Bucketing Failing Tests via Symbolic Analysis

Van-Thuan Pham[1], Sakaar Khurana[2], Subhajit Roy[2], Abhik Roychoudhury[1]

[1]National University of Singapore, Singapore
[2]Indian Institute of Technology Kanpur, India
{thuanpv,abhik}@comp.nus.edu.sg sakaark@gmail.com subhajit@iitk.ac.in

Abstract. A common problem encountered while debugging programs is the overwhelming number of test cases generated by automated test generation tools, where many of the tests are likely to fail due to same bug. Some coarse-grained clustering techniques based on point of failure (PFB) and stack hash (CSB) have been proposed to address the problem. In this work, we propose a new symbolic analysis-based clustering algorithm that uses the *semantic reason* behind failures to group failing tests into more "meaningful" clusters. We implement our algorithm within the KLEE symbolic execution engine; our experiments on 21 programs drawn from multiple benchmark-suites show that our technique is effective at producing more fine grained clusters as compared to the FSB and CSB clustering schemes. As a side-effect, our technique also provides a semantic characterization of the fault represented by each cluster—a precious hint to guide debugging. A user study conducted among senior undergraduates and masters students further confirms the utility of our test clustering method.

1 Introduction

Software debugging is a time consuming activity. Several studies [4], [6], [8], [9], [18] have proposed clustering techniques for failing tests and proven their effectiveness in large-scale real-world software products. The Windows Error Reporting System (WER) [8] and its improvements such as ReBucket [6] try to arrange error reports into various "buckets" or clusters. WER employs a host of heuristics involving module names, function offset and other attributes. The Rebucket approach (proposed as an improvement to WER) uses specific attributes such as the call stack in an error report.

Although the techniques have been applied widely in industry, there are three common problems that they can suffer from (as mentioned in [8]). The first problem is *"over-condensing"* in which the failing tests caused by multiple bugs are placed into a single bucket. The second problem is *"second bucket"* in which failing tests caused by one bug are clustered into different buckets. The third one, *"long tail"* problem, happens if there are many small size buckets with just one or a few tests. For example, using failure type and location (as used in KLEE [4]) for clustering tests are more likely to suffer from both *over-condensing* and *second bucket* problems as they would group all tests that fail at the same location, completely insensitive to the branch sequence and the call-chain leading to the error. Call stack similarity for clustering tests also suffers from the "over-condensing" and "second bucket" problems because it is insensitive to the intraprocedural program paths (i.e. the conditional statements within functions). One of the main reasons why techniques in [4], [6], [8], [9], [18] suffer from these problems is that they do not take program *semantics* into account.

In this work, we propose a novel technique to cluster failing tests via symbolic analysis. Unlike previous work that drive bucketing directly from error reports, we adapt symbolic path exploration techniques (like KLEE [4]) to cluster (or bucket) the failing tests on-the-fly. We drive bucketing in a manner such that tests in each group fail due to the same *reason*. Since we use symbolic analysis for clustering, our technique leads to more accurate bucketing; that is (a) tests for two different bugs are less likely to appear in the same bucket, and (b) tests showing the same bug are less likely to appear in different buckets. We experimentally evaluate our semantics-based bucketing technique on a set of 21 programs drawn from five repositories: IntroClass, Coreutils, SIR, BugBench and exploit-db. Our results demonstrate that our symbolic analysis based bucketing technique is effective at clustering tests: for instance, the **ptx** program (in our set of benchmarks) generated `3095` failing tests which were grouped into `3` clusters by our technique. Similarly, our tool clustered `4510` failing tests of the **paste** program into `3` clusters.

In addition to bucketing failures, our tool provides a *semantic characterization* of the reason of failure for the failures in each cluster. This characterization can assist the developers better understand the nature of the failures and, thus, guide their debugging efforts. The existing approaches are not capable of defining such an accurate charaterization of their clusters (other than saying that all tests fail at a certain location or with a certain stack configuration).

While our algorithm is capable of bucketing tests as they are generated via a symbolic execution engine, it is also capable of clustering failures in existing test-suites by a post-mortem analysis on the set of failures.

The contributions of this paper are as follows:

- We propose an algorithm to efficiently cluster failing test cases, both for the tests generated automatically by symbolic execution as well as tests available in existing test-suites. Our algorithm is based on deriving a *culprit* for a failure by comparing the failing path to the nearest correct path. As we use semantic information from the program to drive our bucketing, we are also able to derive a *characterization* of the reason of failure of the tests grouped in a cluster. The existing approaches are not capable of defining such characterization for the clusters they produce.
- We implement a prototype of the clustering approach on top of the symbolic execution engine KLEE [4]. Our experiments on 21 programs show that our approach is effective at producing more meaningful clusters as compared to existing solutions like the point of failure and stack hash based clustering.

2 Overview

We illustrate our technique using a motivating example in Figure 1. In the `main()` function, the code at line 27 manages to calculate the value of $(2^x +$

$x! + \sum_{i=0}^{y} i)$ in which x and y are non-negative integers. It calls three functions, `power()`, `factorial()` and `sum()`, to calulate 2^x, $x!$ and sum of all integer numbers from 0 to y. While `sum()` is a correct implementation, both `power()` and `factorial()` are buggy.

