Resolving loop based path explosion during Symbolic Execution

Shehbaz Jaffer Ashvin Goel Angela Demke Brown
University of Toronto

Abstract

Symbolic execution is a widely used method for testing device drivers and achieving high code coverage. One problem in exploring code using symbolic execution is loop based path explosion. First, trying all paths using dynamic techniques does not guarantee additional code coverage. Second, large formulas to reach new branch using static analysis do not scale well due to constraint solver limitations.This paper describes *challenger deep*, a static analysis tool that precomputes a sequence of sub-paths within the loop body so that a specified branch of code can be reached. We use Z3 constraint solver for small queries. For larger queries where Z3 takes exceedingly large time, we switch to a genetic algorithm. Our tool is generic and can be used in both functional testing or whole program testing when a loop construct is encountered.

1. Introduction

Most modern Operating System code-bases are extremely large, and continue to grow with time. Increased code base leads to increased bugs, and increased need to write test cases to check unexpected behaviour. A major part of each Operating System source code are modules, and considerable research effort has been made to automate testing for the modules[9][10][5][1].

Although each of the techniques use practical approaches for majority of code, almost all approaches elude *deeper parts of code*, which can be reached only after a large path traversal. These large paths are typically introduced by loop constructs, which execute the same part of code multiple times potentially changing multiple variables in each iteration. We call such variables *loop dependent variables*, as their value depends on how many times the loop executed. Moreover, to generate an input that would hit a loop dependent variable requires the loop to execute a specific number of times. To make things worse, different branches within the loop may change the variable state differently, which makes deriving a particular sequence of sub-paths within a loop a combinatorial problem, which is NP Hard.

Genetic Algorithms are optimization algorithms that give approximate set of values that meet a particular constraint[12]. They are faster than classical algorithms that look for an exact solution. Instead of getting the correct answer, they focus on giving a set of *approximate* answers to a constraint problem. Genetic Algorithms have been extensively used to solve combinatorial problems like the Travelling Salesman Problem in the past to give promising solution. To reach branch conditions with loop dependent variables, we also *need not* find all possible solutions. Any possible sequence of paths within a loop that sets the loop branch condition to the required value would work.

This paper makes the following contributions: First, given a loop construct and a branch, we compute one of the inputs that would lead to execution of loop body in a way that the given loop dependent branch will get executed. Second, for cases where there are exponentially large number of possible inputs, we use genetic algorithm followed by greedy technique to reach from a relatively close sequence of executions to exact sequence of executions to enter a branch. For linear operations, we are able to generate solutions for upto 2x larger array sizes within a minute. Z3 solver, on the other hand, does not provide a solution after more than 5 minutes of operation. Genetic algorithms do not guarantee a solution, however, if a solution exists, we provide a faster technique to converge to a solution.

2. Overview

Operating Systems code is complicated. With each branch condition, a new state is created that either evaluates the branch condition to true or false. For n iterations in a loop that has a branch condition, there exist 2^n possible program execution paths. Table 1 shows the degree of complexity of some well known linux kernel modules. We observe excessive use of loop constructs in each module. loop dependent branches (column 6-7) can only be executed if specific number of loop iterations are done in order to evaluate a branch to true or false. It is unreasonable to expect traditional symbolic execution to guess the number of iterations required to

[Copyright notice will appear here once 'preprint' option is removed.]

module	sequential loop	nested loop	direct loop dependent variable	conditional loop dependent variable	direct loop dependent branch	conditional loop dependent branch
f2fs	55	152	337	441	148	167
ext4	78	288	370	1140	386	418
btrfs	147	754	778	2648	1136	1184
acpi	105	95	544	605	334	405
block	118	409	1438	1896	971	996
e1000	35	75	192	329	223	227
crypto	259	236	720	295	176	290

Table 1: Loop Dependent Complexity in Linux Drivers

reach each branch condition.

For the BTRFS code in Listing 1, we have a nested loop condition, where taking different sequence of paths within the C code leads to different values of the *slot_count* variable. We want to check if reaching line 15 is possible. Reaching this position is unsatisfiable, as the maximum value slot_count can reach is 24. slot_count should be greater than 171 to reach ENOMEM condition.

