Striking a New Balance Between Program Instrumentation and Debugging Time

Olivier Crameri
EPFL
olivier.crameri@epfl.ch

Ricardo Bianchini
Rutgers University
ricardob@cs.rutgers.edu

Willy Zwaenepoel
EPFL
willy.zwaenepoel@epfl.ch

Abstract

Although they are helpful in many cases, state-of-the-art bug reporting systems may impose excessive overhead on users, leak private information, or provide little help to the developer in locating the problem. In this paper, we explore a new approach to bug reporting that uses partial logging of branches to record the path leading to a bug. We use static and dynamic analysis (both in isolation and in tandem) to identify the branches that need to be logged. When a bug is encountered, the system uses symbolic execution along the partial branch trace to reproduce the problem and find a set of inputs that activate the bug. The partial branch log drastically reduces the number of paths that would otherwise need to be explored by the symbolic execution engine. We study the tradeoff between instrumentation overhead and debugging time using an open-source Web server, the *diff* utility, and four coreutils programs. Our results show that the instrumentation method that combines static and dynamic analysis strikes the best compromise, as it limits both the overhead of branch logging and the bug reproduction time. We conclude that our techniques represent an important step in improving bug reporting and making symbolic execution more practical for bug reproduction.

Categories and Subject Descriptors D.2.5 [*Software*]: Testing and Debugging

General Terms Design, Experimentation, Reliability

Keywords Debugging, Bug Reporting, Symbolic Execution, Static Analysis

1. Introduction

Despite considerable advances in testing and verification, programs routinely ship with a number of undiscovered bugs. Some of those bugs are later uncovered and reported by users. Debugging is an arduous task in general, and it is even harder when bugs are uncovered by users. Before the developer can start working on a fix, the problem must be reproduced. Reporting systems are meant to help with this task, but they need to strike a balance between privacy concerns, recording overhead at the user site, and time for the developer to reproduce the cause of the bug.

Reporting systems. The current commercial state of the art is represented by the Windows Error Reporting System [Glerum 2009], which automatically generates a bug report when an application crashes. The bug report includes a per-thread stack trace, the values of some registers, the name of the libraries loaded, and portions of the heap. While that information is helpful, the developer must manually find the path to the bug among an exponential number of possible paths.

Furthermore, the information contained in the report may leak private information, which is undesirable and in certain circumstances prevents the use of the tool altogether. Recently, Zamfir and Candea have shown that symbolic execution can be used to partially automate the search for the path to the bug. However, as the manual approach, symbolic execution suffers from a potentially exponential number of paths to be explored [Zamfir 2010].

An alternative is to record the inputs to the program at the user site. Inputs leading to failures can be transmitted to the developer, who uses them to replay the program to the occurrence of the bug. While avoiding the problem of having to search for the path to the bug, divulging user inputs is often considered unacceptable from a privacy viewpoint. Castro *et al.* generate a set of inputs that is different from the original input but still leads to the same bug [Castro 2008]. Their approach does not transfer the original input to the developer, but requires input logging, whole-program replay, and invocation of a constraint solver at the user site.

A more direct approach is to record the path to the bug at the user site, for instance, by instrumenting the program to record the direction of all branches taken. In its naive form, this approach is infeasible because of the CPU, storage and transmission overhead incurred, but in this paper we demon-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'11, April 10–13, 2011, Salzburg, Austria.
Copyright © 2011 ACM 978-1-4503-0634-8/11/04…$10.00

```
int main(int argc, char **argv) {                 1
  char option = read_option(input);               2
  int result = 0                                  3
  if (option == 'a')                              4
    result = fibonacci(20);                       5
  else if (option == 'b')                         6
    result = fibonacci(40);                       7
                                                  8
  printf("Result: %d\n", result).                 9
  return 0;                                       10
}                                                 11
```

Listing 1. A simple program computing the fibonacci sequence.

strate that the approach can be optimized by instrumenting only a limited number of branches.

Our approach. We base our work on the following three observations. First, a large number of branches do not depend on the program input, and therefore their outcome need not be logged, because it is known a priori. We denote branches whose outcome depends on program input as *symbolic*. Other branches are denoted as *concrete*.

Consider the example program in Listing 1. Depending on the input parameter, the program computes the fibonacci number F_n for one of two different values of n. The only input to this program is the parameter that indicates for which value to compute the fibonacci number. While this program may have many branches (especially in the $fibonacci$ function, not shown for conciseness), it is sufficient to know the outcome of the branches at line 4 and 6 to fully determine the behavior of the program. Indeed, those two branches are the only symbolic ones, and it suffices to record their outcome.

A second, related observation is that application branches are typically either always symbolic or always concrete. In other words, it is rare that a particular branch at some point in the execution depends on input and at other points does not. An example of this can also be seen in Listing 1. The branches at lines 4 and 6 always depend on input, the others never do. This property is almost always true for branches in the application, and often, albeit not always, true for branches in the libraries.

Restricting our attention to branches that do depend on the input, the third observation is that it is not strictly necessary to record the outcome of all of those. When we record all such branches, the result is a unique program path, and therefore no search is required at the developer's site. When we record a subset of those branches, their outcomes no longer define a unique path but a set of possible paths, among which the developer has to search to find the path to the bug. In other words, there is a spectrum of possibilities between (1) no recording at the user site and a search at the developer site among all possible paths, and (2) complete recording of the outcome of all branches at the user site and no search at the developer site. Various points in this spectrum constitute different tradeoffs between instrumentation overhead at the user and bug reproduction time at the developer site. (We define bug reproduction as finding a set of inputs that leads the execution to the bug, or, equivalently, finding the direction of all branches taken so that they lead the execution to the bug.) It is this tradeoff that is explored further in this paper.

In particular, we propose three approaches for deciding the set of branches to instrument. The first approach uses dynamic analysis to determine the set of branches that depend on input. This approach is constrained in its effectiveness by the limited coverage of the program that the symbolic execution engine used for dynamic analysis can achieve in a given amount of time. It tends to under-estimate the number of branches that need to be instrumented, therefore leading to reduced instrumentation cost but increased bug reproduction time. The time that the symbolic execution engine is allowed to execute gives the developer an additional tuning knob in the tradeoff: the more time invested in symbolic execution, the better the coverage can be, and therefore the more precise the analysis. The second approach is based on static analysis of the program. Data flow analysis is used to determine the set of branches that depend on the input. This approach is limited by the precision of the static analysis, and in general tends to over-estimate the number of branches to be instrumented. Thus, this approach favors increased instrumentation cost in exchange for reduced bug reproduction time. The third approach combines the above two. It uses symbolic execution for a given amount of time, and then marks the branches that have not yet been visited according to the outcome of the static analysis.

When a bug in the program occurs, the developer runs the program in a modified symbolic execution engine that takes the partial branch trace as input. At each instrumented branch, the symbolic execution engine forces the execution to follow the branch direction specified by the log. In case a symbolic branch has not been logged, the engine explores both alternatives. Symbolic execution along the incorrect alternative eventually causes a further branch to take a direction different from the one recorded in the log. The symbolic execution engine then aborts the exploration of this alternative, and turns its attention to the other, correct branch direction.

Non-deterministic events add another dimension to the tradeoff between logging overhead and bug reproduction time. Either we log all non-deterministic events during execution so that we can reproduce them exactly during replay, but we add overhead at runtime. Or we do not log all of them, but then a non-deterministic event during replay may produce an outcome different from the one during actual execution. The importance of this tradeoff is amplified if some branches are not logged. If all branches are logged, then with high likelihood a different outcome of a non-deterministic event during replay is detected quickly, because a subsequent branch takes a direction different from the one logged during execution. If, however, not all branches are logged and in particular branches that follow the non-deterministic

event are not logged, then the replay may require considerable searching to discover the path followed during the actual execution. We explore this tradeoff for system calls, one of the principal sources of non-determinism during sequential execution.