In the `power()` function, the programmer attempts an optimization of saving a multiplication: she initializes the result (the integer variable `pow()`) to `2` (line 2) and skips the multiplication at line 5 if n equals `1`. However, the optimization does not handle the special case in which n is zero. When n is zero, the loop is not entered and the function returns `2`: it is a wrong value since 2^0 must be 1. Meanwhile, in the `factorial()` function the programmer uses a wrong condition for executing the loop at line 13. The correct condition should be $i \leq n$ instead of $i < n$. The incorrect loop condition causes the function to compute factorial of $n - 1$ so the output of the function will be wrong if $n \geq 2$.

```
 1  unsigned int power(unsigned int n) {
 2      unsigned int i, pow = 2;
 3  /* Missing code: if (n == 0) return 1; */
 4      for(i=1; i<=n; i++) {
 5          if(i==1) continue;
 6          pow = 2*pow;
 7      }
 8      return pow;
 9  }
10  unsigned int factorial(unsigned int n) {
11      unsigned int i,result = 1;
12  /* Incorrect operator: < should be <= */
13      for(i=1;i<n;i++)
14          result = result*i;
15      return result;
16  }
17  unsigned int sum(unsigned int n) {
18      unsigned int result = 0, i;
19      for (i=0; i<=n; i++)
20         result += i;
21      return result;
22  }
23  int main() {
24      unsigned int x, y, val, val_golden;
25      make_symbolic(x, y);
26      assume(x<=2 && y<=2);
27      val = power(x)+factorial(x)+sum(y);
28      assert(val == golden_output(x, y));
29      return 0;
30  }
```

Fig. 1: Motivating Example

We can use a symbolic execution engine (like KLEE) to generate test cases that expose the bugs. In order to do that, we first mark the variables x and y as symbolic (line 25) and add an assert statement at line 28. The assertion is used to check whether the calculated value for $2^x + x! + \sum_{i=0}^{y}$ (as stored in *val*) is different from the expected value which is fetched from `golden_output()`.

The specification *oracle* `golden_output()` can be interpreted in many ways depending on the debugging task: for example, it can be the previous version of the implementation when debugging regression errors, or the expected result of each test when run over a test-suite. For the sake of simplicity, we add an `assume()` statements at line 26 to bound the values of symbolic variables x and y.

Figure 2 shows the symbolic execution tree that KLEE would explore when provided with this example. In this paper, we use the term **failing path** to indicate program paths that terminate in error. The error can be assertion violation or run-time error detected by symbolic execution engine such as divide-by-zero or memory access violation (as supported in KLEE). In contrast, the term **pass-**

Table 1: Clustering result: Symbolic analysis

Path ID	Test Case	Path Condition	Culprit Constraint	Clus. ID
1	x=0, y=0	$(0 \le x, y \le 2) \wedge (\boldsymbol{x < 1}) \wedge (y \le 0)$	$(x < 1)$	1
2	x=0, y=1	$(0 \le x, y \le 2) \wedge (\boldsymbol{x < 1}) \wedge (y > 0) \wedge (y \le 1)$	$(x < 1)$	1
3	x=0, y=2	$(0 \le x, y \le 2) \wedge (\boldsymbol{x < 1}) \wedge (y > 0) \wedge (y > 1) \wedge (y \le 2)$	$(x < 1)$	1
4	x=1, y=0	$(0 \le x, y \le 2) \wedge (\boldsymbol{x \ge 1}) \wedge (\boldsymbol{x < 2}) \wedge (y \le 0)$	NA	NA
5	x=1, y=1	$(0 \le x, y \le 2) \wedge (\boldsymbol{x \ge 1}) \wedge (\boldsymbol{x < 2}) \wedge (y > 0) \wedge (y \le 1)$	NA	NA
6	x=1, y=2	$(0 \le x, y \le 2) \wedge (\boldsymbol{x \ge 1}) \wedge (\boldsymbol{x < 2}) \wedge (y > 0) \wedge (y > 1) \wedge (y \le 2)$	NA	NA
7	x=2, y=0	$(0 \le x, y \le 2) \wedge (x \ge 1) \wedge (\boldsymbol{x \ge 2}) \wedge (y \le 0)$	$(x \ge 2)$	2
8	x=2, y=1	$(0 \le x, y \le 2) \wedge (x \ge 1) \wedge (\boldsymbol{x \ge 2}) \wedge (y > 0) \wedge (y \le 1)$	$(x \ge 2)$	2
9	x=2, y=2	$(0 \le x, y \le 2) \wedge (x \ge 1) \wedge (\boldsymbol{x \ge 2}) \wedge (y > 0) \wedge (y > 1) \wedge (y \le 2)$	$(x \ge 2)$	2

ing path indicates paths that successfully reach the end of the program (or the *return* statement in the intraprocedural setting) with no errors.

As shown in Figure 2, KLEE explores 9 feasible executions and detects 6 failing paths; the paths are labeled from 1 to 9 in the order tests are generated while following the Depth-First-Search (DFS) search strategy. If we apply failure location based or call-stack based bucketing techniques, *both of them will place all 6 failing tests in a single cluster* as there is only one failure location at line 28, and the call stacks are identical when the failure is triggered. Hence, both the techniques suffer from the "over-condensing" problem as the failures are due to two different bugs (in the `power()` and `factorial()` functions).

Let us now present our approach informally: given a failing test t encountered during symbolic exploration, our algorithm compares the path condition of t with the path condition of a successful test t' that has the *longest common prefix* with t. The branch b at which the execution of t and t' differ is identified as the *culprit branch* and the branch condition at b which leads to the failing path is identified as the *culprit constraint*—the "reason" behind the failure of t. Intuitively, the reason behind blaming this branch for the failure is that the failing path t could have run into the passing execution t'—only if this branch b had not misbehaved!

