does not contain a space in the first ten characters, the buffer overflows and WsMp3 usually terminates with a segmentation fault. Amusingly, there is a (commented out) attempt to instead do some sort of copy using the safe `strncpy` routine which will only up to a pre-specified length.

This routine is involved in a second bug. As part of the checking it does do, it will return `NULL` if the input is `NULL` or if the size of the incoming message is less than three characters. However, the caller of this routine does not check for a `NULL` return and always passes the buffer to `strcmp`, causing a remote-triggered segmentation fault.

The third final bug was interesting: for certain rare request messages (where the sixth character is either a period or a slash, and is followed by zero or more periods or slashes, which are immediately followed by a zero), WsMp3 goes into an infinite loop. Our EGT system automatically generates the very unusual message required to hit this bug. The problematic code is shown below:

```
while (cp[0] == '.' || cp[0] == '/')
  for (i=1; cp[i] != '\0'; i++) {
      cp[i−1] = cp[i];
      if (cp[i+1] == '\0')
         cp[i] = '\0';
    }
```

7 Related Work

To the best of our knowledge, while there has been work related to test generation and synthesis of program inputs to reach a given program point, there is no previous approach that effectively generates comprehensive tests automatically from a real program. There certainly exists no tool that can handle systems code. We compare EGT to past test generation work and then to bug finding methods.

Static test and input generation. There has been a long stream of research that attempts to use static techniques to generate inputs that will cause execution to reach a specific program point or path.

One of the first papers to attack this problem, Boyer at al. [13], proposes the use of symbolic execution to follow a given path was in the context of a system, SELECT, intended to assist in debugging programs written in a subset of LISP. The usage model was that the programmer would *manually* mark each decision point in the path that they wanted executed and the system would incrementally attempt to satisfy each predicate. More recently, researchers have tended to use static analysis to extract constraints which then they try to solve using various methods. One example is Gotlieb et al [14], who statically extracted constraints which they tried to solve using (naturally) a constraint solver. More recently, Ball [15] statically extracted predicates (i.e., constraints) using "predicate abstraction" [16] and then used a model checker to try to solve these predicates for concrete values. There are many other similar static efforts. In general, static techniques are vastly weaker than dynamic at gathering the type of information needed to generate real test cases. They can deal with limited amounts of fairly straightforward code that does not interact much (or at all) with the heap or complex expressions, but run into intractable problems fairly promptly.

Dynamic techniques test and input generation. Much of the test generation work relies on the use of a non-trivial manually-written specification of some kind. This specification is used to guide the generation of testing values ignoring the details of a given implementation. One of the most interesting examples of such an approach is Korat [17], which takes a specification of a data-structure (such as a linked list or binary tree) and exhaustively generates all non-isomorphic data structures up to a given size, with the intention of testing a program using them. They use several optimizations to prune data structure possibilities, such as ignoring any data structure field not read by a program. EGT differs from this work by attempting to avoid any manual specification and targeting a much broader class of tested code.

Past automatic input generation techniques appear to focus primarily on generating an input that will reach a given path, typically motivated by the (somewhat contrived) problem of answering programmer queries as to whether control can reach a statement or not. Ferguson and Korel[18] iteratively generate tests cases with the goal of hitting a specified statement. They start with an initial random guess, and then iteratively refine the guess to discover a path likely to hit the desired statement. Gupta et al. [19] use a combination of static analysis and generated test cases to hit a specified path. They define a loss function consisting of "predicate residuals" which roughly measures by "how much" the branch conditions for that path were not satisfied. By generating a series of test cases, they use a numerical solver to find test case values that can trigger the given path. Gupta's technique combines some symbolic reasoning with dynamic execution, mitigating some of the problems inherit in either approach but not in both. Unfortunately, the scalability of the technique has more recently been called into question, where small systems can require the method to take an unbounded amount of time to generate a test case [20].

In EGT differs from this work by focusing on the problem of comprehensively generating tests on all paths controlled by input. This prior work appears to be much more limited in this regard.

Software Model Checking. Model checkers have been previously used to find errors in both the design and the implementation of software systems [21–26, 22]. These approaches tend to require significant manual effort to build testing harnesses. However, to some degree the approaches are complementary: the tests our approach generates could be used to drive the model checked code.