The total number of iterations involved are only 24. However, to understand the exponentiation problem further, At line 8, for each iteration, list_empty_block[c] may or may not be set. There are 2^6 ways in which list_empty_block can be set, and dynamic run over the loop would have to exhaustively run 24 * 2^6 times to only realise and report that line 15 cannot be reached. Our tool will precompute the value of slot_count required to reach line 15 statically before executing the loop, and determine how many iterations in what sequence may be required to reach the branch condition. Once the iterations required(171) and actual loop iterations (24) are given to the constraint solver, the constraint solver returns unsat value, after which we do not try to reach line 15, thus saving precious time for solving other paths and telling the scheduler not to make an attempt to reach line 15 any further.

```
1  btrfs_ioctl_space_info()
2  {
3      // num_types = 4; BTRFS_NR_RAID_TYPES = 6;
4      slot_count = 0;
5      for (i = 0 ; i < num_types ; i++)
6      {
7          for(c = 0 ; c < BTRFS_NR_RAID_TYPES; c++)
           {
8              if(list_empty_block[c]){
9                  slot_count++;
10             }
11         }
12     }   // sizeof(dest) is 24
13     alloc_size = sizeof(dest) * slot_count;
14     if(alloc_size > PAGE_SIZE)
15         return -ENOMEM;
16 }
```

Listing 1: btrfs code from btrfs/ioctl.c

2.1 Loop Types

Loops can classified into two broad types

1. **Fixed Iteration Loops(FIL)**. Having constant number of loop iterations.
2. **Input Dependent Loops(IDL)**. Iteration count depends on user or system input.

For Fixed iteration loops, computing the a sequence of operations to reach loop dependent branch condition is finite. However, for Input Dependent Loops, the possible sequences may be exponential. In such cases, we use k-bounding of loops, i.e. we only check if a branch may be reachable within k loop iterations.

2.2 Variable Types

To reach a branch containing a loop dependent variable, we classify variables that lie in a basic block dominated by a loop entry block as one of the following:

1. **loop invariant variable** variable whose value does not change with respect to loop iteration. we perform LLVM loop independent code motion (LICM) pass to move these variables outside the loop and do not check for branch conditions on these variables.
2. **direct loop dependent variable (dldv)** a variable that is assigned a value each time the loop executes.
3. **conditional loop dependent variable (cldv)** variable whose value is set inside the loop, but is guarded by a branch condition. This variable may / may-not be set each time the loop executes.
4. **direct loop dependent branch condition** A branch condition inside a basic block dominated by the loop entry block (may be inside or outside the loop body) having a dldv as one of the operators.
5. **conditional loop dependent branch condition** A branch condition inside a basic block dominated by the loop entry block having cldv as one of the operators. This may be inside or outside the loop body.

The objective of our tool is to predetermine what sequence of paths on each iteration should be taken inside the loop body to reach a direct or conditional loop dependent branch condition, without executing the loop body exponentially large number of times.

3. Architecture

Figure 1 shows the components of our tool. We do C++ analysis pass on LLVM bytecode. We use Python to write constraints that are either solved by Z3 constraint solver or genetic algorithm followed by a 0/1 Knapsack problem based on the size of constraints generated to reach a branch condition.

3.1 Components

1. **Loop Analyzer** Given a function, loopAnalyzer detects all sequences of basic blocks that form a path from function entry point to any termination basic block, executing only 1 sub-path within loop body. First, it traverses the CFG of the program without inspecting instructions inside the basic blocks. Next, It marks input variables (function arguments and return values from external calls). Further, it extracts loop information - start iteration value, branch condition, end iteration value, and iterator operations for Fixed Iteration Loops. This gives us total number of times loop executes. At the end of Loop Analysis, we generate paths which either skip loop body, or execute only 1 subpath within the loop. This ensures we traverse all paths within the loop body atleast once, and do not iterate over same path multiple times.

```
ldv = {} ldcv = {}
for each path in loopBody:
    for each BB in path:
        currentBB = path-> next
        for each I in currentBB
            if i = a
                if branchCondition
                    = false
                    ldv.add(i)
                else
                    ldcv.add(i)
            if i = branch
                branchCondition = true
            if i = a op b
                ( a , path ) =
                store ( op b)
        if currentBB -> predecessor > 1
        // branch end condition
            branchCondition = false
```

Listing 2: **Branch Analysis - Populates ldv and ldcv (loop dependent variable and loop dependent condition variable) maps with loop dependent and conditional loop dependent operands. if a branch condition is encountered(12), all subsiquent assignment operations(6) till branch end(17) are added to ldcv map. if assignment operation happens without a branch condition, variable is added to ldv map (8).**

We *expand* these paths later (Section 4) after computing what subpaths within loop would lead to a loop dependent branch condition.

