# Oasis: Concolic Execution Driven by Test Suites and Code Modifications

*Olivier Crameri, Rekha Bachwani, Tim Brecht, Ricardo Bianchini, Dejan Kostic, Willy Zwaenepoel*

*EPFL Technical report #LABOS-REPORT-2009-002*

## Abstract

Testing remains an important aspect of checking software correctness. Manually constructed test suites are one option: they typically complete quickly, but they require human involvement in producing test cases, and their coverage may be limited. Recently, symbolic execution and concolic execution have been investigated as alternatives to test suites. These approaches require little manual intervention, and coverage can in theory be complete. However, their running times may be prohibitive for programs with complex control flow and large inputs.

The system we present in this report, called Oasis, is a research prototype that attempts to combine the advantages of test suites (speed) and concolic execution (coverage). Oasis leverages any valid inputs to the program, from test suites or past execution logs, to quickly explore the paths covered by these inputs and reach deep program paths. It then uses concolic execution to automatically explore alternative paths. This exploration starts with those paths that derive directly from the executions with valid inputs. When used for regression testing, Oasis prioritizes the exploration of paths and constraints resulting from new or modified code.

We study our techniques using two real applications, the wget Web client and the uServer Web server. Our experiments demonstrate that Oasis can quickly reach "deep" program paths (in both old and new code), and that it effectively tests the new code more extensively during regression testing. Using bug injection, we demonstrate that Oasis is able to uncover bugs that regular symbolic or concolic engines may not be able to reach within a given time budget.

## 1 Introduction

Deriving a correct software implementation is a difficult and arduous task. Programmers are often under time constraints to meet release deadlines, and therefore use a variety of static and dynamic analysis techniques to improve software reliability.

Manual code inspection by experienced programmers often yields good results, but is by no means exhaustive. Static analysis [1, 2] can automatically check many useful properties, e.g., whether the API is being used properly. Unfortunately, static analysis can be incomplete and miss many important problems, or it can overwhelm the programmer with a large number of spurious warnings.

A larger class of programming errors can be identified using dynamic analysis or testing [22]. Any mature program has a test suite that has been put together over time. A key requirement for testing is adequate path coverage [6]. Ideally, testing should cover each path of a program at least once. This seemingly simple task is complicated by the exponential explosion in the number of paths, resulting from branches in the source code. Many tools can maximize the path coverage achieved by a set of inputs [29]. Unfortunately, achieving full coverage is difficult, even for mature code with several decades of development [6]. Furthermore, manual test generation is a labor-intensive task.

An approach for comprehensive, automatic test generation that has gained considerable attention recently is symbolic execution [6, 7, 15]. The goal of symbolic execution is to systematically explore all possible paths in a program, looking for conditions that may lead the program to crash (e.g., dereferenced null pointers, divisions by zero, off-by-one array accesses, and failed `asserts`). In more detail, a symbolic execution engine marks all program input as symbolic (i.e., having arbitrary value), and then

runs the code while propagating the symbolic inputs to program variables. When it encounters a branch, the engine queries a constraint solver to compute the conditions that can lead to the two sides of the branch, updates the cumulative constraints for each side, and takes both sides in parallel. The engine proceeds to systematically cover all branches and explore all paths, if sufficient time is available. As it traverses the entire program, the engine also creates and explores constraints that check for potential crash situations.

A variation of symbolic execution, called concolic execution [5], has the same full-coverage goal but explores paths in a different manner. In concolic execution, the program is first executed with a random set of inputs, e.g., all zeros. Every time the program executes a branch, the concolic execution engine records the constraints on the input that led to the taken side of the branch. It also records constraints that check for possible crashes. After execution with this input is completed, the engine has a list of constraints. It then negates, one at a time, each of these constraints, and invokes the constraint solver to find a set of inputs that satisfies the constraints. The engine then re-executes the program, using as inputs the values selected by the solver. As before, it records the constraints it finds. The process repeats until all paths are explored or the available time expires.

Although promising, these approaches have several problems. Most obviously, the number of paths to explore is usually extremely large. Reaching certain parts of the code may require exploration of a very large number of paths with a significant amount input. Second, the running time of the solver can be considerable. A cache of existing constraint solutions is commonly used, but misses in the cache can be expensive. Consequently, these approaches may fail to explore important code paths within the time available for testing.

In this report, we introduce Oasis, a research prototype for studying the scalability issues of symbolic execution. Oasis combines the speed of test suites with the high coverage of concolic execution. Oasis leverages any existing valid inputs (i.e., those that correctly pass input parsing and/or validation), from a test suite or previous execution logs, in two ways. First, it quickly explores the paths covered by these inputs, and records all the constraints it encounters, along with the corresponding inputs, in a persistent cache. These executions achieve at least the same coverage as the test suite, while degrading testing time only slightly. Moreover, cache lookup can be orders of magnitude faster than constraint solving, so warming up the cache can accelerate the execution later on. Second, Oasis quickly explores "deep" program paths, starting from those that directly derive from the paths taken by the test cases. Other engines (symbolic or concolic) would have difficulty reaching these deep paths within the same time budget. For example, we demonstrate with a Web server that a conventional engine spends all of its time exploring the input parser part of the server with invalid inputs. In contrast, Oasis allows immediate exploration of the server code past the parser.

Along the same lines of trying to cover hard-to-reach parts of the code, Oasis also includes an optimization for regression testing (i.e., testing code upgrades). Specifically, Oasis pinpoints the differences between the old and the new versions of the program, and prioritizes the exploration of paths and constraints relating to new (or modified) code. This priority allows Oasis to more thoroughly test the new code, which is the most likely place to find bugs when regression testing. Other engines (symbolic or concolic) do not explicitly target new code and may waste precious time exhaustively exploring parts of the program that are likely to be correct. For example, our software upgrade experiments demonstrate that, in some cases, a conventional engine spends most of the available time testing old code. In contrast, Oasis spends the vast majority of the time exploring new paths and constraints. Thus, Oasis can be used to tackle the difficult problem of trying to guarantee that software works as expected after upgrades [10].