Overview of results. We have implemented the three branch instrumentation methods described above, in addition to the naive approach that logs all branches. We explore the tradeoff between instrumentation overhead and bug reproduction time using an open-source Web server, the *diff* utility, and four coreutils programs. We find that the combined approach strikes a better balance between instrumentation overhead and bug reproduction time than the other two. Moreover, it enables bug reproduction times that are only slightly higher than those for the approach based solely on static analysis. In contrast, this latter approach only marginally reduces logging overhead compared to logging all branches, whereas the approach based solely on dynamic analysis leads to excessively long times to reproduce bugs. More concretely, our results show that the combined approach reduces instrumentation overhead between 10% and 92%, compared to the approach based solely on static analysis. At the same time, it always reproduces the bugs we considered within the allotted time (1 hour), whereas the dynamic approach fails to do so in 6 out of 16 cases. In all circumstances, we find that selectively logging the results of system calls is advantageous: it limits bug reproduction time, and adds only marginally to instrumentation overhead.

Contributions. The contributions of this paper are:

1. The use of symbolic execution prior to shipping the program to discover which branches depend on input and which not.
2. An optimized approach to symbolic execution for bug reproduction that is guided by a symbolic branch log collected at the user's site.
3. The exploration of the tradeoffs between the amount of pre-deployment symbolic execution, the instrumentation overhead resulting from logging the outcome of branches, and the time necessary to reproduce a bug at the developer's site.
4. A combined static-dynamic method for deciding which branches to instrument. The method leads to a better tradeoff than previous systems, making the approach of logging branches more practical and reducing the debugging time.
5. A quantification of the impact of logging the result of selected system calls, demonstrating that it only marginally increases the instrumentation overhead, but considerably shortens the bug reproduction time.

Roadmap. The rest of this paper is organized as follows. Section 2 describes the program analysis and instrumentation methods we study. Section 3 shows how we modify a symbolic execution engine to take as input a partial branch recorded at the user site to reproduce a bug. Section 4 describes some implementation details and our experimental methodology. Section 5 presents the results for an open-source Web server, diff, four coreutils programs, and two microbenchmarks. Section 6 discusses the results and our future work. Section 7 summarizes the related work. Finally, Section 8 concludes the paper.

2. Program Analysis and Instrumentation

Our approach involves analyzing the program to find the symbolic branches and instrument them for logging. We study both dynamic and static analyses. Our instrumentation may use the results of the dynamic analysis only, those of the static analysis only, or combine the results of both analyses. Next, we describe our analyses and instrumentation methods.

2.1 Dynamic Analysis

Our dynamic analysis is based on symbolic execution. We mark input to the program as symbolic and then use symbolic execution to determine whether or not a branch condition depends on input.

Symbolic execution repeatedly and systematically explores all paths (and thus branches) in a program. In the particular form of symbolic execution used in this paper (sometimes called concolic execution [Sen 2005]), the symbolic engine executes a number of runs of the program, each with a different concrete input. Initially, it starts with a randomly chosen set of values for the input and proceeds down the execution path of the program. At each symbolic branch, it computes the branch condition for the direction of the branch followed, and adds this condition to the constraint set for this run. When the run terminates, the overall constraint set describes the set of branch conditions that have to be true for the program to follow the path that occurred in this particular run. One of the conditions is then negated, the constraint set is solved to obtain a new input, and a new run is started.

We initially mark $argv$ as symbolic, as well as the return values of any functions that return input. During symbolic execution, we track which variables depend on input variables, and mark those as symbolic as well. When we execute a branch for the first time, we label it symbolic if any of the variables on which the branch condition depends is symbolic, and concrete otherwise. If during further symbolic execution we revisit a branch labeled symbolic, it stays that way. If, however, we revisit a branch labeled concrete, and now its branch condition depends on at least one symbolic variable, we relabel that branch as symbolic.

Symbolic execution tries to explore all program paths, and is therefore very time-consuming. If it is able to explore all paths, then all branches are visited, with some marked symbolic and some marked concrete. However, this usually

can only be done for very small programs. For others, it is necessary to cut off symbolic execution after some amount of time. As a result, at the end of the analysis, in addition to branches labeled symbolic and concrete, some branches remain unlabeled.

All branches marked symbolic are indeed symbolic, but some branches marked concrete may actually be symbolic, because symbolic execution was terminated before the branch was visited with a condition depending on input. The unvisited branches may be either symbolic or concrete.

2.2 Static Analysis

We use interprocedural, path-sensitive static analysis, in which we use a combination of dataflow and points-to analysis. The basic idea of the algorithm is to identify the sources of input (typically I/O functions or arguments to the program), and construct a list of variables whose values depend on input and are thus symbolic. Symbolic branches are then identified as the branches whose condition depends on at least one symbolic variable.

The algorithm works by maintaining a queue of functions to analyze. Initially, the queue only contains the $main$ function. New entries are queued in as function calls are discovered. The set of symbolic variables is initialized to $argv$. In the initialization, the functions that are normally used to read input are marked as returning symbolic values.

Algorithms 1 and 2 show a simplified version of the algorithm that analyzes each function. Each instruction in the program is visited, and the $doInst$ method is called. The dataflow algorithm takes care of loops by using a fixed-point algorithm so that instructions in a loop body are revisited (and $doInst$ is called) only as long as the algorithm output changes.

For an assignment instruction (i.e., an instruction of the form $variable$ = $expression$;), $doInstr$ resolves the variables to which the expression may be pointing, and checks whether any of these is already known to be symbolic. If this is the case, it adds $variable$ to the list of symbolic variables, otherwise it continues. If the instruction is a function call, the algorithm first checks whether the function has already been analyzed or not. If it has, it looks up the results to determine whether with the current set of parameters the function can propagate symbolic memory or not, and updates the list of symbolic variables accordingly. If the function has not yet been visited [1], the algorithm enqueues it for analysis with a reference to the current instruction, so that analysis of the current function can resume when the function has been visited.

The $doStatment$ method in algorithm 2 is called by the dataflow framework on each control flow statement. For if statements, it resolves the list of variables to which the condition expression may be pointing. If any of them is symbolic, the branch is labeled symbolic.

While the algorithm in Figure 1 is simplified for the sake of clarity, our actual implementation handles the fact (1) that symbolic variables can be propagated to global variables; (2)

[1] More precisely, if the function has not been visited with the particular combination of symbolic and concrete parameters encountered here.

Algorithm 1: Static analysis algorithm propagating symbolic information (simplified)

```
   /* called on each instruction in the program
      as many times as required by the fixed
      point dataflow algorithm                     */
1  method doInstr(instruction i):;
2  begin
3      match i with begin
           /* assignment                            */
4          case target_variable = expression;
5          begin
               /* If any of the variables
                  referenced by expression symbolic
                  is symbolic mark targe_variable
                  symbolic                          */
6              if isSymbolic(e) then
7                  makeSymbolic(target_variable);
8              end
9          end
           /* function call                         */
10         case: target_variable = fun_name(parameters);
11         begin
12             symbolic_params =
               getSymbolicParameters(parameters);
               /* If we already visited fun_name
                  with this combination of symbolic
                  parameters, propagate symbolic
                  flag                              */
13             if alreadyVisited(fun_name,
               symbolic_params) then
14                 if returnsSymbolicMemory(fun_name)
                   then
15                     makeSymbolic(target_variable);
16                 end
17             end
18             else
                   /* fun_name not visited yet.
                      Queue it and stop analysis of
                      current function. The
                      algorithm will return to this
                      location once fun_name has
                      been visited                  */
19                 queueFunction(fun_name,
                   symbolic_parameters, i);
20                 return abort
21             end
22         end
23     end
24     return continue
25 end
```

Algorithm 2: Static analysis algorithm identifying symbolic branches (simplified)

```
   /* called on each control flow statement of
      the program.                                */
1  method doStatement(statement s): begin
2      match s with;
3      begin
4          Branch(condition_expression)  begin
5              if isSymbolic(condition_expression) then
6                  logThisBranch();
7              end
8          end
9      end
10     return continue;
11 end
```

that the state of global variables changes depending on the path that is being analyzed; and (3) that functions may propagate symbolic variables not only to their return variables, but also to their parameters (passed by reference) or to global variables.