2. **Branch Analyzer** We do data flow analysis of each loop skipped path and mark loop dependent variables (both direct and conditional). We explain data flow analysis in 2. For each variable that is assigned a value inside a loop, we mark the operation on the variable, associated operand, and loop path condition (in case the loop dependent variable is conditional). After this step, we know what operation occurs on which loop dependent variable along what path.
3. **LDV Expression Generator** any branch condition that has loop dependent variable operand is analyzed. these branches may be inside or outside the loop body. for each such branch, the conditions are written based on rules we describe in Section 4.
4. **Solve for Operations** with different operations known for a loop dependent variable, we generate one sequence of operations that would lead to loop dependent branch condition evaluating to true.
5. **Create Extended Path** from operation sequence in previous step, we know which sub-paths in loop body should be executed in what sequence to reach a loop dependent branch condition. we extend loop body with the sequence of subpaths in previous step to generate extended path which would lead to target branch condition.
6. **Get Input Value** Once the extended path is derived, we solve path constraints associated with this path to generate the function arguments that would lead to loop dependent branch condition execution.

3.2 Algorithm

Our approach to use either z3 constraint solver or genetic Algorithm is simple. We check if z3 based constraint is satisfiable. If the constraint is not satisfiable, we return unsat and do not invoke the genetic Algorithm. If the path is satisfiable, we check if z3 would be able to provide us with a solution within a predefined time. If Z3 times out, we try solving the constraints using Genetic Algorithm. We discuss our solution metric further in Section 4

```
if z3.unsat:
    return unsat
else:
    z3.solve() // timeout
    geneticAlgorithm()
```

4. Experiments

Our implementation works on LLVM bytecode and does function level analysis. we do not handle recursions. we do not handle paths that span multiple function calls. A return value from an external function call is treated as symbolic. We classify different types of conditional loop dependent variables based on the operations they undergo inside a loop

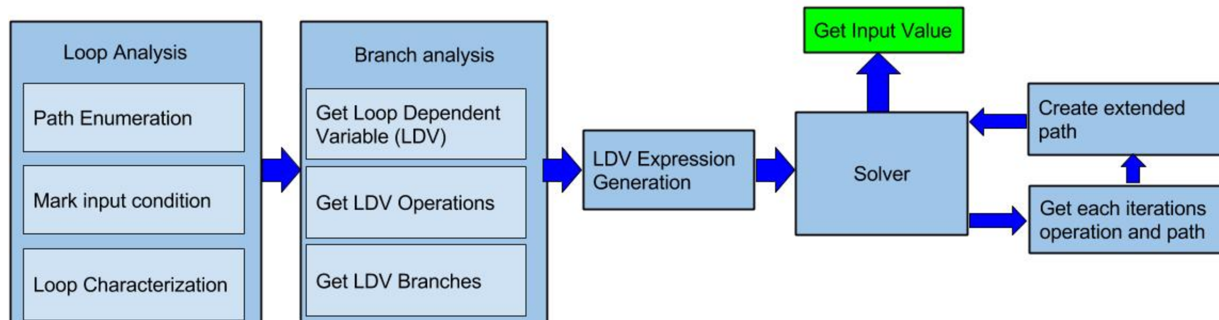

Figure 1: Challenger Deep Architecture

block. Our current system handles three types of loop dependent variables - constant operation, single operation and two operation variables.