We evaluate Oasis on the wget command line Web client [30] and a Web server (uServer) [26]. To study Oasis for software upgrades, we consider two versions of these programs. To study Oasis' ability to find bugs, we also consider two versions of the programs in which we inject bugs. Our experimental results for Oasis are promising and isolate the benefits of each of its optimizations. For example, the results show that, in one hour of testing, at least 62% of the Oasis iterations (each of them an execution of the corresponding program with a different set of inputs) reach past input validation. During the same time, no executions do so with a traditional engine (or with Oasis without using valid inputs from test cases). For regression testing, Oasis spends almost 100% of its time exploring paths and constraints deriving from new code. Using valid inputs, Oasis is capable of finding 8 out of the 10 bugs we injected, whereas in the same time budget, a traditional engine would have found only 3.

Our main contribution is to demonstrate that we can significantly improve the coverage of hard-to-

```
 1 main() {
 2 /* Memory for 200 config parameter strings,
 3    each max of 80 chars long */
 4   char configVars[200][81];
 5   for (int j = 0; j < 200; j++ )
 6     strcpy (configVars[j], "");
 7
 8   int cf = open(configfile);
 9   int i = 0;
10   while (!EOF(cf) && i < 200) {
11     char* r = readline(&configVars[i], cf);
12     if (!r || !isValid (configVars[i])) {
13       exit(-1);
14     }
15     i++;
16   }
17
18 /* Buffer to read file contents */
19   char* buffer;
20   buffer = (char*)malloc(sizeof(char)*1024);
21   do_something(buffer);
22   free(buffer);
23
24 /* The printf is an invalid mem reference */
25   if (!strcmp(configVars[10], "value")) {
26     printf("\%c", buffer[someindex]);
27   }
28 }//end main
```

Listing 1: This example bug demonstrates the difficulty that concolic executions have in discovering bugs "deep" in the code and how execution with valid inputs can alleviate the problem.

reach parts of programs by prioritizing the exploration of modified code. This characteristic allow Oasis to test parts of the code that a concolic engine would explore substantially later.

The rest of this paper is organized as follows. Section 2 provides additional background and motivation. Section 3 details the design and implementation of Oasis. Section 4 discusses the use of Oasis with two sample programs, uServer and wget. Section 5 describes the related work. Finally, Section 6 concludes the paper.

## 2 Overview

We motivate the design of Oasis using the example in Listing 1. This listing is a small variation of a real bug in the *ssh−keysign* tool of OpenSSH version 3.0 [25]. The code reads a line at a time from the configuration file and validates it (lines $8-16$). When this is done, it allocates a buffer, uses it (line 21), and frees it (line 22). However, under a very specific condition, it tries to print the now freed buffer (line 26).

We now describe how a conventional concolic execution engine would handle this program. The first input to the program is the configuration file. We assume that this file has 1000 bytes, for example. Concolic execution starts execution with all inputs set to a random value, e.g., all zeros. Thus, the entire configuration file would have 1000 bytes of zeros. We then start execution of the program with this input. The function readline (line 11) reads the input until it sees a line termination character (LTC) or until it has read 80 bytes, whichever comes first. With the given input, readline would read 80 zeros, which would fail to validate, and the program would exit. At that point, the execution would have generated 80 constraints of the form $byte_i \neq LTC$, where $0 \leq i \leq 80$. The engine would then negate each of these constraints in turn, causing it to take an input sequence that does not have 80 zeros as its first 80 bytes. For instance, it could try a 1 followed by 79 zeros next. With this input, the same sequence of events would occur, and so on. It is easy to see that the concolic engine would spend an enormous amount of time, simply exploring the very first part of the program, where input is read and validated. It would do so until it guessed a set of string values that pass the validation tests, and, until it did so, it would never reach the code past the input validation, and therefore it would not be aware of the code that causes the bug. Figure 1 illustrates path exploration by a conventional concolic execution engine on our example code.

Let's now consider the situation where we have a number of valid configurations from a test suite or from previous executions. For simplicity, assume we have just one such configuration file. Oasis leverages this input as we described in the previous section. It would start executing with this configuration file as input, pass the input validation, buffer allocation and free, and arrive at the if statement on line 25. Most likely, the if condition would not hold, and Oasis would exit without accessing the buffer in the body of the if statement. In this execution, Oasis would have collected constraints on each byte of input plus a constraint stating that the if condition is false.

In the next step, Oasis would now negate one of these constraints. Let's say that our strategy is to negate them one by one, starting from the first recorded one. This would cause us to pick an input different from the one we had on the first line of the configuration file, and most likely fail input validation. The same would happen for all the constraints related to the input variables, but then we would hit on the the constraint that says that the if condition

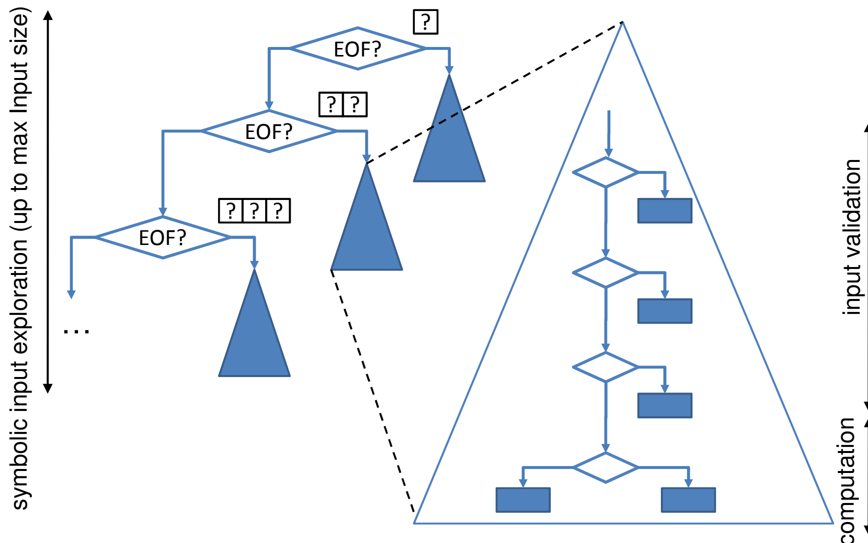


Figure 1: Path exploration by a conventional concolic execution engine on typical code. EOF in this figure refers to the check for the end of file or any other way of limiting input. Boxes with question marks refer to symbolic input that grows in size during path exploration.

was false. We would negate this one, find an input, and this one would trigger the bug.