Static analysis is imprecise because the points-to analysis tends to over-estimate the set of aliases to which a variable may point. As a result, all symbolic branches in the program are labeled symbolic by the static analysis, but some concrete branches may also be labeled symbolic. All branches labeled concrete are indeed concrete.

2.3 Program Instrumentation

The developer instruments the branches in the program before the code is shipped. We consider four methods for instrumentation:

- *dynamic* instruments branches according to dynamic analysis.
- *static* instruments branches according to static analysis.
- *dynamic+static* instruments branches according to a combination of dynamic and static analysis.
- *all branches* instruments all branches.

Regardless of which method is used, the list of instrumented branches is retained by the developer, because it is needed to reproduce the bug (Section 3).

Dynamic method. After dynamic analysis, branches are labeled symbolic or concrete, or remain unlabeled. The *dynamic* method only instruments the branches labeled as symbolic. By the nature of the dynamic analysis, we are certain that these branches are symbolic. We do not instrument the branches labeled as concrete, since application branches are typically either always symbolic or always concrete. We also do not instrument the unlabeled branches. The *dynamic* method thus potentially underestimates the number of branches that need to be instrumented. In essence, *dynamic* favors reducing instrumentation overhead at the expense of increased bug reproduction time.

Static method. After static analysis, branches are labeled symbolic or concrete. We instrument the branches labeled as symbolic. By the nature of static analysis, the *static* method guarantees that all symbolic branches are instrumented, but it may instrument a number of concrete branches as well. *Static* therefore favors bug reproduction time at the expense of increased instrumentation overhead.

Dynamic+static method. In the combined method, we run both the static and the dynamic analysis, the latter for a limited time. The dynamic analysis labels branches as symbolic or concrete, or they may remain unlabeled. The static analysis labels branches as symbolic or concrete. The combined method instruments the branches (1) that are labeled symbolic by the dynamic analysis, and (2) that are labeled symbolic by the static analysis, with the exception of those labeled concrete by the dynamic analysis. In other words, when a branch is not visited by dynamic analysis, we instrument it based on the outcome of the static analysis, because this is the only information about this branch. When a branch is visited by dynamic analysis, we instrument it based on the outcome of this analysis. For branches labeled symbolic by dynamic analysis, this is obvious as they are guaranteed to be symbolic and have been labeled symbolic by static analysis as well. For branches labeled concrete by dynamic analysis, this means that we potentially override the outcome of static analysis which may have labeled these branches symbolic. The reasons for this decision are that (1) static analysis may conservatively label concrete branches symbolic, due to an imprecise points-to analysis, and (2) application branches are typically always concrete or always symbolic, as mentioned above.

Dynamic+static may be imprecise in two ways. Symbolic branches may or may not be instrumented. The latter case occurs if they are left concrete by the symbolic execution (for instance, due to the limited coverage of the symbolic execution). Concrete branches may or may not be instrumented. The former case occurs when the static analysis mistakenly labels them as symbolic and they are not visited during symbolic execution.

Although seemingly suffering from a greater degree of imprecision than the other methods, our evaluation shows that this method actually leads to the best tradeoff between instrumentation overhead and time necessary to reproduce the bug.

Logging system calls. In addition to deciding which branches to instrument, we also consider the choice of whether or not to log the results of certain system calls. Doing so adds to the runtime overhead, but can be very beneficial for system calls that can produce a large number of possible outcomes during replay. For example, consider a *select()* system call for reading from any of N file descriptors. Without information about which descriptors became ready and when, symbolic execution during replay would have to explore all

combinations of N available descriptors upon each return from *select()*. To avoid having to explore all possible combinations, we instrument the code to log the descriptors that are available when a call to *select()* returns. During replay, we simply re-create these conditions. For the same reasons, it makes sense to instrument calls to *read* to log the number of bytes read.

We log the results of all system calls for which logging considerably simplifies replay, including *select()* and *read()*. The input data itself is never logged. In principle, all instrumentation methods can be combined with logging system calls.

3. Reproducing a Bug

3.1 Replay Algorithm

We use a modified symbolic execution engine [Crameri 2009] to reproduce bugs. The following information is available to the engine prior to bug reproduction: a list of all instrumented branches (saved when the program was instrumented – see Section 2.3), and the bitvector indicating which way the instrumented branches were taken (one bit for each instrumented branch taken during execution at the user site). When the symbolic execution engine encounters a branch, it immediately knows whether or not the branch is symbolic, because it can check whether the branch condition depends on the input.

We refer to a run of the symbolic execution engine as the execution of the engine with a single set of inputs, until it either finds the path to the bug or aborts. A run is aborted when the engine discovers that it is on a path that deviates from the path described by the received bitvector. Each run is started with the bitvector as received from the user site. A constraint set is associated with each run, describing the path followed by the run through the program, and consisting of the conjunction of the conditions for the branch directions taken so far in the run. To later explore alternative paths should the current run abort, the engine also maintains a list of pending constraint sets, describing these alternative unexplored paths.

The engine performs a number of these runs. The initial run is done with random inputs. Subsequent runs use an input resulting from the solution of a constraint set by the constraint solver.

For example, in Listing 1, the set of constraints: {*not (option == 'a'); option == 'b'*} needs to be solved to take the program in the `else if` branch at line 6.

When visiting the `else if` branch at line 6, the engine puts in the pending list the following constraint set: {*not (option == 'a'); not (option == 'b')*}.

During a run, the engine proceeds normally for instructions other than branches. For branches, because of the imprecision of the decision of which branches to instrument, the following four cases have to be distinguished.

1. **The branch is symbolic and not instrumented.** The constraint for the particular direction of the branch taken is added to the constraint set for the run, and symbolic execution proceeds. In addition, a new constraint set is formed by adding the negated constraint to the constraint set for the run. The new constraint set is put on the list of pending constraint sets. The bitvector is left untouched.
2. **The branch is symbolic and instrumented:** The engine takes the next bit out of the bitvector, and compares the direction that was taken during recorded execution to the direction the symbolic execution would take with its input.
   (a) If the two are the same, the constraint is added to the constraint set for the run, and the symbolic execution proceeds.
   (b) If not, the constraint implied by the direction of the branch taken during recorded execution is negated and added to the constraint set for the run. This constraint set is added to the list of pending constraint sets, and the run aborts.
3. **The branch is concrete and instrumented:** The engine takes the next bit out of the bitvector, and compares if the direction that was taken during recorded execution is the same as the direction the symbolic execution would take with the current input.
   (a) If yes, it proceeds further.
   (b) If not, this run of the symbolic execution is aborted.
   The latter case can only occur as a result of the run earlier having taken the wrong direction on a branch that (because of insufficient instrumentation) was not instrumented but should have been.
4. **The branch is concrete and not instrumented:** The engine proceeds. The bitvector is left untouched.

When a run is aborted, the engine looks at the pending list of constraint sets, picks one, solves it, and starts another run with the input resulting from the solver. When this new run passes the branch at which it was produced, the new input, by construction, causes the symbolic execution to take the direction opposite from the one followed in the run during which the constraint set was produced.