4.1 Z3 Solver Expression Generation

Code Listing 3, 5 and 7 show three types of loop dependent variable operations

1. conditional loop dependent branch with constant operation

```
1  count = 0;
2  constantOp(int input[100])
3  for(i = 0 ; i < 100 ; i++)
4  {
5
6      if(input[i] == 'B'){
7          count += 1;
8      }
9  }
10 if(count == 75){
11     bug();
12 }
```

Listing 3: **conditional loop dependent branch with 1 iterator dependent operation**

```
1  noop = 0 // skip if
2  op1 = 1 // enter if
3  0 <= ite0 <= 1
4  ...
5  0 <= ite99 <= 1
6  op1 * ite0 + op1 * ite1 + ...
7  + op1 * ite99 = 75
8  ite(ite0 == 1 ? input[0] == 'B',
9      input[0] != 'B')
10 ite(ite1 == 1 ? input[1] == 'B',
11     input[1] != 'B')
12 ...
13 ite(ite99 == 1 ? input[99] == 'B',
14     input[99] != 'B')
```

Listing 4: **z3 Constraints for reaching if condition**

Listing 3 has count variable as conditional loop dependent branch variable. There are two operations that occur on count, increment by constant 1 if line 6 is true, or no operation if it is false. In order to reach line 7, we need *any* 75 if conditions to evaluate to true, as the operation is constant. Solving the z3 constraints gives us an array of input[0] to input[99] values that would lead to the sum 75.

2. conditional loop dependent branch with only 1 iterator dependent operation (increment by iterator)

```
1  constantOp(int input[100])
2  count = 0;
3  for(i = 0 ; i < 100 ; i++)
4  {
5      if(input[i] == 'B'){
6          count += i;
7      }
8  }
9  if(count == 75){
10     bug();
11 }
```

Listing 5: **conditional loop dependent branch with an iterator dependent operation**

The resultant z3 expression for this equation is

```
1  noop = 0 // enter if
2  op1 = 1 // skip if
3  op2 = 2
4  ...
5  op99 = 99
6  0 <= ite0 <= 1
7  ...
8  0 <= ite99 <= 1
9  op1 * ite0 + op2 * ite1 + ...
10 + op99 * ite99 = 75
11 ite(ite0 == 1 ? input[0] == 'B',
12     input[0] != 'B')
13 ite(ite1 == 1 ? input[1] == 'B',
14     input[1] != 'B')
15 ...
16 ite(ite99 == 1 ? input[99] == 'B',
17     input[99] != 'B')
```

Listing 6: **z3 Constraints for reaching if condition**

The difference here is that the operations have increased from single constrant increment in the previous example to 99 different operations. this problem is a combinatorial problem, and takes exponentially large time to solve.

3. conditional loop dependent branch with two iterator dependent operations (increment and decrement by iterator)

Finally, in the example with 2 operations, we increase operation set from op1 to op100 to op1 to op200, with each iterator operation taking either of the positive or the negative value in each iteration. other conditions are extended appropriately. we do not show the expression due to space limitations.

```
constantOp(int input[100])
count = 0;
for(i = 0 ; i < 100 ; i++)
{
    if(input[i] == 'B'){
        count += i;
    }else{
        count -= i;
    }
}
if(count == 75){
    bug();
}
```

Listing 7: **conditional loop dependent branch with two iterator dependent operations**

4.2 Genetic Algorithm

genetic algorithm is an optimization technique widely used to get near-perfect solutions to a combinatorial problem. At every instance, genetic algorithm produces generations or a set of solutions. with each iteration or pass, the algorithm aims to bring some set of solutions closer to the actual solution.

We counter path exponentiation with exponential solutions first we generate a population of possible solutions. each solution should adhere to certain rules; for example in the loop path sequence generation for increment by iterator operation, at any index i of input[100], either 0 or i value can exist. after generating a random sequence of such candidate arrays, which we call *chromosomes*, we swap different potential solutions by interchanging values at random indexes. This process is called cross over. Further, we induce mutation flipping random position values within each chromosome, such that value at any random index i changes from i to 0 or 0 to i. We do each of the above operations for userDefinedCount times. Each set of such operations form a generation.

```
createPopulation()
for each generation;
do
    parentSelection()
    crossOver()
    mutation()
    fitnessCalculation()
    aging()
while generationCount > userDefinedCount
```

Listing 1: **Genetic Algorithm**

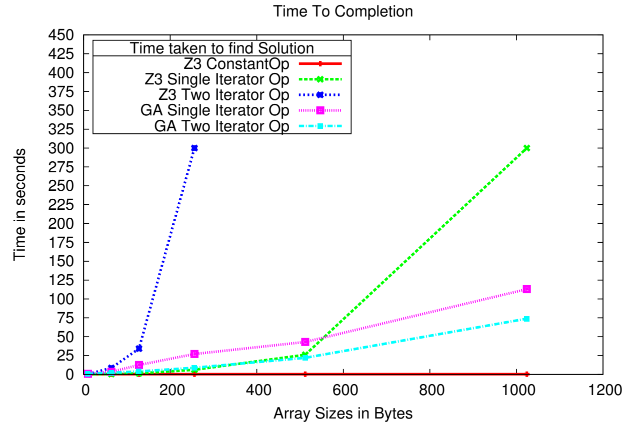

Figure 2: time taken to find solution of array sum problem. Genetic Algorithm scales linearly whereas Z3 solver does not provide results for array sizes ¿ 256 elements for upto 300 seconds.