Exploring this line of thought a little further, assume that the if statement and the printf were the two new lines of code that were added as the result of a new version. In that case we would first negate that condition and immediately find the problem. A concolic engine could not have found this bug so soon, because it would for a long time not know that the constraint even existed.

Assume alternatively (although this is not the case in the program in question here) that the if statement is replaced by an assert. In that case, we would start by negating the constraint generated by the assert (all concolic engines prioritize negating conditions that may lead to crashes, such as the condition associated with an assert), and we would also have found the bug immediately. The point is, again, that the execution with valid inputs descended deep into the program and that this provides visibility into the constraint that leads to the problem, whereas regular concolic execution with random inputs does not.

This style of program is quite common: an extensive first phase in which the input is read, parsed and validated, followed by a second phase in which the program performs its real function with inputs that have been validated. Web servers are another example. They read a large configuration file, before they start accepting incoming requests. Even in the handling of an individual request, the input string is first parsed and validated, before the requested operation is executed.

It is this observation that motivated the use of valid inputs to start Oasis execution. With these inputs we can proceed past the input validation phase and test the part of the program in which it does its real work. With regular concolic execution, the engine is liable to get bogged down for a long time in testing the validation code with invalid inputs.

We want to stress that we do not argue that Oasis is better than regular concolic execution in some absolute sense, in terms of finding more bugs, or finding them more quickly, or providing much better coverage. Rather, we argue that we are able to explore certain parts of a program much sooner than a regular engine, in particular new parts of the code that are deep in the execution.

## 3 Design and Implementation

We start by giving an overview of Oasis. We then describe its two principal innovations: the ability to start exploration from valid inputs, and to prioritize the exploration of paths introduced in a new version of the code under test. Finally, we describe our path exploration heuristic, which takes advantage of these two innovations.

## 3.1 Oasis Overview

We have derived Oasis through extensive modification of the Crest open-source concolic execution engine [5]. Oasis instruments C programs using CIL [23]. The instrumentation is used to propagate symbolic information and to flag crash-prone statement (i.e., statements such as assertions, divisions, memory reads and writes). Oasis incorporates many of the features of current state-of-the-art concolic and symbolic execution engines [6, 7, 5].

Programs being tested with Oasis always run with concrete inputs. This makes interaction with the external environment simple: programs running with Oasis can interact with outside libraries or the filesystem. However, in order to be able to track symbolic constraints on input that is passed to external libraries, Oasis needs visibility into the library functions. This is achieved by either recompiling those libraries with the instrumentation, or by providing models. Oasis provides its own instrumented version of *uclibc*, a simple *libc* implementation. In addition, Oasis provides a set of functions modeling interaction with the filesystem and the network. This allows Oasis to consider input coming from these sources as being symbolic (i.e., to log the constraints imposed on this input), and to support system calls operating on symbolic memory.

To generate inputs exploring different paths in the program, Oasis uses an off-the-shelf constraint solver, called STP [12]. Because the running time of the solver can be quite high, Oasis tries to reduce the complexity of the query to the solver by determining the minimal set of dependent constraints that need to be solved. To reduce the number of queries to the solver, Oasis caches previous solutions from the solver.

### 3.1.1 What is Symbolic?

Oasis tracks symbolic constraints on the input, therefore it needs to know what in the address space of the program is input. There are different ways of marking input as symbolic. By using the $-symbolic_argv$ parameter of the *oasis* command, the user can specify the number of symbolic command line parameters, along with their size. By inserting calls to $oasis_sym(address, size)$ within the program, the user can mark a specific address as the beginning of symbolic input of size $size$. This is very helpful to mark input coming from uninstrumented libraries. Finally, in a way similar to KLEE [6], we provide a set of system call wrappers that are able to return either symbolic memory or concrete input, depending on whether the user has decided to treat a certain file or socket as symbolic input or not.

### 3.1.2 Collecting Symbolic Constraints

Oasis instruments the program under test using CIL[23]. For each statement in the program, the instrumentation adds a few calls to a library that is used to keep track at runtime of the statements executed in the program and record the symbolic constraints. The way Oasis instruments the program is inherited from Crest[5] and extended in order to allow tracking for bugs. We therefore only describe here the main aspects of our approach and refer the reader to the literature for further details.

During execution, Oasis maintains a map of addresses to expressions, representing the address space of the program, hereafter named symbolic address space or SAS. An expression in Oasis is a tree composed of symbolic variables, constants, memory objects (arrays of expressions), and operators. Whenever input is marked as being symbolic, Oasis inserts in the SAS, at the address of the real input, one symbolic variable per byte of input. Oasis records in the SAS an expression composed of a memory object for each array[1] which is allocated (either statically or dynamically) in the program. Similarly, Oasis deletes memory objects from the SAS, when their corresponding array is freed. When the program executes a binary operation, such as an addition or a subtraction, Oasis maintains the corresponding binary expressions in the SAS.

We refer to an expression as being symbolic if, somewhere in the tree, the expression references a symbolic variable. In other words, an expression is symbolic if it involves the input of the program.

While maintaining the symbolic address space, Oasis logs the symbolic constraints corresponding to control flow in the program. For instance, when a conditional branch is executed, and the expression corresponding to the condition is symbolic, a new constraint is logged. If the program takes the $true$ side of the program, the constraint logged is the symbolic expression. If, however, the program follows the $false$ side of the branch, the constraint logged is the negation of the symbolic expression.