Generic bug finding. There has been much recent work on bug finding [27, 26, 28, 29]. Roughly speaking because dynamic checking runs code, it is limited to just executed paths, but can more effectively check deeper properties implied by code. For example that the code will infinite loop on bad inputs, that a formatting command is not obeyed correctly. Many of the errors in this paper would be difficult to get statically. However, we view static analysis as complementary to EGT testing — it is lightweight enough that there is no reason not to apply it and then use EGT.

8 Conclusion

This paper has proposed a simple method of automatically generating test cases by executing code on symbolic inputs called execution generated testing. We build a prototype EGT system and applied it to real code. We found numerous corner-case errors ranging from simple memory overflows and infinite loops to subtle issues in the interpretation of language standards.

These results, and our experience dealing with and building systems suggests that EGT will work well on systems code, with its often complex requirements and tangled logic.

Acknowledgements

The authors thank Ted Kremenek for his help with writing and related work and David Dill for writing comments. The authors especially thank Ben Pfaff for his extensive help with the code and results in Section 5. This research was supported by NSF ITR grant CCR-0326227, NSF CAREER award CNS-0238570-001, and a Junglee Corporation Stanford Graduate Fellowship.

References

1. Cadar, C., Engler, D.: Execution generated test cases: How to make systems code crash itself. Technical Report CSTR 2005-04 3, Stanford University (2005)
2. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), Chicago, IL USA, ACM Press (2005)
3. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Communications of the Association for Computing Machinery **33** (1990) 32–44
4. Miller, B., Koski, D., Lee, C.P., Maganty, V., Murthy, R., Natarajan, A., Steidl, J.: Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin - Madison (1995)
5. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating valid ity checker. In Alur, R., Peled, D.A., eds.: CAV. Lecture Notes in Computer Science, Springer (2004)
6. Ganesh, V., Berezin, S., Dill, D.L.: A decision procedure for fixed-width bit-vectors. Unpublished Manuscript (2005)
7. Necula, G.C., McPeak, S., Rahul, S., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: International Conference on Compiler Construction. (2002)
8. Securiteam: Mutt exploit. `http://www.securiteam.com/unixfocus/5FP0T0U9FU.html` (2003)
9. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., William S. Beebee, J.: Enhancing server availability and security through failure-oblivious computing. In: Symposium on Operating Systems Design and Implementation. (2004)

10. : Wsmp3 webpage. `http://wsmp3.sourceforge.net/` (2005)
11. Associates, C.: Wsmp3 exploit. `http://www3.ca.com/securityadvisor/vulninfo/Vuln.aspx?ID=15609` (2003)
12. Secunia: Wsmp3 exploit. `http://secunia.com/product/801/` (2003)
13. Boyer, R.S., Elspas, B., Levitt, K.N.: Select – a formal system for testing and debugging programs by symbolic execution. ACM SIGPLAN Notices **10** (1975) 234–45
14. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, ACM Press (1998) 53–62
15. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO'2004: Symp. on Formal Methods for Components and Objects, Springer-Press (2004)
16. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, ACM Press (2001) 203–213
17. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). (2002) 123–133
18. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Trans. Softw. Eng. Methodol. **5** (1996) 63–86
19. Gupta, N., Mathur, A.P., Soffa, M.L.: Automated test data generation using an iterative relaxation method. In: SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (1998) 231–244
20. Edvardsson, J., Kamkar, M.: Analysis of the constraint solver in una based test data generation. In: ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (2001) 237–245
21. Holzmann, G.J.: The model checker SPIN. Software Engineering **23** (1997) 279–295
22. Godefroid, P.: Model Checking for Programming Languages using VeriSoft. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages. (1997)
23. Holzmann, G.J.: From code to models. In: Proc. 2nd Int. Conf. on Applications of Concurrency to System Design, Newcastle upon Tyne, U.K. (2001) 3–10
24. Brat, G., Havelund, K., Park, S., Visser, W.: Model checking programs. In: IEEE International Conference on Automated Software Engineering (ASE). (2000)
25. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: Extracting finite-state models from java source code. In: ICSE 2000. (2000)
26. Ball, T., Rajamani, S.: Automatically validating temporal safety properties of interfaces. In: SPIN 2001 Workshop on Model Checking of Software. (2001)
27. Das, M., Lerner, S., Seigle, M.: Path-sensitive program verification in polynomial time. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany (2002)
28. Coverity: SWAT: the Coverity software analysis toolset. `http://coverity.com` (2005)
29. Bush, W., Pincus, J., Sielaff, D.: A static analyzer for finding dynamic programming errors. Software: Practice and Experience **30** (2000) 775–802