3.2 Replay Under Different Instrumentation Methods

The *all branches* instrumentation method instruments all symbolic branches (and all concrete ones as well). Thus, cases 1 and 4 above cannot occur with this method. Furthermore, case 3(b) cannot occur either. The reason is that the engine always proceeds past a symbolic branch in the same direction as recorded during execution. When the run hits a concrete branch, since this branch does not depend on input, the engine is bound to follow the correct direction.

The *static* method also instruments all symbolic branches, since imprecision in the points-to analysis is resolved conservatively (if a pointer might depend on input, it is flagged

symbolic). Under this method, cases 1 and 3(b) cannot occur. The latter case cannot occur for the same reason above.

The *dynamic* method may not instrument all symbolic branches, because it may not run long enough to find them. Similarly, the *dynamic+static* method may fail to instrument all symbolic branches, but only when symbolic execution does not run long enough *and* static analysis is inaccurate. In these cases, a run can take the wrong direction at a symbolic branch that was not instrumented, and later hit a concrete branch for which the input fails to satisfy the branch condition. In this case, the engine needs to back up and explore an alternative direction at a symbolic but uninstrumented branch. The constraint sets for these alternative directions reside on the pending list. The search can use any heuristic for deciding which constraint set to pick from the pending list. We currently use a simple depth-first approach.

3.3 Replaying System Calls

As mentioned in Section 2.3, we consider scenarios with and without instrumentation for logging the results of certain system calls.

When these system calls are not logged, we replay them using models of their behavior. The models use symbolic variables to allow the engine to reproduce any behavior that may be produced by the kernel. For instance, for the *read()* system call, we use a symbolic variable for the return value that determines how much input is read. This symbolic variable is constrained to be between -1 and the amount of input requested. During replay, a program executing the *read()* call initially returns the amount of (symbolic) input requested, and execution carries on. Because the return value of the system call is symbolic, if the program checks it in a branch and if the branch has been logged, the log specifies which direction needs to be taken. If that direction fails, the symbolic execution engine aborts the run. Eventually, the number of bytes actually read at the user's site is found.

When these system calls are logged, we replay their execution based on the logs. During replay, the calls for which there are logged results always return exactly the recorded value. Thus, the symbolic execution engine need not search for the actual call results.

4. Implementation and Methodology

Software. For program instrumentation and analysis, we use CIL (C Intermediate Language [Necula 2002]), which is a collection of tools that allow simple analysis and modification of C code.

For static analysis we start by merging all the source code files of the program in one large C file. This allows us to run the analysis on the whole program, making the results more accurate.

We then use two CIL plug-ins for the dataflow and points-to analysis.

For symbolic execution, we use a home-grown concolic execution engine for C programs [Crameri 2009]. The engine instruments the C program and links it with a runtime library for logging constraints.

We also use CIL to instrument the branches in the program. The instrumentation simply uses a bit per branch in a large buffer, and flushes the buffer to disk when it is full. We use a buffer of 4KB in order to avoid writing to disk too often. We do not use any form of online compression, as this would impose additional CPU overhead. Moreover, we could have used a simple branch prediction algorithm to avoid logging all instrumented branches, but this would have required recording the program location for each logged branch. This approach would have required at least another 32 bits of storage per branch, probably ruining any savings obtained by the prediction algorithm.

Benchmarks. We first evaluate the instrumentation overhead in isolation using two microbenchmarks. Next, we reproduce real bugs previously reported in the coreutils programs [Cadar 2008]. Then, we evaluate the tradeoff between instrumentation overhead and bug reproduction time using an open-source Web server, the *uServer* [Pariag 2007]. The *uServer* was designed for conducting performance experiments related to Internet service and operating system design and implementation. We use version 0.6.0, which has approximately 32K lines of code. In our final experiment, we again evaluate the tradeoff between instrumentation overhead and bug reproduction, but this time with the *diff* utility. *Diff* is an input-intensive application that pinpoints the differences between two files provided as input. It contains about 22K lines of code. For all experiments we link the programs with the uClibc library [uClibc].

We study five configurations of each benchmark: four resulting from our four instrumentation methods plus a configuration called *none*, which involves no instrumentation. For the instrumented benchmarks, unless mentioned otherwise, selective system call logging is turned on.

Hardware. Our experimental setup consists of two machines with two quad-core 2.27-GHz Xeon CPUs, 24 GBytes of memory, and a 160-GByte 7200rpm hard disk drive.

5. Evaluation

5.1 Microbenchmarks

We evaluate the cost of the instrumentation by using two simple microbenchmarks. The first microbenchmark comprises a loop that increments a counter 1 billion times. The loop condition (checking the loop bound) consists of a single branch, executing as many times as there are iterations.

We compare the *none* (no instrumentation) and *all branches* versions of this microbenchmark using the Linux *perf* profiler. The results show that the branch logging instrumentation takes 17 instructions on average, including the periodic flushing of the log to disk. In terms of running time, we see

an average cost of 3 nanoseconds per instrumented branch (with an average of 2.1 instructions per cycle), for a total overhead of 107%. While considerable, this overhead is still lower than that reported in ODR [Altekar 2009] (about 200% just for the branch recording). The most likely reason is that our instrumentation only logs one bit per branch, while ODR uses a more complicated branch prediction algorithm.

We run a second microbenchmark consisting of the example in Listing 1, which computes the fibonacci sequence for one of two numbers. We instrument this program using the four configurations of our system. Not surprisingly, the configurations other than *all branches* instrument only the symbolic branches corresponding to lines 4 and 6. The results are consistent with our previous microbenchmark: an average overhead of 17 instructions per instrumented branch or about 3 nanoseconds. The *all branches* configuration suffers from a total overhead of 110%, whereas the three others do not incur any noticeable overhead because only two branches are logged.

5.2 Coreutils

We now evaluate our approaches using four real bugs in different programs from the Unix coreutils set: *mkdir*, *mknod*, *mkfifo*, and *paste*. We ran the programs with up to 10 arguments, each 100 bytes long.

Branch behavior. Recall that our approach is based on two assumptions about branches: (1) that there are many (concrete) branches whose outcomes do not depend on input, and (2) that if a branch is executed once with a concrete (symbolic) branch condition, it is most likely always executed with a concrete (symbolic) condition.

To check these assumptions, we modify our symbolic execution engine to run with concrete inputs (instead of generating concrete inputs itself to explore different paths). In addition, at each executed branch, we record whether it is executed with a symbolic or a concrete branch condition.

We show the results of a sample run of $mkdir$ in Figure 1; the graphs for the other 3 programs are similar. The figure shows per-branch-location statistics for this experiment. We use the term "branch location" to mean the location of a branch in the source code, and "branch execution" to mean the actual execution of a branch location during run time. Each point on the x axis denotes a branch location that is executed at least once. The y axis shows how many times each branch is executed. The gray bars denote the overall number of branch executions, whereas the black bars denote the number of executions with a symbolic condition. The black bars are therefore a subset of the gray bars.

As the figure illustrates, only a limited number of branch locations is responsible for all the symbolic branch executions. Furthermore, where black bars are present, they completely cover the gray bars. This shows that a particular branch is executed either always with concrete conditions or always with symbolic conditions. These observations support our two assumptions.

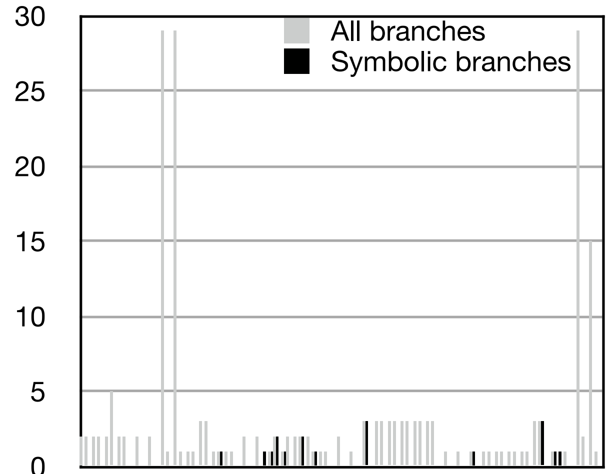

Figure 1. Number of executions of each branch in a sample run of $mkdir$. The overlaid black bars represent the branches executed with symbolic conditions.