Before letting the program execute any crash-prone statement, such as a division or a memory read/write, Oasis checks that the statement is not going to crash the program. For divisions, it checks that the divisor is not zero, and if it is, it reports a bug. Otherwise, if the divisor is a symbolic expression, Oasis logs a constraint stating that the divisor was not zero. Similarly, for memory reads

[1]We handle C structures and buffers similarly.

or writes, Oasis checks that the address being accessed points to a memory object and is within the bounds of this object. If this is not the case, Oasis reports a bug. If it is, and the memory accessed is symbolic, Oasis logs a constraint stating that the offset is within the range of the memory object (i.e., $0 < offset < memory_object_size - 1$).

Oasis handles reads and writes to memory through symbolic pointers in the same way it handles reads and writes to arrays and structures. Oasis supports symbolic pointers to pointers. For each access, Oasis logs a constraint stating the current value of the symbolic pointer. During path exploration, it then enumerates every possible location to which the symbolic pointer to pointer can point.

#### 3.1.3 Constraint Solving

To drive the exploration of different code paths, after each execution of the program, Oasis negates one of the logged constraints and invokes STP to try to find a set of input values satisfying the symbolic constraints.

Because of the way constraints are modeled in Oasis, the translation of those constraints into STP is straightforward. Each symbolic variable is translated into an 8-bit STP bitvector, memory objects into STP arrays, and binary operations into STP operations.

During an execution, Oasis routinely collects hundreds or even thousands of constraints. To explore a different path, Oasis negates one of those constraints and seeks to find a suitable input satisfying the negated constraint and all the previous constraints. The time needed for the solver to find a solution can be considerable. Fortunately, the negated constraint most of the time only involves one or a few symbolic variables, which are themselves involved only in a few other constraints. Thus, it suffices to query the solver for a solution to the set of constraints involving the variables appearing in the negated constraint. For example, consider the following set of recorded constraints: $\{x + y = 10; y < 10; z = 3\}$. Solving the constraint $z \neq 3$ does not require solving the two first constraints. We therefore only query the solver for the constraint $z \neq 3$, and reuse previous values of $x$ and $y$. Oasis inherits from Crest its ability to compute subsets of dependent constraints before querying the solver.

#### 3.1.4 Constraint Cache

Despite trying to optimize the query to the solver, the time needed for the solver to find a solution can still be considerable.

To avoid expensive calls to the solver, state-of-the-art concolic (and symbolic) execution engines implement a cache of solutions. In Oasis, we use a simple table mapping a set of dependent constraints to the corresponding solution. For instance, reusing the example of the previous section, assume that we collect constraints $\{x+y = 10; y < 10; z = 3\}$ with solution $\{x = 4; y = 6; z = 3\}$, after executing the program. We insert two entries into the cache: one lists the constraints $\{x + y = 10; y < 10;\}$ mapping to the solution $\{x = 4; y = 6\}$, whereas the other lists the constraint $\{z = 3\}$ mapping to solution $\{z = 3\}$.

Because programs frequently re-execute the same parts of the code with different inputs, they impose the same constraints on different symbolic variables. Oasis uses this observation to increase hit rates. Specifically, before inserting a solution for a set of dependent constraints in the cache, we rename the symbolic variables in the constraints in the order in which they appear. Returning to the previous example, we rename the variables $x$ and $y$ in the constraint $\{x + y = 10\}$ to $\{v_0 + v_1 = 10\}$. If we ever encounter another constraint of the same form, but referencing different variables, we quickly find a solution in the cache.

### 3.2 Running with Previous Inputs

Oasis leverages any valid input available in order to explore deeper and longer code paths sooner than traditional concolic engines. The input can come from a test suite or from execution logs of users.

To take advantage of these valid inputs, Oasis needs to know the constraints resulting from the execution of the program with those inputs. This is done by running the program within Oasis in ”recording” mode, and feed the input normally to it. In recording mode, the models of the filesystem and the network described in Section 3.1 operate as simple pass-through, in which they read from the filesystem or the network normally and tag the input as being symbolic.

This has to be done only once for each set of inputs. Oasis saves the inputs and the constraints to disk, and reuses them whenever the program needs to be tested again. Of course, if a new version of the program is to be tested, then the new version needs to be executed with the old inputs, and new constraints collected. Oasis does that automatically.

### 3.3 Prioritizing New Code

When doing regression testing, Oasis takes advantage of the code differences between the old and the

new version of the program. It uses this information to prioritize the exploration of paths and constraints that are affected by new code.

In the current version of Oasis, we use a very simple code differencing scheme: we simply compute a (textual) diff between the old version and the new version. More sophisticated approaches could be used, e.g., relying on information from a program development environment. These techniques could do a better job of identifying the precise differences, but the principle of how to use the information in Oasis remains the same.

During an Oasis execution, when a statement creates or manipulates an expression, Oasis checks if the statement is present in the diff. If so, it flags the symbolic expression as being "new". This new flag is propagated to every expression in which a new expression is referenced. The propagation continues until the expression is logged in a symbolic constraint, which is then also flagged as being new. Such new constraints will be given priority in the exploration of different paths, as explained below.

```
1  Result = (inputs, constraints)
2
3  Initialization {
4    run each test case
5    put result in CACHE and QUEUE
6  }
7
8  Main Code {
9    for each execution E in QUEUE {
10     sorted = Sort(constraints in E)
11     for each constraint C in sorted {
12       negate C
13       input = FindSolution(!C, E)
14       {
15         run the program with input
16         put result into CACHE and QUEUE
17       }
18       remove C from E
19     }
20     Remove E from Queue
21   }
22 }
23
24 FindSolution(C, E) {
25   query cache for input satisfying C
26   if not there, query solver
27   return input
28 }
29
30 Sort(constraints) {
31   sort constraints in the following order:
32   1. dangerous constraints in new code
33   2. remaining dangerous constraints
34   3. remaining constraints in new code
35   4. remaining constraints
36   within each category, start from the first
37     constraint occurring in the execution
38 }
```

Listing 2: Search heuristic used in Oasis

### 3.4 Search strategy

Listing 2 shows pseudo-code for the heuristic that Oasis implements. There are two important data structures: the CACHE that holds solutions to constraints obtained in earlier executions, and the QUEUE that holds a queue of executions (each a set of inputs) still to be considered.

