Deferred Concretization in Symbolic Execution via Fuzzing

Awanish Pandey
Computer Sc. and Engg.
IIT Kanpur, India
awpandey@cse.iitk.ac.in

Phani Raj Goutham Kotcharlakota*
IIT Kanpur, India
gouthamk@alphonso.tv

Subhajit Roy
Computer Sc. and Engg.
IIT Kanpur, India
subhajit@cse.iitk.ac.in

ABSTRACT

Concretization is an effective weapon in the armory of symbolic execution engines. However, concretization can lead to loss in coverage, path divergence, and generation of test-cases on which the intended bugs are not reproduced. In this paper, we propose an algorithm, *Deferred Concretization*, that uses a new category for values within symbolic execution (referred to as the *symcrete* values) to pend concretization till they are actually needed. Our tool, Colossus, built around these ideas, was able to gain an average coverage improvement of 66.94% and reduce divergence by more than 55% relative to the state-of-the-art symbolic execution engine, KLEE. Moreover, we found that KLEE loses about 38.60% of the states in the symbolic execution tree that Colossus is able to recover, showing that Colossus is capable of covering a much larger coverage space.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Formal software verification**; **Dynamic analysis**.

KEYWORDS

Symbolic Execution, Software Testing, Fuzzing

ACM Reference Format:
Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. 2019. Deferred Concretization in Symbolic Execution via Fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330554

1 INTRODUCTION

Accurate modeling of real-world constructs like external libraries, floating-point operations, system calls, vector instructions, and non-linear arithmetic is perhaps the biggest challenge for symbolic execution. Symbolic execution engines circumvent these challenges via *concretizations*: they natively execute the problematic instructions, and then, pull the result of the operation (as concrete values) back into symbolic execution, thereby allowing the analysis to continue. Symbolic execution enabled with concretizations (often referred to as Dynamic Symbolic Execution) has been applied to wide-spread applications from program repair [22, 23], debugging [3, 10], bug synthesis [27], regression testing [20] and failure clustering [24].

Though effective at handling real-world code, concretization introduces some problems:

- **Loss in Coverage**: some program paths are omitted from the analysis, leading to missed bugs and loss in coverage;
- **False Positives**: the execution can be led into infeasible program paths, potentially raising false alarms;
- **Reproducibility**: generation of incorrect failing tests, i.e, a failure cannot be replayed on the generated test-case.

In this paper, we discuss how each of the above guarantees gets broken due to concretizations and, then, detail our solution, *Deferred Concretization*. Our algorithm introduces a new category of (symbolic) values, *symcrete*s, to handle concretized values (like return values from external library calls). A *symcrete* value masquerades as a symbolic value for almost all purposes but also "hides" a concrete value consistent with the respective execution path (resulting from concretizations). As the *symcrete* values are retained in symbolic constraints, it prevents any loss of information that could have led to loss in coverage, false-positive, or irreproducibility.

As the witnesses in the *symcrete* values feed from program constructs that cannot be modeled symbolically (like external library calls), we design a *fuzz-based constraint solver* to handle constraints on *symcrete* values. Our fuzz-based solver translates satisfiability queries on logical constraints to reachability queries on programs and, then, uses an off-the-shelf fuzzer on the generated program. If the execution along any path is prohibited by the current set of concrete values, we use the fuzz-based solver to search for new concrete values that can draw the symbolic execution engine along the required path; this allows us to recover from loss in coverage due to concretization. The breakthrough improvements in fuzzing in the last couple of years (which, we believe, will continue) makes this an interesting approach for the formal methods community.

Our tool, Colossus, improves the coverage significantly for many programs: for instance, it increases the coverage in `cut` from a mere 5.37% to 71.81% (an improvement of over 1237%); many other programs like `date`,`mkfifo`,`split`,`tr` show an increase in coverage over 27% (i.e. an improvement by over 115%) over KLEE[6]. We conducted a deeper analysis on *state coverage*: we found that Colossus is able to cover about 38.60% (on an average) more states than KLEE (that are otherwise lost to concretizations). Finally, in our experiments, Colossus improves (reduces) the rate of divergence by over 55% relative to KLEE.

The contributions of this work are as follows:

- We articulate the core problems that lead to loss in coverage, path divergence and irreproducibility in symbolic execution.

*Now at Alphonso Labs Private Limited, India

- We propose *Deferred Concretization* to solve the above problems: our algorithm introduces a new category of (symbolic) values in the symbolic execution, *symcrete*, to drive demand-driven concretizations;
- We design a *fuzz-based constraint solver* that employs an off-the-shelf fuzzer to solve constraints on *symcrete* values;
- We build our ideas into a tool, Colossus; our experiments demonstrate that our ideas improve upon a state-of-the-art symbolic execution engine, KLEE, in all three dimensions—coverage, divergence and reproducibility of tests.

2 OVERVIEW

2.1 Preliminaries

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. A path p is *feasible* if its path condition ψ_p is satisfiable; otherwise p is *infeasible*.

An execution state, $\mathcal{S}$, is maintained as a tuple (l, pc, Ω): the location l, the path condition pc and the variable map Ω. The variable map, $\Omega : v \mapsto \{\alpha, c\}$, maps each program variable $v \in \mathcal{V}$ to a *symbolic* value α (notated using greek letters) or a *concrete* value c (notated using latin characters a–e). We use strings or latin characters u–z for variable names.