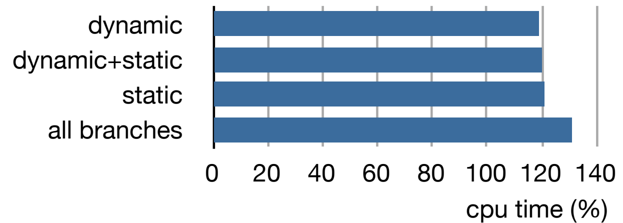

Figure 2. CPU time of *mkdir* instrumented with the four configurations of our system. Results are normalized to the non-instrumented version.

Instrumentation overhead. Figure 2 shows the CPU time associated with the instrumentation for $mkdir$ (again, the results for the other programs are similar). Those results are obtained by running the program in the symbolic execution engine for one hour. The figure shows that the time is almost identical for the *dynamic*, *dynamic+static*, and *static* configurations. The static and dynamic analyses produce accurate results in those programs. The *all branches* configuration is the slowest, with an overhead of 31%.

Reproducing bugs. Each program suffers a crash bug that only manifest itself when a very specific combination of arguments is used. For instance, the bug in $paste$ occurs with the following command line:

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
```

Table 1 shows the time needed to reproduce the crash bugs in the four programs. The programs being relatively small, symbolic execution is able to cover most of the important branches in a very short amount of time, and static analysis produces accurate results. Thus, we can reproduce

the bug in less than two seconds in all of the four instrumented configurations.

Program	Replay time
mkdir	1 sec
mknod	1 sec
mkfifo	1 sec
paste	1.5 sec

Table 1. Time needed to replay a real bug in four coreutils programs. The results are the same with all four configurations of our system.

These bugs have also been used to evaluate ESD [Zamfir 2010]. Interestingly, ESD took significantly more time to reproduce the bugs (between 10 and 15 seconds, albeit with no runtime overhead). This can be attributed to the fact that our system essentially knows the exact path to the bug (during bug reproduction), whereas ESD needs to search many paths.

5.3 uServer

We further evaluate our system using a much larger application (32K lines of code), the *uServer* [Pariag 2007], an open-source Web server sometimes used for performance studies. Unlike the coreutils programs, which are very small, this benchmark elucidates the differences between the approaches for deciding which branches to instrument, and the tradeoff between instrumentation overhead and bug reproduction time.

Branch behavior. We run the *uServer* in our modified symbolic execution environment with 5,000 HTTP requests to demonstrate the nature of the branches (symbolic or concrete). In total, approximately 18 million branches are executed, out of which only 1.8 million or roughly 10% are symbolic. These 1.8 million symbolic branches correspond to multiple executions of the same 53 branch locations in the program.

Figure 3 shows per-branch-location statistics for this experiment. As we can see, most of the black bars entirely cover their corresponding gray bars. This means that these particular branch locations are executed either always with concrete conditions, or always with symbolic conditions. However, the situation is slightly different for the branches in uClibc, where in some cases the black bars almost but not completely cover the gray bars. This situation corresponds to library functions that are sometimes called with concrete values. In this experiment, the number of those cases was very small.

The figure also shows that most branches are executed in the library (81%). However, only 28% of the symbolic branches are executed in the library.

Version	# of instrumented branch locations	
	LC	HC
dynamic	78	246
dynamic + static	1654	1490
static	2104	
all branches	5104	

Table 2. Number of branch locations instrumented in the *uServer* with the different configurations of our system.

Identifying symbolic branch locations. In total, there are 5104 branch locations in the *uServer* code (and 8516 in uClibc).

Table 2 shows the number of instrumented branches in the *uServer* for each configuration of our system. We symbolically execute the *uServer* using 200 bytes of symbolic memory for each accepted connection, and for each file descriptor. We stop the symbolic execution phase after one hour and two hours, obtaining a coverage of 20% (denoted LC for lower coverage) and 33% (HC for higher coverage), respectively. Running longer does not significantly improve branch coverage.

Out of a total of 5104 branches, *static* marks 2104 as symbolic, *dynamic* 246, and *dynamic+static* 1490, in the HC configuration. In addition, with *dynamic*, 1434 branches are marked as concrete, and the remaining branches are not visited.

For the static analysis tool, we need to merge all the source files of both the application and the library. Unfortunately, doing so resulted in a file too large for the points-to analysis to handle.[2] Therefore we perform static analysis only on the *uServer* application code. All branches in the library are treated as symbolic by the static analysis.

With less coverage, there are more branches left unvisited, and therefore more opportunity for the static phase to mark them symbolic. This is why there are fewer instrumented branches in the *dynamic* configuration and more in the *dynamic+static* configuration. Those numbers are used to highlight the effect of branch coverage in our approach.

Instrumentation overhead. We use the httperf [Mosberger 1998] benchmarking tool to compare the performance of the configurations for the *uServer*. We run httperf with a static workload saturating the CPU core on which the *uServer* is running. We consider three performance metrics: number of instrumented branches executed, CPU time, and storage requirement of the instrumentation. All three are roughly proportional to each other.

Figure 4(a) shows CPU time (relative to the non-instrumented version). As we can see, the overhead of *all branches* is significant. The results of *static* are only marginally better, since it instruments all branches in the uClibc library.

[2] After six hours, the analysis had made little progress and we aborted it.

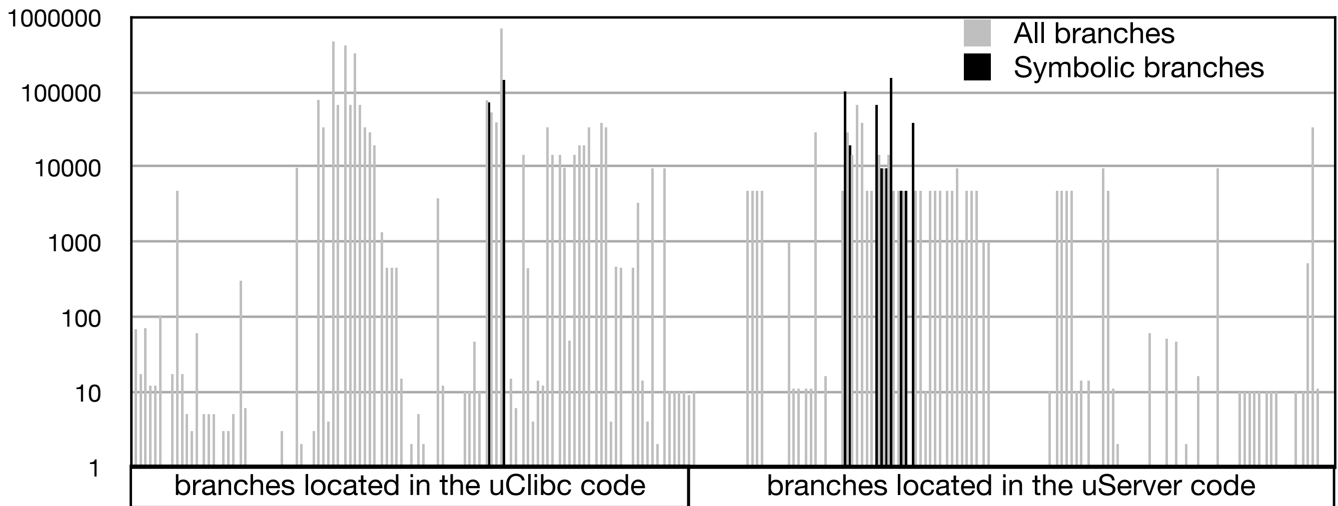

Figure 3. Number of executions of each branch location in a sample run of the *uServer*. The y axis is in log scale.

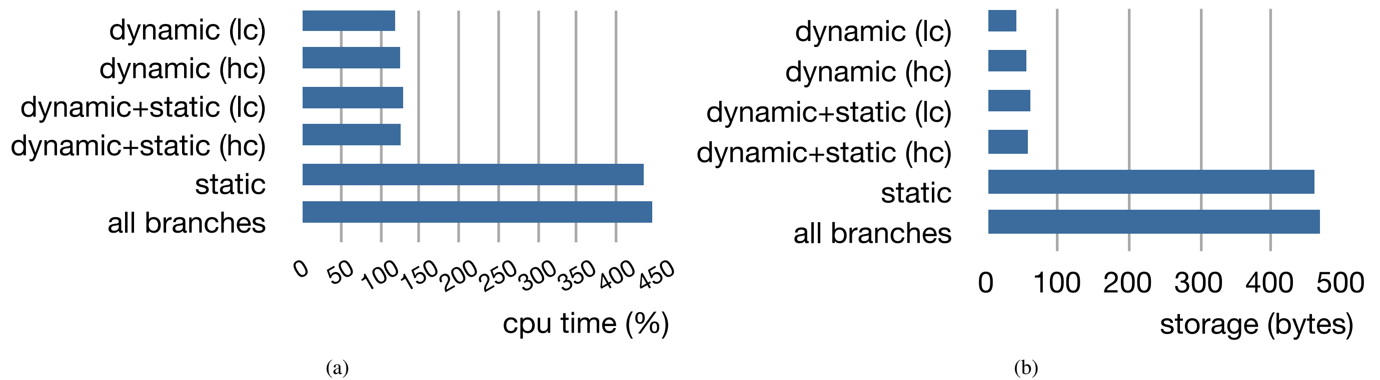

Figure 4. CPU time (a) and storage requirement (b) of the *uServer* instrumented with the four configurations of our system.

The two configurations using dynamic analysis perform notably better. The overhead is 17% and 20%, respectively, for the *dynamic* and *dynamic+static* configurations. This is not surprising for *dynamic*, as it instruments only 284 branch locations. The *dynamic+static* configuration instruments many more branch locations, but still far fewer than *all* or *static*.

The coverage obtained with symbolic execution affects the instrumentation overhead of *dynamic* and *dynamic+static*. Increased coverage increases the overhead of *dynamic*, since symbolic execution instruments more branches. In contrast, increased coverage leads to reduced instrumentation in *dynamic+static*, corresponding to branches marked symbolic by the static analysis, left unvisited by the dynamic analysis with low coverage, and marked concrete by the dynamic analysis with high coverage.

Figure 4(b) shows the storage requirements per HTTP request processed by the Web server. The storage overhead is reasonable; around 50 bytes per request in the *dynamic* and *dynamic+static* configurations. This is roughly the same number of bytes in a typical entry in the access log of the Web server.

Both the processing and storage overheads are more significant in the *static* and *all branches* configurations. These configurations represent worst-case scenarios for the areas of the code that are not covered by the symbolic execution.