At the end of the genetic algorithm, we have close-to-required input array that would lead us to the if condition. We define fitness of a solution as the degree to measure how close the solution is to the actual solution. for instance, if we get an array whose sum of values is 97, the fitness of the array would be 97 - 75 = 22. Our objective is to minimize this value.

Once a set of close enough solutions is retrieved, we perform greedy 0/1 knapsack for limited number of times on 10 top solutions to reach the actual solution. This has higher chance of convergence when the arrays are large, since we have a larger set of values to choose from. Our results show that for smaller values, z3 performs better and genetic algorithm fails to converge for some inputs. However for larger data set, genetic algorithm with greedy knapsack provides us with solutions within seconds where z3 computation goes into minutes without providing us a solution.

5. Results

Although previous work shows how direct loop dependent branches [8][14] and conditional loop branches with constant operations[1] can be resolved with comparable efficiency, we show how genetic algorithm in conjunction with Knapsack problem outperforms z3 solver for loops having conditional statements.

5.1 Size of solution space

We test the three forms of expression described in code listing 3, 5 and 7 using both Z3 Constraint solver and Genetic Algorithm. Figure 2 shows how Genetic algorithm scales for larger array input sizes. Z3 times out (more than 300 seconds) for array sizes over 256 bytes. Z3 However, is able to report unsatisfiable formulas within few seconds. This is because internally, Z3 first negates the set of constraints and

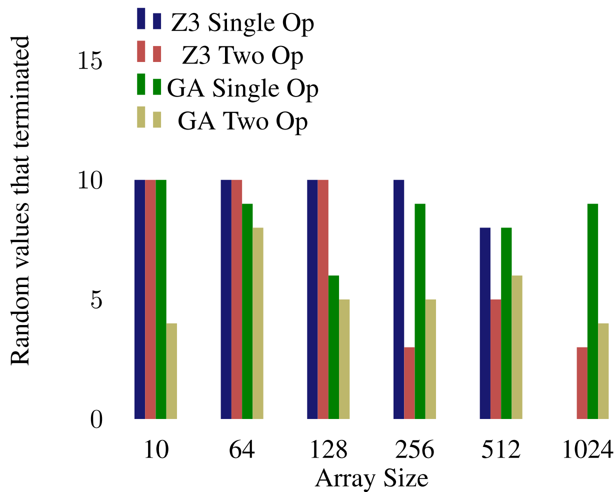

Figure 3: 10 Random values between minimum and maximum sums possible were selected. For larger arrays, Z3 solver fails to compute satisfiable queries. Genetic algorithm provides solutions even for larger array sizes which have a solution.

checks if any solution exists for the negated path constraint. If it finds that a solution is possible, it marks the original Path Constraint unsat. For satisfiable queries, however, it takes computationally large time.
Genetic algorithm provides solution for few inputs for upto 4x large array sizes. The rate of convergence of the algorithm, however, decreases for larger array sizes. Since Genetic algorithm does not guarantee convergence even on unsat problems, we first query the Z3 solver for satisfiability before calling genetic algorithm.

5.2 Convergence

We ran the Z3 and Our genetic algorithm programs for 10 random input variables between maximum and minimum values that sum of variables may assume. Blue and Red bars show the number of random input variables out of 10 that converged within 5 minutes of run. We see that none of the 10 random input variables converged for Z3 for 1024 array size. The red bar corresponding to two operations that converged were unsatisfiable values. Hence Z3 only converges for unsatisfiable values for large inputs.
Now, we look at Genetic Algorithm and how it converges (Yellow and Green). We note that for smaller values, even though Z3 was able to converge, Genetic Algorithm continued to run without converging. Hence for small array sizes, using Z3 constraint solver is advisable. For larger array sizes, Z3 fails to converge for satisfiable queries.