Table 1 presents the result produced by our clustering algorithm (refer to Figure 2 for the symbolic execution tree). The failing tests 1-3 fail due to the bug in the `power()` function. The *culprit constraint* or "reason" for these failures is attributed as $x < 1$, since it is the condition on the branch where these failing tests diverge from their nearest passing test (Test 4), after sharing the *longest common prefix* $((0 \le x \le 2) \wedge (0 \le y \le 2))$. Hence, we create the first cluster (Cluster 1) and place tests 1-3 in it, with the characterization of the cluster as $(x < 1)$. Similarly, the failing tests 7-9 (failing due to the bug in `factorial()`) share the longest common prefix $((0 \le x \le 2) \wedge (0 \le y \le 2) \wedge (x \ge 1))$ with Test 4; thus, the culprit constraint for tests 7-9 is inferred as $(x \ge 2)$. Hence, these tests are placed in Cluster 2 with the *characterization* $(x \ge 2)$. Note that the culprit constraints $(x < 1)$ and $(x \ge 2)$ form a neat *semantic characterization* of the failures in these two clusters.

Summary. In this example, our semantic-based bucketing approach correctly places 6 failing tests into 2 different clusters. Unlike the two compared tech-

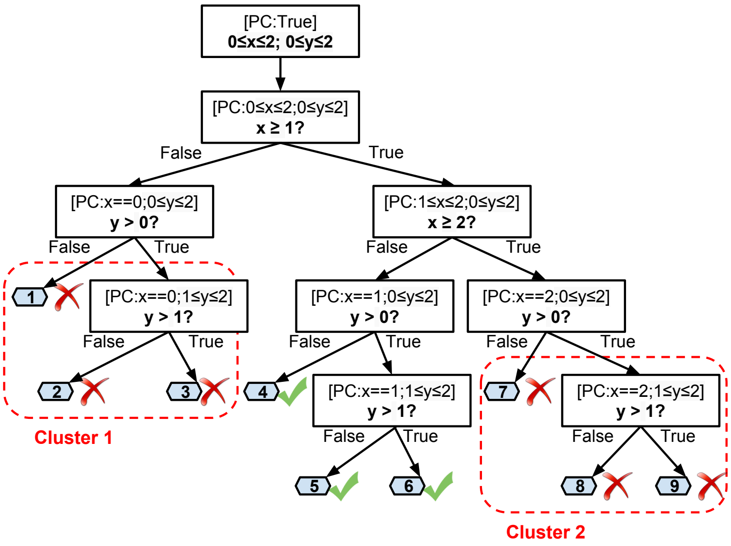

Fig. 2: Symbolic execution tree for motivating example

niques, it does not suffer from the "over-condensing" problem, and therefore, yields a more *meaningful* clustering of failures. Moreover, we provide a *semantic characterization* for each cluster that can assist developers in their debugging efforts. In fact, the characterization for `Cluster1` ($x < 1$) exactly points out the bug in `power()` (as x is non-negative integer, $x < 1$ essentially implies that x equals zero). Likewise, the characterization for `Cluster2` ($x \geq 2$) hints the developer to the wrong loop condition in the `factorial()` function (as the loop is only entered for $x \geq 2$). We, however, emphasize that our primary objective is **not** to provide root-causes for bugs, but rather to enable a good bucketing of failures.

3 Reasons of failure

The *path condition* ψ_p of a program path p is a logical formula that captures the set of inputs that exercise the path p; i.e. ψ_p is true for a test input t if and only if t exercises p. We say that a path p is *feasible* if its path condition ψ_p is satisfiable; otherwise p is *infeasible*.

We record the *path condition* ψ_p for a path p as a list of conjuncts l_p. Hence, the *size* of a path condition ($|\psi_p|$) is simply the cardinality of the list l_p. We also assume that as symbolic execution progresses, the branch constraints (encountered during the symbolic execution) are recorded in the path condition *in order*. This enables us to define *prefix(i,* ψ_p*)* as the prefix of length i of the list l_p that represents the path condition ψ_p. Hence, when we say that two paths p and q have a *common prefix* of length i, it means that *prefix(i,* ψ_p*)* = *prefix(i,* ψ_q*)*.

Definition 1 (*Culprit Constraint*)

Given a failing path π_f *with a path condition* ψ_f *(as a conjunct* $b_1 \wedge b_2 \wedge \cdots \wedge b_i \wedge \ldots b_n$*) and an exhaustive set of all feasible passing paths* Π*, we* *attribute b_i (the i-th constraint where i ranges from 1 to n) as the culprit constraint if and only if $i - 1$ is the maximum value of j $(0 \leq j < n)$ such that $prefix(j, \psi_f) = prefix(j, \psi_p)$ among all passing paths $p \in \Pi$.*

We use the *culprit constraint* (as a symbolic expression) as the reason why the error path "missed" out on following the passing path; in other words, the failing path could have run into a passing path, only if the branch corresponding to the culprit constraint had not *misbehaved*.

Our heuristic of choosing the culprit constraint in the manner described above is primarily designed to achieve the following objectives:

- **Minimum change to Symbolic Execution Tree**: Our technique targets well-tested production-quality programs that are "almost" correct; so, our heuristic of choosing the latest possible branch as the "culprit" essentially tries to capture the intuition that the symbolic execution tree of the *correct* program must be similar to the symbolic execution tree of the faulty program. *Choosing the latest such branch as the culprit is a greedy attempt at encouraging the developer to find a fix that makes the minimum change to the current symbolic execution tree of the program.*
- **Handle "burst" faults**: In Figure 2, all paths on one side of the node with $[PC : 1 \leq x \leq 2; 0 \leq y \leq 2]$ fail. So, the branching predicate for this node, $x \geq 2$, looks "suspicious". Our heuristic of identifying the latest branch as the culprit is directed at handling such scenarios of "burst" failures on one side of a branch.