During initialization, Oasis executes the program under test with each set of the valid inputs, inserting the results in CACHE and QUEUE. Next, Oasis iterates over all executions E in QUEUE (outer for loop in Listing 2). In the inner loop, it iterates over each constraint recorded during the execution of E, negates it, finds a solution for the negated constraint (either from the cache or by invoking the solver), executes the program with the new set of inputs, and finally inserts the new execution, with its inputs and constraints, in CACHE and QUEUE.

The constraints in the inner loop are examined in the following order. First, we examine constraints that involve crash-prone operations (those that may crash the program) coming from new code. Second, we consider the remaining constraints that involve crash-prone operations. Third, we examine any remaining constraints coming from new code. Finally, we consider all remaining constraints. The idea behind this ordering is to drive the search first towards new code that can potentially cause bugs, then to new code, then old code that can potentially cause bugs, and finally to other code. Within each of these four sets, we look at the constraint in the order that they were recorded in the program. Since Oasis starts execution from valid input, it already reaches parts of the code deep in the execution. Exploring constraints in the order in which they appear allows to explore those paths with more breadth.

All constraints resulting from a particular execution are examined before going to the next execution. The executions are examined in a simple FIFO manner. One could potentially explore executions using a different ordering here as well, but we have not found the need to do so.

## 4 Experimental Results

In this section, we first evaluate Oasis' ability to find bugs that we inject into real programs. This initial study demonstrates that Oasis finds bugs in

| Application | Version | Old version | LOC |
|---|---|---|---|
| wget | 1.9.1 | 1.8.2 | 23333 |
| uServer | 0.5.1 | 0.4.7 | 28550 |

Table 1: Versions and numbers of lines of C code of the applications used in our experiments.

| Config. | Uses Valid Input | Prioritize New Code |
|---|---|---|
| dfs | no | no |
| oasis-noinput | no | no |
| oasis-input | yes | no |
| oasis-input-diff | yes | yes |

Table 2: Summary of the Oasis configurations used in the experiments.

fewer iterations than other engines. We later detail the benefits of Oasis' two novel features, namely to leverage previous valid inputs to start the program exploration, and to prioritize the exploration of new or modified code. These latter studies demonstrate that Oasis achieves higher and more targeted coverage than other engines, in the same number of iterations.

## 4.1 Methodology

We evaluate Oasis for two real applications, wget and uServer. wget is a command line Web client that supports many different protocols. uServer is a Web server designed for experimentation and measurement. We consider two versions of each of the applications: wget versions 1.9.1 (called wget) and 1.8.2, and uServer versions 0.5.1 (called simply uServer) and 0.4.7. Both applications have $20,000 - 30,000$ lines of code, and thus are non-trivial. Table 1 displays the number of lines of code (LOC), measured using the `scount` utility, and the version information for each application. We also study variations of these programs, where we inject simple bugs.

Table 2 summarizes the Oasis configurations we studied. The `oasis-noinput` configuration represents Oasis running a straightforward variation of the algorithm described in Section 3.4, where we do not start from valid inputs and do not prioritize the exploration of new code. The `oasis-input` strategy is the same configuration, but this time we start from valid inputs. Finally, the `oasis-input-diff` configuration combines both Oasis optimizations, i.e., it implements the algorithm from Section 3.4 exactly.

To explore the effects of an alternative strategy, we have implemented a simple depth-first search exploration strategy, called `dfs`, in Oasis. This strategy recursively negates the last constraint generated, until a suitable input for the negated constraint cannot be found. Obviously, it neither prioritizes the exploration of new code, nor does it use valid inputs to start exploration.

## 4.2 Inputs to the Applications

### 4.2.1 wget

**Symbolic Input.** wget is run with a valid URL that it uses to query the server. It reads from the socket the reply that it gets from the server, namely a header and the file itself. Therefore, we configured Oasis to mark all the data coming on a network socket as symbolic.

**Previous Valid Input.** In the case of wget, we used one simple test case consisting of running wget to fetch a Web page from the server of a local ISP.

### 4.2.2 uServer

**Symbolic Input.** The input to uServer arrives over a network socket in the form of a Web request. Therefore, we configured Oasis to mark all the data coming on a network socket as symbolic. Command line arguments and input from the file system are kept concrete.

**Previous Valid Inputs.** Below is the form of all the requests to uServer, except that the file/URI changes each time.

```
GET file.txt HTTP/1.1\r\n
User-Agent: httperf/0.8.4\r\n
Host: 127.0.0.1\r\n
\r\n
```

We ran uServer six times, each time with slightly different command line options, requesting only one file that is found by the server. These tests are simplified versions of the regression tests that come with uServer. To ensure a bounded execution time while testing, we modified uServer's infinite "get request and reply" loop to exit after it responds to one request. We use a locally modified version of httperf [19] to send requests to the server using the same host the server is running on.

## 4.3 Results

All experiments were run on a 32-bit Xeon machine with 4GB of RAM running Linux 2.6.15 or 2.6.24.

| Config. | b1 | b2 | b3 | b4 | b5 |
|---|---|---|---|---|---|
| dfs | Y | Y | N | N | N |
| oasis-noinput | Y | Y | N | N | N |
| oasis-input | Y | Y | Y | Y | Y |
| oasis-input-diff | Y | Y | Y | Y | Y |

Table 3: Summary of the bugs found by each configuration of Oasis in wget, when using a test case with cookies. A 'Y' indicates that the bug has been found.

| Config. | b1 | b2 | b3 | b4 | b5 |
|---|---|---|---|---|---|
| dfs | Y | Y | N | N | N |
| oasis-noinput | Y | Y | N | N | N |
| oasis-input | Y | Y | Y | N | Y |
| oasis-input-diff | Y | Y | Y | N | Y |

Table 4: Summary of the bugs found by each configuration of Oasis in wget, when using a test case without cookies. A 'Y' indicates that the bug has been found.