When a bug occurs, the branch log has to be transferred to the developer. Compression can be used to reduce the transfer time. We observe a compression ratio of 10-20x using gzip. In Section 6, we discuss how to deal with long-running executions, which could generate extremely large branch logs.

Reproducing bugs. To evaluate the amount of effort required to reproduce a bug, we run the *uServer* with five different input scenarios. To demonstrate the impact of code coverage on our approach, we design those scenarios to hit different code areas of the HTTP parser. More specifically, we use HTTP queries of various lengths (between 5 to 400 bytes), with different HTTP methods (e.g., GET, POST) and parameters (e.g., Cookies, Content-Length). We crash the

Version	Exp. 1		Exp. 2		Exp. 3		Exp. 4		Exp. 5	
	LC	HC	LC	HC	LC	HC	LC	HC	LC	HC
dynamic	27s	27s	2877s	79s	∞	170s	∞	287s	∞	168s
dynamic+static	27s	27s	79s	79s	532s	170s	175s	175s	248s	168s
static	27s		79s		170s		175s		168s	
all branches	27s		79s		170s		175s		168s	

Table 3. Time in seconds needed to reproduce each of the five input scenarios to the *uServer* with the four instrumented configurations of our system. The infinity symbol means that the experiment did not terminate in one hour.

Version		# of symbolic branch locations logged / corresponding # of executions		# of symbolic branch locations NOT logged / corresponding # of executions	
		LC	HC	LC	HC
Exp. 1	dynamic	18 / 112	18 / 112	0	0
	dynamic+static	18 / 112	18 / 112	0	0
	static	18 / 112		0	
	all branches	18 / 112		0	
Exp. 2	dynamic	25 / 129913	39 / 2215	11 / 23062	0
	dynamic+static	36 / 2105	39 / 2215	3 / 110	0
	static	39 / 2215		0	
	all branches	39 / 2215		0	
Exp. 3	dynamic	25 / 554617	42 / 28848	17 / 48485	0
	dynamic+static	45 / 10971	42 / 28848	3 / 1023	0
	static	42 / 28848		0	
	all branches	42 / 28848		0	
Exp. 4	dynamic	24 / 236608	43 / 24012	21 / 185945	6 / 268
	dynamic+static	46 / 11089	48 / 12111	3 / 1023	1 / 1
	static	49 / 12112		0	
	all branches	49 / 12112		0	
Exp. 5	dynamic	25 / 410723	44 / 29136	15 / 45706	0
	dynamic+static	46 / 54539	44 / 28785	3 / 3391	0
	static	44 / 28785		0	
	all branches	44 / 28785		0	

Table 4. Number of symbolic branch locations and symbolic branch executions logged and not logged for each configuration of each experiment of Table 3.

server by sending it a SEGFAULT signal after sending it the input, making sure it crashes at the same location in the code for all four versions. After replay, we verify that each configuration produced input that correctly leads to the same location.

Table 3 shows the bug reproduction times for all four instrumented versions of the five scenarios. Table 4 shows the corresponding number of symbolic branch locations logged and not logged, as well as the number of actual symbolic branch executions. Both tables include the configuration with low coverage (LC) and high coverage (HC).

Unsurprisingly, the *all branches* and *static* versions, which instrument all symbolic branches, perform best. Of course, these versions do so at high runtime and storage overheads (Figure 4).

Dynamic+static in most cases performs only slightly worse than *static*, despite the much lower instrumentation overhead of the former configuration. *Dynamic* comes last, with many LC experiments not finishing in one hour. This is not surprising, as the number of branches identified as symbolic, and therefore the amount of logging, is very low. In fact, Tables 3 and 4 show that the number of symbolic branch locations not logged is well correlated with the replay time. As soon as replay encounters more than a dozen symbolic branch locations that are not instrumented, the replay time exceeds one hour. An approach that does not instrument the code at all, would result in even longer bug reproduction times.

Dynamic+static obtains similar results regardless of coverage. The reason is that symbolic execution may incorrectly

Version	Exp. 1		Exp. 4	
	LC	HC	LC	HC
dynamic	112s	112s	∞	712s
dynamic+static	112s	112s	991s	694s
static	87s		362s	
all branches	56s		343s	

Table 5. Time in seconds needed to reproduce two input scenarios with the *uServer* when not logging system call results. The infinity symbol means that the experiment did not terminate in one hour.

classify some branches as concrete, and thus slow down the search. When running longer, those branches may later be marked symbolic, therefore correcting the error. In our experiment, those differences have a marginal effect on *dynamic+static*, which again suggests that this does not happen frequently.

Impact of logging system calls.

By default, we log the results for some key system calls (Section 2.3). For instance, we log the return value of the *read()* system call and the order of ready file descriptors from *select()* calls.

The measurements in Figure 4 include the overhead of logging these return values. As we only log a limited number of values for a few system calls, logging these values introduces little extra work compared to the logging of the branches. As a result, when not logging system call results, the overhead is reduced by a marginal 0.2%.

Tables 5 and 8 present the bug reproduction times of two of our experiments without logging any system calls (we omit the three other experiments for brevity; their results are similar). All configurations of our system take significantly longer to replay, as the symbolic execution engine needs to determine the exact return values of the selected system calls. The *dynamic* and *dynamic+static* configurations are further penalized, since the back-tracking needed by the unlogged symbolic branches compounds the search for the return values. Interestingly, the *static* configuration performs slightly slower than *all branches*, whereas when logging system calls it performs identically. The reason is that fewer concrete branches are logged, therefore the engine takes slightly more time to realize a wrong turn due to a system call.

5.4 Diff

We now consider the *diff* utility. *Diff* is more challenging than the *uServer* for our dynamic analysis, as its behavior depends more heavily on input. Moreover, *diff* generates very long constraint sets, placing a heavy burden on the constraint solver. For these reasons, dynamic analysis attains a coverage of only 20% of branches, during 1 hour of symbolic execution. In total, there are 8840 branches in the program. *dynamic* identifies 440 of them as being symbolic, *static* 4292, and *dynamic+static* 3432.

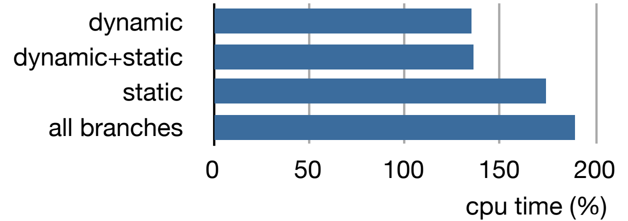

Figure 5. CPU time of *diff* instrumented with the four configurations of our system. Results are normalized to the non-instrumented version.

Version	Exp. 1	Exp. 2
dynamic	∞	∞
dynamic + static	1s	12s
static	1s	12s
all branches	1s	12s

Table 6. Time in seconds needed to reproduce two input scenarios to *diff*. The infinity symbol means that the experiment did not terminate in one hour.

	Version	symbolic branch locations logged / corresponding executions	symbolic branch locations NOT logged / corresponding executions
Exp. 1	dynamic	3 / 2125686	32 / 2369765
	dyn.+static	13 / 904	0
	static	13 / 904	0
	all branches	13 / 904	0
Exp. 2	dynamic	3 / 2478280	24 / 2102506
	dyn.+static	21 / 54623	0
	static	21 / 54623	0
	all branches	21 / 54623	0

Table 7. Number of symbolic branch locations and symbolic branch executions logged and not logged for two input scenarios to *diff*.

Instrumentation overheads. We run *diff* on two sample text files. Figure 5 shows the CPU time of the four configurations of our system, normalized against the non-instrumented execution. Consistent with our prior results, *dynamic* and *dynamic+static* perform best with an overhead of approximately 35%.