4 Clustering framework

4.1 Clustering algorithm

Algorithm 1 shows the core steps in dynamic symbolic execution with additional statements (highlighted in grey) for driving test clustering. The algorithm operates on a representative imperative language with assignments, assertions and conditional jumps (adapted from [3], [13]). A symbolic executor maintains a state (l, pc, s) where l is the address of the current instruction, pc is the path condition, and s is a symbolic store that maps each variable to either a concrete value or an expression over input variables. At line 3, the algorithm initializes the worklist with an initial state pointing to the start of the program $(l_0, true, \emptyset)$: the first instruction is at l_0, the path condition is initialized as *true* and the initial store map is empty.

The symbolic execution runs in a loop until the worklist W becomes empty. In each iteration, based on a search heuristic, a state is picked for execution (line 7). Note that to support failing test bucketing, the search strategy must be DFS or an instance of our clustering-aware strategy (*clustering-aware search strategy* discussed in Section 4.2). A worklist S (initialized as empty) keeps all the states created/forked during symbolic exploration.

If the current instruction is an assignment instruction, the symbolic store s is updated and a new state pointing to the next instruction is inserted into S (lines $8 - 9$). A conditional branch instruction is processed (line 10) via a constraint solver that checks the satisfiability of the branch condition; if both its

Algorithm 1 Symbolic Exploration with Test Clustering

```
procedure SymbolicExploration($l_0$, $W$)
    $C \leftarrow \{\}$; $passList \leftarrow [\,]$; $failList \leftarrow [\,]$        ▷ initialization for bucketing
    $W \leftarrow \{(l_0, true, \emptyset)\}$                                         ▷ initial worklist
    while $W \neq \emptyset$ do
        $(l, pc, s) \leftarrow pickNext(W)$
        $S \leftarrow \emptyset$
        switch $instrAt(l)$ do                                                        ▷ execute instruction
            case $v := e$                                                             ▷ assignment instruction
                $S \leftarrow \{(succ(l), pc, s[v \rightarrow eval(s, e)])\}$
            case $if\ (e)\ goto\ l'$                                                  ▷ branch instruction
                $e \leftarrow eval(s, e)$
                if $(isSat(pc \wedge e) \wedge isSat(pc \wedge \neg e))$ then
                    $S \leftarrow \{(l', pc \wedge e, s), (succ(l), pc \wedge \neg e, s)\}$
                else if $(isSat(pc \wedge e)$ then
                    $S \leftarrow \{(l', pc \wedge e, s)\}$
                else
                    $S \leftarrow \{(succ(l), pc \wedge \neg e, s)\}$
                end if
            case assert(e)                                                            ▷ assertion
                $e \leftarrow eval(s, e)$
                if $(isSat(pc \wedge \neg e))$ then
                    $testID \leftarrow$ GenerateTest(l, $pc \wedge \neg e$, s)
                    $pc' \leftarrow ConvertPC(pc \wedge \neg e)$
                    AddToList(failList,(testID,$pc'$))
                    continue
                else
                    $S \leftarrow \{(succ(l), pc \wedge e, s)\}$
                end if
            case halt                                                                 ▷ end of path
                $testID \leftarrow$ GenerateTest(l,pc,s)
                $pc' \leftarrow ConvertPC(pc)$
                AddToList(passList,(testID,$pc'$))
                if $failList \neq [\,]$ then
                    ClusterTests(C,passList,failList)
                    $failList \leftarrow [\,]$                                        ▷ empty failing list
                end if
                continue
        $W \leftarrow W \cup S$                                                       ▷ update worklist
    end while
    if $failList \neq [\,]$ then
        ClusterTests(C,passList,failList)
    end if
end procedure
```

branches are satisfiable, two new states are created and inserted into S. If only one of the branches is satisfiable, the respective state is added to S. For *assert* instructions, the symbolic execution checks the assert condition, and if it holds, a new program state is created and the state is added to S. If the condition does not hold, it triggers an assertion failure, thereby, generating a failing test case (we call the respective test case a "**failing test**"). Some symbolic execution engines (like KLEE [4]) perform run-time checks to detect failures like divide-by-zero and memory access violations; in this algorithm, the *assert* instruction is used to represent the failures detected by such checks as well. On encountering a halt instruction, the symbolic execution engine generates a test-case for the path (we refer to such a test case as a "**passing test**"). The *halt* instruction represents a normal termination of the program.

Algorithm 2 Clustering failing tests

```
 1: procedure ClusterTests(Clusters,passList,failList)
 2:     for (failID, failPC) ∈ failList do
 3:         maxPrefixLength ← 0
 4:         for (passID, passPC) ∈ passList do
 5:             curPrefixLength ← LCP(failPC, passPC)
 6:             if curPrefixLength > maxPrefixLength then
 7:                 maxPrefixLength ← curPrefixLength
 8:             end if
 9:         end for
10:         reason ← failPC[maxPrefixLength+1]
11:         Update(Clusters,failID,reason)
12:     end for
13: end procedure
14: ————————————————————————————————
15: procedure Update(Clusters,failID,reason)
16:     for r ∈ Clusters.Reasons do
17:         if isValid(reason ⇒ r) then
18:             Clusters[r].Add(failID)
19:             return
20:         else if isValid(r ⇒ reason) then
21:             UpdateReason(Clusters[r], reason)
22:             Clusters[reason].Add(failID)
23:             return
24:         end if
25:     end for
26:     AddCluster(Clusters, reason, failID)
27: end procedure
```