#### 4.3.1 Finding Bugs in wget

We inserted five asserts in different places in the code of wget. The assert conditions are dependent on the input string, but otherwise relatively simple. Our goal here is to study the results of different concolic engines as a function of the location in the code where a potential problem manifests itself. Our goal is not to examine very complicated error conditions.

The locations at which the asserts were inserted are as follows:

**Bug 1** after the first read from the socket;

**Bug 2** in the HTTP status parsing;

**Bug 3** in the HTTP header parsing;

**Bug 4** in the cookie parsing parsing;

**Bug 5** after the Web page has been downloaded.

We explicitly chose to include locations in the parsing part of the code to show the strength of the conventional concolic execution engine in finding problems at these locations, but we also chose to include locations elsewhere to show the ability of Oasis to find problems past the parsing stage.

We run all four configurations of Oasis as listed in Table 2. For the configurations that use previous valid inputs (`oasis-input` and `oasis-input-diffs`), we run the experiment twice: once with a Web page that uses cookies as previous input, and once without. The results are shown in Tables 3 and 4, respectively. Each row of the tables corresponds to an Oasis configuration and each column to a particular bug. A 'Y' in each table entry indicates that the assert was found during the experiment.

When using a Web page that uses cookies as the previous input, both `oasis-input` and `oasis-input-diffs` find all 5 bugs early in the exploration. The previous input reaches all five locations where we inserted the assertions, and thus the ensuing concolic execution quickly finds the asserts.

When using a Web page that does not use cookies (Table 4), both configurations find 4 out of 5 bugs early in the exploration. They did not find Bug 4, the one in the cookie parsing code. The previous input did not come anywhere near this code, and therefore the concolic execution did not find the problem in the 3000 iterations that we allowed. A more complete test suite would surely include a test input that uses cookies, and would therefore in all likelihood lead the concolic execution to find this assert as well.

The two configurations that do not leverage previous inputs (`dfs` and `oasis-noinput`) were able to find Bugs 1 and 2 early on. The bugs are in the parsing of the status line and are therefore reachable easily. The three other bugs, however, could not be found.

The results obtained in this experiment confirm our hypothesis. Bugs located in the deeper parts of the code, or the code that is reachable only after a significant amount of input has been read, can be found much more quickly starting the exploration from valid input, especially if that input drives the execution near the code that contains the bugs.

#### 4.3.2 Finding Bugs in uServer

We inserted five asserts in different places in the code of uServer. The assert conditions are dependent on the input string, but otherwise relatively simple. Our goal here is to study the results of different concolic engines as a function of the location in the code where a potential problem manifests itself. Our goal is not to examine very complicated error conditions.

The locations at which the asserts were inserted are as follows:

**Bug 1** after the HTTP method parsing;

**Bug 2** before opening the requested file;

**Bug 3** in the HTTP column header parsing;

| Config. | b1 | b2 | b3 | b4 | b5 |
|---|---|---|---|---|---|
| dfs | Y | N | N | N | N |
| oasis-noinput | N | N | N | N | N |
| oasis-input | Y | Y | Y | N | N |
| oasis-input-diff | Y | Y | Y | N | N |

Table 5: Summary of the bugs found by each configuration of Oasis in the uServer. A 'Y' indicates that the bug has been found.

**Bug 4** in the HTTP content length header parsing;

**Bug 5** in the cookie parsing.

We explicitly chose to include locations in the parsing part of the code to show the strength of the conventional concolic execution engine in finding problems at these locations, but we also chose to include locations elsewhere to show the ability of Oasis to find problems past the parsing stage.

We run all four configurations of Oasis as listed in Table 2. For the configurations that use previous valid input (`oasis-input` and `oasis-input-diffs`), we used the same input as described in Section 4.2.2. The results are shown in Table 5. Each row of the tables corresponds to an Oasis configuration and each column to a particular bug. A 'Y' in the table entry indicates whether that configuration found that bug.

The Oasis configurations that leverage valid input found bugs 1 to 3. The `dfs` configuration found Bug 1, but after many more iterations than the above two configurations. The `oasis-noinput` configuration found none of the bugs. Bug 4 and 5 were not found by any of the configurations in the allotted number of iterations.

`dfs` was able to find Bug 1 somewhat accidentally, as a result of the way the uServer parser is written. Reaching this bug requires passing the HTTP method parsing. The parser treats all of the characters up to a white space as the method, and then checks if it is valid. If the method is not one handled by the server and the rest of the request is valid, it returns a "501 : Not Implemented" message. As a result, even with a symbolic input that is not a valid HTTP method, we generate an execution sufficiently "near" the bug.

In contrast with the `dfs` configuration, the other configuration which does not start from valid input, `oasis-noinput`, did not find any bug. We attribute this to the fact that this search heuristic explores paths in a breadth-first manner, and is therefore not very effective at going deeper in the code when not starting from valid input.

Two out of the five bugs we inserted were not found by any of the Oasis configurations. Those were the bugs in the parsing of the 'Content-Length' header and the cookies. Those parts of the code where not exercised by any of the valid inputs we used, and therefore even the Oasis configurations driven by previous inputs did not find the problems. A more extensive test suite with valid input requests that exercised this code would probably have brought us near this code, and may have resulted in finding the bug sooner.