Reproducing bugs. We replay two executions of *diff* comparing relatively small but different text files. Tables 6 and 7 list the results of these experiments. Because the coverage obtained during dynamic analysis is relatively low, the *dynamic* configuration is unable to finish the experiments in 1

Version		# of symbolic branch locations logged / corresponding # of executions		# of symbolic branch locations NOT logged / corresponding # of executions	
		LC	HC	LC	HC
Exp. 1	dynamic	40 / 722	43 / 725	8 / 173	5 / 170
	dynamic+static	40 / 722	43 / 725	8 / 173	5 / 170
	static	49 / 871		0	
	all branches	49 / 465		0	
Exp. 4	dynamic	43 / 245017	65031 (61)	21 / 107650	7 /1950
	dynamic+static	64 / 88739	64612 (67)	3 / 7397	1 / 1021
	static	50 / 39581		0	
	all branches	50 / 37346		0	

Table 8. Number of symbolic branch locations and symbolic branch executions logged and not logged for two input scenarios with the *uServer* when not logging system call results.

hour. The few tens of unlogged symbolic branch locations quickly create a very large number of paths to explore, making it impossible for this approach to finish within the allotted time. In contrast, the three other configurations, and in particular *dynamic+static*, do not suffer from any unlogged symbolic branches and therefore replay quickly.

Collectively, these results again demonstrate that *dynamic+static* strikes the best balance between instrumentation overhead and bug reproduction time.

6. Discussion and Future Work

Branch coverage. Our current results suggest that using a branch trace, even partial, is effective at reducing the amount of searching needed to reproduce a specific buggy execution path. To maintain low instrumentation overhead, it is necessary to obtain sufficient coverage with the initial symbolic execution phase. This problem has received attention in the literature ([Cadar 2008; Godefroid 2008]), and significant progress has been made in recent years. While it is not always possible to achieve 100% coverage, this is a typical goal when testing an application *prior to shipping*. Therefore, the testing effort can be leveraged to identify the symbolic branches at the same time. Moreover, manual test cases can be used in conjunction with symbolic execution to boost code coverage, and many applications already have test suites covering most of their codes.

Constraint solving. Symbolic execution is limited by the ability to solve the resulting constraints. In particular, certain types of programs generate constraints that current state of the art solvers cannot solve. Constraint solving is an active research topic and our approach should directly benefit from any advances in this field.

Non-determinism. Two approaches exist for dealing with non-determinism, resulting, for instance, from system calls or random number generators. One can either log the outcome of the non-deterministic event or one can treat it as (symbolic) input during replay. In this paper, we strike a middle ground between these approaches, logging the outcome of non-deterministic events that are likely to cause a great deal of search during replay. The results in Section 5 validate this approach, but a more comprehensive treatment of non-deterministic events could be explored.

Multithreading. We can extend our system to support multi-threaded applications by modifying it in two ways. First, the branch trace needs to be split into multiple traces, one per thread. Second, the ordering of thread execution needs to be recorded as well. Implementing the first modification is trivial, and is unlikely to impose any significant additional overhead. The second modification is more difficult. Others [Altekar 2009; Zamfir 2010] have experimented with ideas for recreating a suitable thread scheduling to find race conditions. Our approach of logging a partial trace of branches is complementary to those efforts and could considerably speed up the replay of multi-threaded programs with races.

Long-running applications. The storage overhead and replay time of long-running applications can be problematic. Consider, for example, the case of a Web server running for weeks before crashing. Our current approach reduces storage overhead as much as possible, but with these applications this overhead may still be high. Furthermore, replaying such a long trace may be infeasible, as it will be longer than the original run. Pushing the concept of a partial branch trace further, our approach could be extended to simplify this problem by implementing support for checkpointing. An instrumented application could periodically take a checkpoint of its state and discard the current branch log. Logging branches would then continue from the checkpoint only. The checkpoint would include enough information on the data structures of the program (but not its content). With this information, a symbolic execution engine can treat their content as symbolic, and replay the branch log starting from there. We leave the implementation and the associated research questions of the checkpointing mechanism for future work.

Concolic vs. symbolic. The particular form of symbolic execution we use in this paper is called concolic execution [Sen

2005]. The main difference from pure symbolic execution (as in [Cadar 2008], for instance) is that the engine repeatedly executes the program from beginning to end with concrete inputs, instead of exploring multiple paths in parallel. This implementation difference has no fundamental impact on our system, as in both cases the engine can select the paths in the same order. On one hand, the fact that the application is rerun from the beginning for every path imposes some additional overhead. On the other hand, because concrete inputs are always used, it makes the work of the solver easier. In many instances, branch conditions are already satisfied by the random input chosen. To the best of our knowledge, no comparative studies of the impact of the different implementations have been published yet.

7. Related Work

At one end of the spectrum between instrumentation overhead and bug reproduction effort are record-replay systems that try to capture the interactions between the program and its environment. Different systems capture interactions at different levels. Most systems capture them at the system call or library level. For example, ReVirt [Dunlap 2002] logs interactions at the virtual machine level. All record-replay systems suffer from the overhead of logging the interactions. To reduce this overhead, R2 [Guo 2008] asks the developer to manually specify at what interfaces to capture the program's interactions with its environment.

At the other end of the spectrum are conventional bug reporting systems, which provide a coredump, but no indication of how the program arrived at the buggy state. Obviously, there is no recording overhead, but it takes considerable manual search to find out how the problem came about.

ESD (Execution Synthesis Debugger [Zamfir 2010]) is an attempt to automate some of that search. It tries to do so without recording any information about the program execution. Instead, it uses the stack trace at the time of the program crash, and symbolically executes the program to find the path to the bug location. Although it uses static analysis and other optimizations to reduce the number of paths it needs to explore, it remains fundamentally limited by the exponential path explosion of symbolic execution. Our approach instead performs logging of a set of judiciously chosen branches. This allows us to speed up the automated search with limited runtime overhead. As our results with the coreutils show, our methods reproduce bugs faster, albeit at some modest cost in runtime.