To support clustering of tests, we define two new variables, *passList* and *failList*, to store information about all explored passing and failing tests (respectively). For each test, we keep a pair (*testID*, *pathCondition*), where *testID* is the identifier of the test generated by symbolic execution, and *pathCondition* is a list of branch conditions (explained in Section 3). We also introduce a variable C that keeps track of all clusters generated so far; C is a map from a culprit constraint (cluster reason) to a list of identifiers of failing tests. The bucketing functionality operates in two phases:

Phase 1: Searching for failing and passing tests. The selected search strategy guides the symbolic execution engine through several program paths, generating test cases when a path is terminated. We handle the cases where tests are generated, and update the respective list (*passList* or *failList*) accordingly. In particular, when a failing test case is generated, the path condition (*pc*) is converted to a list of branch conditions (*pc'*). The pair comprising of the list *pc'* and the identifier of the failing test case form a representation of the failing path; the pair is recorded in *failList* (lines 23–24). The *passList* is handled in a similar manner (lines 31–32).

Phase 2: Clustering discovered failing tests. Once a passing test is found (lines 35–37) or the symbolic execution engine completes its exploration (lines 42–43), the clustering function *ClusterTests* will be invoked. The procedure *ClusterTests* (Algorithm 2) takes three arguments: 1) all clusters generated so far (*Clusters*), 2) all explored passing tests (*passList*) and 3) all failing tests that have not been clustered (*failList*). In this function, the culprit constraints of all failing tests in *failList* is computed (lines 2–9) and, then, the function *Update* is called (line 10) to cluster the failing tests accordingly.

The *Update* function (Algorithm 2) can place a failing test into an existing cluster or create a new one depending on the culprit constraint (reason) of the test. We base our clustering heuristic on the intuition that the reason of failure of each test within a cluster should be *subsumed* by a core reason (r_c) represented by the cluster. Hence, for a given failing test f (with a reason of failure r_f) being clustered and a set of all clusters *Clusters*, the following three cases can arise:

- **There exists $c \in C$ such that r_c subsumes r_f**: in this case, we add the test *f* to the cluster *c* (line 18);
- **There exists $c \in C$ such that r_f subsumes r_c**: in this case, we *generalize* the core reason for cluster *c* by resetting r_f as the general reason for failure for tests in cluster *c* (lines 21–22);
- **No cluster reason subsumes r_f, and r_f subsumes no cluster reason**: in this case, we *create* a new cluster *c'* with the sole failing test *f* and attribute r_f as the core reason of failure for tests in this cluster (line 26).

4.2 Clustering-aware Search Strategy

It is easy to see that Algorithm 1 will yield the correct *culprit constraints* if the search strategy followed is DFS: once a failing path f_i is encountered, the passing path that shares the maximum common prefix with f_i is either the last *passing* path encountered before the failure, or is the next *passing* path after f_i (i.e. ignoring all failures in the interim). Hence, a depth-first traversal of the symbolic execution tree will always find the culprit constraints by constructing the *largest common prefix* of the failing paths with at most two passing paths (the passing executions just before and just after encountering the failures).

However, DFS has a very poor coverage when used with a time-budget. Hence, we require search strategies different than DFS (like the Random and CoverNewCode strategies in KLEE) to achieve a good coverage. In fact, during our experiments, we could not trigger most of the failures in our benchmarks with DFS within reasonable timeouts.

We design a new *clustering-aware search strategy* (CLS) that is capable of discovering the precise culprit constraint while achieving a high coverage at the same time. CLS is built on a crucial observation that we *only* require DFS on a failing test to guide the search to its nearest passing test; on a passing test, the next test can be generated as per any search heuristic. Hence, one can implement any combination of suitable search strategies (to achieve high code coverage) while maintaining the use of DFS on encountering a failure (to identify the culprit constraint precisely).

We leverage a so-called *branching tree*, a data structure maintained by many symbolic execution engines (like KLEE) to record the symbolic execution tree traversed in terms of the branching/forking history (KLEE refers to it as the *process tree*). Let us illustrate as to how we combine an arbitrary search strategy (*SS*) with DFS exploration to implement an instance of CLS using the branching tree in Figure 3. In the tree, $i1$–$i7$ are internal nodes while $p1$–$p8$ are leaf nodes. Note that in the following paragraphs, we will use the term (leaf) node and path interchangeably. Basically, CLS works in two phases:

Phase 1: ***SS*** **searches for a failing test.** The search heuristic *SS* searches for a failure using its own algorithm. Suppose *SS* first detects a failing path $p5$, it returns control to CLS that now switches to the DFS heuristic (to locate the "nearest" passing test, i.e. the one that has the longest common prefix with $p5$).

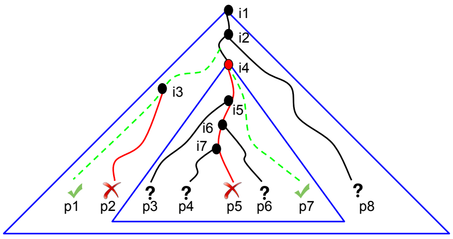

Fig. 3: A Branching Tree

Phase 2: DFS looks for "nearest" passing test. Continuing with our example (Figure 3), by the time *SS* detects the failing path $p5$, assume that we have explored three paths $p1$, $p2$, $p7$ and successfully put the failing path $p2$ into its correct cluster. So, now only four active paths remain: $p3$, $p4$, $p6$ and $p8$. At this point, our CLS seach strategy uses another crucial observation: since $p7$ is a passing path and $i4$ is the closest common ancestor node of $p5$ and $p7$, the nearest passing path for $p5$ must be $p7$ or another passing path spawned from intermediate nodes $i5$, $i6$ or $i7$. Hence, we can reduce the search space for finding the nearest passing path for $p5$ from the space represented by outer blue triangle to the inner (smaller) triangle (as $p7$ is a passing path, it must be the nearest passing path if no "nearer" passing path is discovered in the subtree rooted at $i4$). We omit the details of how it is acheived for want of space.