#### 4.3.3 Impact of Previous Valid Inputs on Coverage

We now detail the Oasis behavior when it can leverage previous valid inputs to drive testing deep into the code. In particular, our goal is to quickly pass input validation, reaching more difficult-to-reach parts of the code. For each experiment, we report the exact number of paths that were explored. Recall that each path, also called an iteration, represents an entire execution of the application with the inputs selected by the solver. In all cases, the experiments were run for at least an hour.

Table 6 presents the coverage results for wget after 1000 iterations. The second column lists the number of paths for which we ran the systems. This number is kept constant for all configurations. The third column lists the configurations' coverage, in terms of the number of unique basic blocks they explored. The fourth column reports the number of paths that represent valid inputs to wget; other paths produced malformed HTTP headers. Finally, the last column lists the number of valid inputs as a percentage of the total number of paths explored.

The results show that the percentage of valid paths in the Oasis configurations that use valid inputs is at least 86%. In contrast, the configurations that use no previous valid inputs did not pass input validation. Overall, the former configurations improve code coverage by at least 51% with respect to the latter ones. These results confirm that Oasis can reach deep parts of the code early on in its exploration.

Table 7 reports the uServer results for each Oasis configuration after 2500 iterations, in the same format as Table 6. In particular, the fourth column reports the number of paths that represent valid inputs to uServer; other paths produced malformed HTTP requests.

These results show that configurations that do not leverage valid inputs were again unable to pass input validation. More importantly, we can see that,

| Config. | #paths | #unique basic blocks | #valid input | %valid input |
|---|---|---|---|---|
| dfs | 1000 | 779 | 0 | 0% |
| oasis-noinput | 1000 | 779 | 0 | 0% |
| oasis-input | 1000 | 1281 | 862 | 86% |
| oasis-input-diff | 1000 | 1180 | 952 | 95% |

Table 6: Basic-block coverage, number of paths that pass input validation, and fraction of paths that pass validation in the wget case. When using previous valid inputs, at least 86% of the Oasis paths pass validation.

| Config. | #paths | #unique basic blocks | #valid input | %valid input |
|---|---|---|---|---|
| dfs | 2500 | 1000 | 0 | 0% |
| oasis-noinput | 2500 | 1050 | 0 | 0% |
| oasis-input | 2500 | 1157 | 1559 | 62% |
| oasis-input-diff | 2500 | 1156 | 1850 | 74% |

Table 7: Basic-block coverage, number of paths that pass input validation, and fraction of paths that pass validation in the uServer case. When using previous valid inputs, at least 62% of the Oasis paths pass validation.

when valid HTTP requests are used, at least 62% of the Oasis iterations reach parts of the code that other engines cannot reach within a limited time budget. Interestingly, `oasis-input` achieves only slightly higher code coverage than `oasis-noinput` (10%) and `dfs` (16%) for uServer. This low improvement can be attributed to the fact that most of the test cases used to drive the uServer were static HTTP GET queries. This resulted in the same code, the code processing static queries, being exercised more often. Furthermore, in the case of an HTTP GET, the amount of work that has to be done after the query has been parsed is minimal: the Web server simply opens the requested file and writes it back to the socket. Therefore, the strategies that do not use valid input and, thus extensively explore the input parser code, achieve good coverage too.

#### 4.3.4 Impact of Prioritizing Exploration of New Code on Coverage

We now evaluate the benefits of Oasis in regression testing. When regression testing, Oasis prioritizes constraints resulting from new or modified application code. It uses differences between two versions of the application to drive testing into the parts of the code that have been either modified or newly added. We expect this priorization to enable Oasis to quickly test new code more thoroughly than existing concolic (and symbolic) execution engines.

Table 8 presents the wget results when version 1.8.2 is considered the "old" code (Table 1). The second column reports the number of paths the configurations explored. We again keep this number constant. The third column lists how often the configurations decide to explore a path by negating a constraint that relates to new or modified code. This number represents the eagerness of the engine to test new code.

| Config. | #paths | #new constraints |
|---|---|---|
| dfs | 1000 | 0 |
| oasis-noinput | 1000 | 0 |
| oasis-input | 1000 | 239 |
| oasis-input-diff | 1000 | 999 |

Table 8: Number of paths visited by negating constraints related to new code in wget. All but one path explored by Oasis involve new code.

| Config. | #paths | #new constraints |
|---|---|---|
| dfs | 2500 | 24 |
| oasis-noinput | 2500 | 1386 |
| oasis-input | 2500 | 2022 |
| oasis-input-diff | 2500 | 2494 |

Table 9: Number of paths visited by negating constraints related to new code in uServer. Almost all paths explored by Oasis involve new code.

Interestingly, Oasis explores all but one path resulting from the new or modified code. This illustrates how successful Oasis is at prioritizing and extensively exploring new code. The `dfs` and `oasis-noinput` systems did not explore any new

code; they got stuck exploring paths in the code parsing the HTTP header, which was not part of the new code. Although the `oasis-input` configuration does not prioritize new code, it was able to explore some of of the new code, as it quickly passed input validation.

Table 9 describes the uServer results in the same format as Table 8. We assume that version 0.4.7 is the "old" code (Table 1).

These results again show that almost all paths explored by full-blown Oasis (i.e., `oasis-input-diff`) involve new code. The other Oasis configurations explore a smaller but still non-trivial number of paths. The reason is that the two versions of uServer are quite different, including differences in the input validation code, so its is fairly easy to "bump into" some new code. However, note that `dfs` explores only a trivial number of new paths, due to its depth-first approach to constraint exploration.

## 5 Related Works

**Static analysis and model checking.** Static analysis [1, 2] can automatically check a number of useful properties in the code, but it can miss many important problems. Moreover, it may generate many unnecessary warnings.

Model checking of implementations [21, 20, 32] can be used to find bugs in a systematic fashion. However, symbolic execution overcomes the need for creating special testing harnesses that are needed for model checking.

Oasis shares the idea of running state-space exploration from relevant state with CrystalBall [31]. CrystalBall runs a model checker from a current neighborhood snapshot at every node in the distributed system to prevent possible inconsistencies. In contrast, Oasis leverages all available inputs, and uses symbolic execution to drive state space exploration.