6. Related Work

Automated test case generation to achieve high code coverage has been explored in the past [7][15][4]. Further, symbolic execution has been used earlier to test device driver code[9][13][10][5]. Code coverage for programs containing loop constructs using symbolic execution is challenging[14][8]. For covering branches dependent on loop iterations, we have to execute the loop body along specific paths specific number of times. Prior techniques to resolve loop dependent branch conditions only handle specific type of loop variable dependencies. Partial loop summarisation [8] techniques only work where induction variables have linear relationship with iterator. conditional relationships are not handled. loop extended symbolic execution[14] marks iterator variable symbolic and expresses loop dependent variable as a function of the iterator. The grammar rules to derive a generic formula however, assumes only direct loop dependent variables.
Veritesting[1] takes linear time to construct formulas for loops - as it unrolls loop dynamically over and over again, until loop completes execution or a timeout is reached. Our approach can list the formula in only 1 pass of the loop. For unbounded loop iterations and non-constant loop variables, both approaches will try until timeout occurs. Static Symbolic Execution Techniques [2] have explored loop summarization by summarising the entire loop within 1 large formula. However, the resultant queries are often very large to be solved by the constraint solver.
Comparision with SAGE[7] shows a generational search strategy that generates multiple paths after negating constraints along a code execution path. SAGE aims at exploring large part of code, but does not have a fitness function to direct its search. The heuristics used by SAGE are code coverage, which would try to cover multiple ways to reach unexplored code **without getting feedback from previous executions**. By using a fitness function, we constantly remove and add new paths that have been tried and failed to converge to reach a particular branch. We look for *any* sequence of operations that would lead to coverage of target branch.
Using Genetic Algorithms to generate tests has also been explored in the past[3][6], However, using these techniques for every equation need not guarantee us a solution. A systematic use of Genetic Algorithm with knapsack problem at places where classic combinatorial approaches such as those used by Z3 fail helps achieve higher code coverage.

7. Conclusion and Future Work

A static analysis tool to precompute a sequence of branches for each iteration in the loop body to reach a loop dependent branch is proposed. The scalability problems associated with Z3 constraint solver are shown and use of genetic algorithm to solve such problems is proposed. Our tool is generic and aids existing symbolic execution tools by resolving loop conditions instead of undirected or random loop exploration. More work is required to handle other types of operations on loop dependent variables.

References

[1] AVGERINOS, T., REBERT, A., CHA, S. K., AND BRUMLEY, D. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 1083–1094.

[2] BABIC, D., AND HU, A. J. Calysto: Scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 211–220.

[3] BALUDA, M. Evose: Evolutionary symbolic execution. In *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation* (New York, NY, USA, 2015), A-TEST 2015, ACM, pp. 16–19.

[4] CADAR, C., DUNBAR, D., AND ENGLER, D. 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* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[5] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2e: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not. 47*, 4 (Mar. 2011), 265–278.

[6] GALEOTTI, J. P., FRASER, G., AND ARCURI, A. Improving search-based test suite generation with dynamic symbolic execution. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)* (Nov 2013), pp. 360–369.

[7] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Sage: Whitebox fuzzing for security testing. *Queue 10*, 1 (Jan. 2012), 20:20–20:27.

[8] GODEFROID, P., AND LUCHAUP, D. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 23–33.

[9] KADAV, A., AND SWIFT, M. M. Understanding modern device drivers. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 87–98.

[10] KUZNETSOV, V., CHIPOUNOV, V., AND CANDEA, G. Testing closed-source binary device drivers with ddt. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 12–12.

[11] LARSON, P., HINDS, N., RAVINDRAN, R., AND FRANKE, H. Improving the linux test project with kernel code coverage analysis.

[12] MITCHELL, M. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[13] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. Symdrive: Testing drivers without devices. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 279–292.

[14] SAXENA, P., POOSANKAM, P., MCCAMANT, S., AND SONG, D. Loop-extended symbolic execution on binary programs. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (New York, NY, USA, 2009), ISSTA '09, ACM, pp. 225–236.

[15] SEN, K., MARINOV, D., AND AGHA, G. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2005), ESEC/FSE-13, ACM, pp. 263–272.