If the symbolic execution is run with a timeout setting, the timeout can potentially fire while CLS is searching for the nearest passing path to a failing execution. In this case, we simply pick the nearest passing path to the failing execution among the paths explored so far to compute the culprit constraint.

Our technique is potent enough to cluster an existing test-suite by running the symbolic execution engine needs to run in a mode that the exploration of a path that is controlled by the failing test (like the "seed" mode in KLEE [4]). During path exploration, the first passing test encountered in a depth-first traversal seeded from the failing test t would necessarily be the passing test that has the longest common prefix with t. Thus, we can compute the *culprit constraint* accordingly, and use it to form a new cluster or update an existing cluster.

4.3 Generalize reasons for failure

Consider Figure 4: the program checks if the absolute value of each element in the array is greater than 0. The buggy assertion contains > comparison instead of ≥), which would cause 10 failing test cases $\forall\ i \in \{0..9\}$. Since each array element is modeled as a different symbolic variable, all 10 cases are clustered separately.

```
int main() {
  int arr[10], int i;
  make_input(arr, sizeof(arr));
  for (i = 0; i < 10; i++) {
    if (a[i] < 0) a[i] = -a[i];
    assert(a[i] > 0); // a[i] >= 0
  }
}
```

Fig. 4: Generalization for arrays

In such cases, we need to *generalize* errors that occur on different indices but due to the same core reason. For example, if the reason is: $arr[4] > 0 \wedge arr[4] < 10$, we change this formula to $\exists x\ (arr[x] > 0 \wedge arr[x] < 10)$. Note that this is only a heuristic, and our implementation allows the user to disable this feature.

5 Experimental Evaluation

We evaluated our algorithm on a set of 21 programs: three programs from *IntroClass* [10] (a micro benchmark for program repair tools) and the remaining eighteen real-world programs taken from four benchmarks-suites: eleven programs from Coreutils[1] version 6.10, three from SIR[7], one from BugBench[16] and three from exploit-db[2]. The three subject programs from exploit-db (downloaded them from the project's website) were used in [11]. The bugs in IntroClass, Coreutils, exploit-db and BugBench programs are real bugs, whereas the ones in SIR are seeded.

We manually inserted `assert` statements in the programs taken from the IntroClass benchmark to specify the test oracle, while all remaing 18 real-world programs were kept unchanged. During symbolic execution, the failing test cases are generated due to the violation of embedded assertions or triggering of runtime errors (captured by KLEE) like divide-by-zero and invalid memory accesses.

We compared our symbolic-analysis based (SAB) test clustering method to two baseline techniques: call-stack based (CSB) and point-of-failure based (PFB) clustering. While SAB refers to the implementation of our algorithm within KLEE, we implemented CSB and PFB on top of KLEE to evaluate our implementation against these techniques. Specifically, our implementation first post-processes the information of test cases generated by KLEE to compute the stack hash (on function call stack) and extract failure locations. Based on the computed and extracted data, they cluster the failing tests.

We conducted all of the experiments on a virtual machine created on a host computer with a 3.6 GHz Intel Core i7-4790 CPU and 16 GB of RAM. The virtual machine was allocated 4 GB of RAM and its OS is Ubuntu 12.04 32-bit. For our experiments, we use the clustering-aware search strategy (CLS), enable array generalization and use a timeout of one hour for each subject program. KLEE is run with the `--emit-all-errors` flag to enumerate all failures.

5.1 Results and Analysis

Table 2 shows the results from our experiments on selected programs. `Size` provides the size of the program in terms of the number of LLVM bytecode instructions. `#Fail Tests` provides the number of failing tests. The rest of the

Table 2: Test Clustering: num. of clusters

Program	Repository	Size (kLOC)	#Fail Tests	#C PFB	#C CSB	#C SAB
median	IntroClass	1	7	1	1	5
smallest	IntroClass	1	13	1	1	3
syllables	IntroClass	1	870	1	1	5
mkfifo	Coreutils	38	2	1	1	1
mkdir	Coreutils	40	2	1	1	1
mknod	Coreutils	39	2	1	1	1
md5sum	Coreutils	43	48	1	1	1
pr	Coreutils	54	6	2	2	4
ptx	Coreutils	62	3095	16	1	3
seq	Coreutils	39	72	1	1	18
paste	Coreutils	38	4510	10	1	3
touch	Coreutils	18	406	2	3	14
du	Coreutils	41	100	2	2	8
cut	Coreutils	43	5	1	1	1
grep	SIR	61	7122	1	1	11
gzip	SIR	44	265	1	1	1
sed	SIR	57	31	1	1	1
polymorph	BugBench	25	67	1	1	2
xmail	Exploit-db	30	129	1	1	1
exim	Exploit-db	253	16	1	1	6
gpg	Exploit-db	218	2	1	1	1