**Testing and dynamic analysis.** Testing plays an important role in ensuring the overall quality of the software, because it aims to detect errors in program logic, check boundary values, and provide high code coverage. However, it is very hard to achieve good coverage in practice, as it requires careful choice of inputs. Oasis takes advantage of test cases to start path exploration, and automatically tries to cover all remaining paths by using concolic execution.

Dynamic analysis tools (such as Valgrind [24]) can provide valuable help to detect bugs during testing. Unfortunately, they detect bugs only on the paths on which they are executed. PathExpander [17] improves path coverage of such tools by selectively executing non-taken paths in a sandbox.

In contrast, Oasis uses dynamic analysis to detect bugs, and concolic execution to improve coverage of the test cases.

**Symbolic and concolic execution.** There has been a large body of work on using symbolic and concolic execution for automated test generation [3, 4, 7, 6, 9, 8, 11, 13, 15, 14, 18, 27, 28, 5].

Most of the work in this area tries to address the two main problems of symbolic execution: dealing with the environment [7, 6] and path explosion [3, 7, 6, 11, 13, 18]. Most recently, KLEE [6] uses environment modeling, and aggressive constraint caching and search heuristics to deal with these two problems, respectively. As a result, KLEE has been successful in identifying bugs in heavily debugged code. In contrast with these works, Oasis effectively leverages available inputs to start exploring parts of the code that are deep in the execution.

Other symbolic and concolic execution engines [6, 5] implement path exploration heuristics that try to drive the program into parts of the code that have not been explored yet. They typically use a statically computed control flow graph and use it to try to follow branches that are more likely to drive the application to yet unexplored parts of the code. Oasis does not implement any such heuristic. Instead, it relies on previous test cases to reach parts of the code deeper into the execution. Although we have not experimented with such heuristics, the fact that Oasis starts execution from real input is a complementary benefit, and therefore Oasis should be able to progress faster using such a heuristic in the same way other symbolic/concolic engines do.

Existing approaches to concolic execution, e.g., DART [15], can use random inputs to drive code execution along any given path. However, random inputs have low probability of passing through a series of input validation checks that typically exist in software. In contrast, Oasis uses any valid inputs to pass through these checks and quickly start exploring deep code paths.

Symbolic Java PathFinder [27] uses system-level concrete execution to reach a unit of code, and then uses unit-level symbolic execution. In contrast with this work, Oasis does not require manual validation of potentially erroneous inputs (that are the result of isolated unit testing).

Using existing input to progress further in the execution has been used before [16]. Oasis is a research prototype that we use to develop the potential of this technique. In addition, it leverages the differences

in the code to more quickly explore paths leading to the new or changed code. Doing so reduces the time needed for regression testing, for example in the case of software upgrades.

# 6 Conclusion

We introduce Oasis, a state-of-the-art concolic execution engine used to study the trade-offs in improving the scalability of symbolic execution. The design of Oasis is motivated by the key observation that traditional concolic and symbolic execution engines spend a large amount of time trying to get past the initial input validation phase in many applications. This is especially problematic in regression testing, when testing should focus on the new pieces of code rather than exhaustively on the input validation phase.

Oasis improves on current state-of-the-art concolic and symbolic engines by leveraging the knowledge of the differences in the code to prioritize exploration of the new code.

Using two real applications, we demonstrated the benefits of Oasis. By starting path exploration from valid input, Oasis is able to pass the input validation phase more than 62% of the time in the first few hours of testing. This allows it to explore parts of the code deeper in the execution much sooner than conventional execution engines. By leveraging the knowledge of the differences in the code, Oasis spends almost 100% of its time exploring paths derived from new code during regression testing. Finally, we used bug injection to demonstrate Oasis' ability to find bugs that remain uncovered by a traditional concolic execution engine within the same time budget.

In the future, we plan to use Oasis to find new bugs in popular open source applications. We are especially interested in using Oasis with applications that already have a large test suite to further demonstrate Oasis' ability to explore some parts of the code sooner than traditional symbolic/concolic execution engines.

# Acknowledgments

This work is supported by a grant from the Hasler foundation (grant 2103).

# References

[1] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.

[2] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[3] P. Boonstoppel, C. Cadar, and D. R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *TACAS*, pages 351–366, 2008.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 2–16, Washington, DC, USA, 2006. IEEE Computer Society.

[5] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley, Sep 2008.

[6] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM.

[8] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing Software by Blocking Bad Input. In *SOSP*, 2007.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th SOSP*, October 2005.

[10] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. *SIGOPS Oper. Syst. Rev.*, 41(6):221–236, 2007.

[11] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162, 2007.

[12] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[13] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

[14] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215, 2008.

[15] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[16] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*, 2008.

[17] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 38–52, Washington, DC, USA, 2006. IEEE Computer Society.

[18] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, pages 416–426, 2007.

[19] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *The First Workshop on Internet Server Performance*, pages 59—67, Madison, WI, June 1998.

[20] M. Musuvathi and D. R. Engler. Model Checking Large Network Protocol Implementations. In *NSDI*, 2004.

[21] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[22] G. J. Myers. *Art of Software Testing.* John Wiley & Sons, Inc., New York, NY, USA, 1979.

[23] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compilier Construction*, 2002.

[24] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[25] Openssh, a free version of the ssh connectivity tools. http://www.openssh.com/.

[26] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton. Comparing the performance of web server architectures. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 231–243, New York, NY, USA, 2007. ACM.

[27] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA*, 2008.

[28] K. Sen, D. Marinov, and G. Agha. 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*, pages 263–272, New York, NY, USA, 2005. ACM.

[29] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. *SIGSOFT Softw. Eng. Notes*, 27(4):97–106, 2002.

[30] Wget, a command line web client. http://www.gnu.org/software/wget/.

[31] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.

[32] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.