BBR (Better Bug Reporting, [Castro 2008]) investigates the same tradeoff, although more from the perspective of maintaining privacy when a bug is reported to the developer. During execution it logs the program's inputs. After a crash, it replays the entire execution on the user machine based on those inputs. Replay uses an instrumented version of the program that collects the constraints implied by the direction of the branches taken in the program. An input set that satisfies these constraints is then returned to the developer. Unlike BBR, we do not log the users' inputs, or require whole-program replay and the execution of a constraint solver on the user machine. Instead, we incur limited logging overhead at the user site and some exploration of alternative paths at the developer site.

Another approach for maintaining privacy is explored by the Panalyst system [Wang 2008]. After a runtime error, the user initially reports only public information. The developer tries to exploit this information using symbolic execution, but can query the user for additional information. The user can choose whether or not to respond to the developer queries. In the limit, Panalyst's effectiveness is constrained by the exponential cost of symbolic execution.

Triage [Tucek 2006] explores yet another way of debugging errors at user sites. It periodically checkpoints programs, and after a failure, it restarts the program from a checkpoint. Heavyweight instrumentation, exploration of alternatives (delta-debugging), and speculative execution may be used during replay. Some applications were successfully debugged using this approach, but the checkpoint may have to be far back in time to allow meaningful exploration.

The same tradeoff between instrumentation overhead and bug reproduction time has also been explored for debugging multithreaded programs. To faithfully replay the execution of a thread, the shared memory interactions with other threads need to be logged.

The cost of doing so is very high, and therefore the PRES [Park 2009] debugger selectively omits logging certain interactions, but requires multiple replay runs before it can recreate a path to a bug. Similarly, in order to avoid logging all shared memory accesses, ODR [Altekar 2009] allows some degree of inconsistency between the actual execution and the replay, provided that the inconsistencies do not affect the output of the program. The techniques used by PRES and ODR could be combined with partial logging of branches as presented in this paper.

Logging of branches has been used to report a program's behavior before a bug in Traceback [Ayers 2005]. The system uses this behavior to reconstruct the control flow leading to the problem. To reduce the instrumentation overhead, Traceback uses static analysis to minimize the number of instructions required to instrument branches, and only logs the most recent branches. Our system goes further by combining dynamic and static analyses to reduce the number of instrumented branches, and reproducing the entire path to the bug.

Symbolic execution for program testing was proposed as far back as 1976 [King 1976], but has recently received renewed attention as increased compute power and new optimizations offer opportunities to tackle the path explosion problem. Klee [Cadar 2008] has been used to symbolically execute the coreutils, and whitebox fuzzing [Godefroid 2008] has been used to allow programs with extensive input parsing to be executed symbolically. We present a new use

for symbolic execution, namely to discover which branches in a program are symbolic. In addition, we propose to speed up symbolic execution for bug reproduction by using a symbolic branch log collected at the user site.

8. Conclusion

In this paper, we consider the problem of instrumenting programs to reproduce bugs effectively, while keeping user data private. In particular, we focus on the tradeoff between the instrumentation overhead experienced by the user and the time it takes the developer to reproduce a bug in the program.

We explore this tradeoff by studying approaches for selecting a partial set of branches to log during the program's execution in the field. Specifically, we propose to use static analysis (dataflow and points-to analysis) and/or dynamic analysis (time-constrained symbolic execution) to find the branches that depend on input. Our instrumentation methods log only those branches to limit the instrumentation overhead. When a user encounters a bug, the developer uses the partial branch log to drive a symbolic execution engine in efficiently reproducing the bug.

Our results show that the instrumentation method that combines static and dynamic analyses strikes the best compromise between instrumentation overhead and bug reproduction time. For the programs we consider, this combined method reduces the instrumentation overhead by up to 92% (compared to using static analysis only), while limiting the bug reproduction time (compared to using dynamic analysis only).

We conclude that our characterization of this tradeoff and our combined instrumentation method represent important steps in improving bug reporting and optimizing symbolic execution for bug reproduction.

References

[Altekar 2009] Gautam Altekar and Ion Stoica. ODR: Output-deterministic Replay for Multicore Debugging. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 193–206, 2009.

[Ayers 2005] Andrew Ayers et al. TraceBack: First Fault Diagnosis by Reconstruction of Distributed Control Flow. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation*, pages 201–212, 2005.

[Cadar 2008] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating systems design and implementation*, pages 209–224, 2008.

[Castro 2008] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better Bug Reporting with Better Privacy. In *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural support for Programming Languages and Operating Systems*, pages 319–328, 2008.

[Crameri 2009] Olivier Crameri et al. Oasis: Concolic Execution Driven by Test Suites and Code Modifications. Technical report, EPFL, 2009.

[Dunlap 2002] George W. Dunlap et al. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[Glerum 2009] Kirk Glerum et al. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 103–116, 2009.

[Godefroid 2008] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing. *SIGPLAN Not.*, 43: 206–215, June 2008.

[Guo 2008] Zhenyu Guo et al. R2: an Application-level Kernel for Record and Replay. In *OSDI'08: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 193–208, 2008.

[King 1976] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

[Mosberger 1998] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. In *The First Workshop on Internet Server Performance*, pages 59—67, June 1998.

[Necula 2002] George C. Necula et al. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of Conference on Compilier Construction*, 2002.

[Pariag 2007] David Pariag et al. 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, 2007.

[Park 2009] Soyeon Park et al. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles*, pages 177–192, 2009.

[Sen 2005] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a Concolic Unit Testing Engine for C. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference*, pages 263–272, 2005.

[Tucek 2006] Joseph Tucek et al. Automatic On-line Failure Diagnosis at the End-user Site. In *HOTDEP'06: Proceedings of the 2nd Conference on Hot Topics in System Dependability*, pages 4–4, 2006.

[uClibc] uClibc. The uClibc Library, a C Library for Linux. http://www.uclibc.org/.

[Wang 2008] Rui Wang, XiaoFeng Wang, and Zhuowei Li. Panalyst: Privacy-aware Remote Error Analysis on Commodity software. In *SS'08: Proceedings of the 17th Conference on Security Symposium*, pages 291–306, 2008.

[Zamfir 2010] Cristian Zamfir and George Candea. Execution Synthesis: a Technique for Automated Software Debugging. In *EuroSys '10: Proceedings of the 5th European Conference on Computer Systems*, pages 321–334, 2010.