Table 3: Test clustering: overhead

Program	#Pass paths	#Fail paths	Time (sec)	Ovrhd (%)
median	4	7	5	∼0
smallest	9	13	5	∼0
syllables	71	870	1800	4.35
mkfifo	291	2	3600	∼0
mkdir	326	2	3600	∼0
mknod	72	2	3600	∼0
md5sum	62449	48	3600	0.42
pr	540	6	3600	∼0
ptx	9	3095	3600	2.04
seq	445	72	1800	0.73
paste	3501	4510	3600	16.17
touch	210	406	3600	0.84
du	44	100	3600	0.81
cut	38	5	3600	∼0
grep	169	7122	3600	34.13
gzip	5675	265	3600	0.7
sed	3	31	3600	0.03
polymorph	3	67	3600	14.36
xmail	1	129	3600	0.06
exim	178	16	3600	0.03
gpg	10	2	3600	∼0

columns provide the number of clusters (`#C`) for Point-of-failure (`PFB`), Stack Hash (`CSB`) and our Symbolic Analysis (`SAB`) based methods. Note that `#C(PFB)` doubles up to also record the number of failing locations. As KLEE symbolically executes the LLVM [14] bitcode, we show the size of the program in terms of the total lines of the LLVM bitcode instructions.

In several programs (like **ptx**, **paste**, **grep**) SAB places thousands of failing tests into managable number of clusters. Compared to CSB, in 12 out of 21 subjects (∼57%), our method produces more fine-grained clustering results. Compared to PFB, our technique expands the number of clusters to get a more fine-grained set in 10/21 subjects. However, our method also collapses the clusters in case the program has failures that are likely to be caused by the same bug but the failures occur at several different locations (like **ptx** and **paste**).

RQ1. Does our technique produce more fine-grained clusters?In the experiments, we manually debugged and checked the root causes of failures in all subject programs. Based on that, we confirm that our SAB approach does effectively produce more fine-grained clusters. For instance, as shown in Figure 5, the buggy **smallest** program, which computes the smallest number among four integer values, does not adequately handle the case in which at least two of the smallest integer variables are equal. For example, if d equals b, none of the four conditional statements (at lines 7, 9, 11 and 13) take the true branch; the result is incorrect as the variable `smallest` then takes an arbitrary value.

```
int a, b, c, d, smallest;
make_symbolic(a, b, c, d);
assume(a>=-10 && a<=10);
assume(b>=-10 && b<=10);
assume(c>=-10 && c<=10);
assume(d>=-10 && d<=10);
if (a < b && a < c && a < d)
    smallest = a;
if (b < a && b < c && b < d)
    smallest = b;
if (c < b && c < a && c < d)
    smallest = c;
if (d < b && d < c && d < a)
    smallest = d;
assert(smallest ==
      golden_smallest(a,b,c,d));
```

Fig. 5: Code snippet from ‘smallest’

```
1  case ’e’:
2  if (optarg)
3    getoptarg (optarg, ’e’, ...);
4  //...
5  break;
6  //other cases
7  case ’i’:
8  if (optarg)
9    getoptarg (optarg, ’i’, ...);
10 //...
11 break;
12 //other cases
13 case ’n’:
14 if (optarg)
15   getoptarg (optarg, ’n’, ...);
16 break;
```

Fig. 6: Code snippet from ‘pr’

As shown in Figure 5, we instrumented the program to make it work with KLEE. During path exploration, KLEE generated 13 failing tests for this program and the CSB technique placed all of them into one cluster as they share the same call stack. However, our SAB approach created three clusters with the following reasons: (Cluster 1) $d \geq b$, (Cluster 2) $d \geq c$ and (Cluster 3) $d \geq a$. The reasons indeed show the corner cases that can trigger the bugs in the program. We observed similar cases in **median** and **syllables** programs (see Table 2).

In the subject program **pr** (a Coreutils utility), we found that 6 failing tests due to two different bugs are placed in two clusters on using stack hash similarity. Meanwhile, our approach placed these 6 failing tests into 4 different clusters: one cluster contained 3 failing tests corresponding to one bug, and the other three clusters contain three failing tests of the second bug. Figure 6 shows a code snippet from **pr** that shows three call sites for the buggy function *getoptarg()* (at lines 3, 9 and 15). In this case, because all of the three call sites are in one function, so the stack hash based technique placed the three different failing paths in the same cluster. Similar cases exist in the **exim** and **du** applications.

RQ2. Can our clustering reasons (culprit constraints) help users to look for root causes of failures? One advantage of our bucketing method compared to CSB and PFB approaches is its ability to provide a *semantic characterization* of the failures that are grouped together (based on the culprit constraint). The existing techniques are only capable of capturing syntactic information like the line number in the program or the state of the call-stack when the failure is triggered.

Table 4: Sample culprit constraints

Program	Culprit constraint
mkfifo	`(= (select arg0 #x00000001) #x5a)`
pr	`(= (select stdin #x00000009) #x09)`

Table 4 shows a few examples of the culprit constraints that our technique used to cluster failing tests for **mkfifo** and **pr**. In **mkfifo**, the culprit constraint can be interpreted as: *the second character in the first argument is the character ‘Z’*. This is, in fact, the correct characterization of this bug in **mkfifo** as the tests in this cluster fail for the "-Z" option. In case of **pr**, the culprit constraint indicates that: *the tenth character of the standard input is a horizontal tab (TAB).* The root cause of this failure is due to incorrect handling of the backspace and horizontal tab characters.

RQ3. What is the time overhead introduced by our bucketing technique over vanilla symbolic execution?Overall, in most of the subject programs the time overhead is negligible (from 0% to 5%), except in some programs where the overhead is dominated by the constraint solving time.

5.2 User Study

A user study was carried out with 18 students enrolled in a Software Security course (CS4239) in the National University of Singapore (NUS) to receive feedback on the usability and effectiveness of our bucketing method. Among the students, there were 14 senior undergraduate and 4 masters students. Before attending the course, they had no experience on applying bucketing techniques.The students were required to run the three bucketing techniques (our method and two others based on call-stack and point of failure information) to cluster the found failing tests, and (primarily) answer the following questions:

Q1. Rate the level of difficulty in using the three techniques for bucketing failing tests.
Q2. To what extent do the bucketing techniques support debugging of program error?
Q3. Are the numbers of clusters generated by the bucketing techniques manageable?

The users' responses for Q1 & Q2 are summarized in Table 5; for example, the first cell of Table 5 shows that 8 of the 18 respondents found the PFB technique "Easy" for bucketing. In response to Q3, 14 out of the 18 respondents voted that the number of clusters generated by our technique is manageable.

Table 5: Responses from the user study. Q1 enquires about the difficulty of using a technique: Easy (E), Moderate (M), Difficult (D) and Very Difficult (VD). Q2 responses are about the usefulness of a method: Not Useful (N), Useful (U) and Very Useful (VU).

Bucketing	Difficult(Q1)				Useful(Q2)		
Technique	E	M	D	VD	N	U	VU
Point of failure (PFB)	8	8	2	0	0	7	11
Stack hash (CSB)	3	13	2	0	3	8	7
Symbolic analysis (SAB)	1	9	7	1	2	4	12

In terms of usefulness as a debugging aid, our technique is ranked "Very Useful" by 12 of the 18 respondents. It gains a high rating for its usefulness as it provides a *semantic characterization* for each bucket (in terms of the culprit constraint), that can help users locate the root cause of failure. At the same time, we found that the main reason that they found our technique harder to use was that this characterization was shown in the form of logical formula in the SMT-LIB format—a format to which the students did not have enough exposure.

We list some of the encouraging feedback we got:

- "I believe it is the most powerful of the three techniques, letting me understand which assert are causing the crash or how it is formed."
- "It is very fine grain and will allow us to check the path condition to see variables that causes the error."

6 Related work

One related line of research involves clustering crash reports or bug reports [6], [8], [12], [17], [20]. Crash Graph [12] uses graph theory (in particular, similarity between graphs), to detect duplicate reports. In terms of duplicate bug report detection, Runeson [20] proposed a technique based on natural language processing to check similarity of bug reports.

Another relevant work involves clustering program failing traces. Liu and Han [15] proposed the technique to use results of fault localization methods for clustering failing traces. Given two set of failing and passing traces which are collected from instrumented predicates of software program, they statistically localize the faults and two failing traces are considered to be similar if the pointed fault locations in the two traces are the same. Podelski et al.[19] cluster failure traces by building symbolic models of their execution (using model checking tools) and use interpolants as signatures for clustering tests. Due to the cost of symbolic model-checking, their technique seems to suffer from scalability issues as in their experiments, even their intraprocedural analysis times out (or the interpolant generator crashes) on a large number of methods.

Although the above-mentioned lines of research are relevant to our work, we target our research on clustering *failing tests* — instead of crash reports, bug reports or failing traces. The other lines of research do not assume they have concrete test inputs to trigger the bugs, so they build the techniques on exploring run-time information collected in the field (i.e.,where the software systems are deployed). In our case, we work on failing tests obtained during symbolic exploration of software programs or provided by test teams.

To the best of our knowledge, all popular symbolic execution engines only borrow and slightly change the techniques that have been proposed for clustering crash reports to cluster their generated failing tests. The clustering approach can be as simple as using point of failure in KLEE [4] or using call stack information in SAGE [9] and MergePoint [3].

7 Conclusions

We leverage the symbolic execution tree built by a symbolic execution engine to cluster failing tests found by symbolic path exploration. Our approach can also be implemented on symbolic execution engines like S2E [5] for clustering tests for stripped program binaries (when source code is not available). Unlike many other prior techniques, our technique should be able to handle changing of addresses when Address Space Layout Randomization (ASLR) is enabled as symbolic expressions are unlikely to be sensitive to address changes.

8 Acknowledgment

This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

References

1. Coreutil benchmarks. `http://www.gnu.org/software/coreutils/coreutils.html`.
2. Exploit-db benchmarks. `https://www.exploit-db.com/`.
3. T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.
4. C. Cadar, D. Dunbar, and D. 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*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.
5. V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multipath analysis of software systems. *SIGPLAN Not.*, 47(4):265–278, Mar. 2011.
6. Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1084–1093, Piscataway, NJ, USA, 2012. IEEE Press.
7. H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, Oct. 2005.
8. K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 103–116, New York, NY, USA, 2009. ACM.
9. P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, Mar. 2012.
10. C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, Dec 2015.
11. W. Jin and A. Orso. F3: Fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 213–223, New York, NY, USA, 2013. ACM.
12. S. Kim, T. Zimmermann, and N. Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 486–493, Washington, DC, USA, 2011. IEEE Computer Society.
13. V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, New York, NY, USA, 2012. ACM.
14. C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.
15. C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 46–56, New York, NY, USA, 2006. ACM.

16. S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools.
17. N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet. Automatically identifying known software problems. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, ICDEW '07, pages 433–441, Washington, DC, USA, 2007. IEEE Computer Society.
18. D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 67–82, Berkeley, CA, USA, 2009. USENIX Association.
19. A. Podelski, M. Schäf, and T. Wies. *Classifying Bugs with Interpolants*, pages 151–168. Springer International Publishing, Cham, 2016.
20. P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.