2.2 Symbolic Execution (SE)

Symbolic execution (SE) has evolved over the last three decades with multiple algorithms; today, most SE engines belong to the following two primary *styles* [8]):

- **Concolic Testing:** Concolic execution, employed in successful projects like CREST [5] and DART [18], commence with random inputs (say $\vec{I_0}$); once the execution terminates, the engine uses the generated path condition (pc_0) of the current path to construct a new path condition pc_1 (say by negating the last predicate [18]); solving pc_1 provides inputs ($\vec{I_1}$) that would explore a new path. The program is again executed with I_1, repeating the above process.
- **Execution-Generated Testing (EGT):** The EGT approach, employed by tools like EXE [7], SPF [25] and KLEE [6], *fork* a symbolic execution at each conditional branch (where both directions are feasible) to maintain multiple partial paths, orchestrating their executions simultaneously.

We describe our algorithm on EGT-style symbolic execution; in particular, our prototype is built on the state-of-the-art EGT-style symbolic execution engine, KLEE.

Algorithm 1 shows the Execution-Generated Testing (EGT) symbolic execution algorithm; **please ignore the parts shaded in color (these refer to our modifications to the base algorithm that we discuss later)**. The algorithm works on a simplified intermediate representation where conditional and looping constructs have been compiled down to conditional control transfers (`if (cond) goto l`). Also, `assert` statements are compiled down to a reachability check: $\boxed{\texttt{assert(e)}} \implies \boxed{\texttt{if (e) then fail}}$.

The function `succ(l)` provides the set of next location(s) after the current location `l`. For simplicity we do not show the implementation of the `call` statements to *local* functions (i.e. functions whose definitions are available); this interprocedural extension is simple—copy actual parameters to formal parameters, invoke the function, and copy return value back to the parent procedures.

Updates to the variable map Ω for the variable v to a new value, say c, is shown via the notation $\Omega[v \to c]$.

Though we call `v:=extop()` as the external operation instruction; any instructions for which symbolic reasoning is not available is denoted by this instruction—external (binary-only) library calls, system calls, vector instructions, expressions involving non-linear arithmetic etc.

Algorithm 1 maintains a set of *active* execution states in a worklist W; execution commences from the initial state—program entry-point l_0, path condition as `true` and an empty variable map (line 1).

Algorithm 1 Symbolic Exploration

```
1:  $W \leftarrow \{(l_0, true, \emptyset)\}$                                  ▷ initial worklist
2:  while $W \neq \emptyset$ do
3:      $(l, pc, \Omega) \leftarrow pickNext(W)$
4:      $S \leftarrow \emptyset$
5:      $T \leftarrow \emptyset$
6:      switch $instrAt(l)$ do                                  ▷ execute instruction
7:          case $input(v)$                                       ▷ input instruction
8:              $S \leftarrow \{(succ(l), pc, \Omega[v \to \alpha])\}$, fresh $\alpha$
9:          case $v := e$                                    ▷ assignment instruction
10:             $S \leftarrow \{(succ(l), pc, \Omega[v \to seval(\Omega, e)])\}$
11:         case if $(b)$ goto $l'$                              ▷ branch instruction
12:             $e \leftarrow seval(\Omega, b)$
13:             if $(isSat(pc \wedge e) \wedge isSat(pc \wedge \neg e))$ then
14:                 $S \leftarrow \{(l', pc \wedge e, \Omega), (succ(l), pc \wedge \neg e, \Omega)\}$
15:             else if $(isSat(pc \wedge e)$ then
16:                 $S \leftarrow \{(l', pc \wedge e, \Omega)\}$
17:                 $(res, \xi) \leftarrow Fuzz(pc \wedge \neg e)$
18:                 if $res = Success$ then
19:                     $T \leftarrow \{(l', pc \wedge e, \Omega[\xi])\}$
20:                 end if
21:             else                                                ▷ $isSat(pc \wedge \neg e)$
22:                 $S \leftarrow \{(succ(l), pc \wedge \neg e, \Omega)\}$
23:                 $(res, \xi) \leftarrow Fuzz(pc \wedge e)$
24:                 if $res = Success$ then
25:                     $T \leftarrow \{(l', pc \wedge e, \Omega[\xi])\}$
26:                 end if
27:             end if
28:         case $fail$                                                         ▷ error
29:             GenerateTest$(l, pc, \Omega, \texttt{FAIL})$
30:         case $v := extop(w_1, w_2, \ldots)$                           ▷ Concretization
31:             $a_1, a_2, \cdots \leftarrow$ GetConcretes$(pc, \Omega, w_1, w_2, \ldots)$
32:             $c \leftarrow$ NativeExecute$(pc, \Omega, extop, a_1, a_2, \ldots)$
33:             $S \leftarrow \{(succ(l), pc, \Omega[v \to c])\}$
34:             Let $\Omega[w_1] = \alpha_1, \Omega[w_2] = a_2, \ldots$
35:                                   ▷ $w_1$ was symbolic, $w_2$ was concrete...
36:             $\Omega' \leftarrow \Omega[v \to \langle\gamma, c\rangle, w_1 \to \langle\alpha_1, a_1\rangle, w_2 \to a_2, \ldots]$
37:                                                                        ▷ $\gamma$ fresh
38:             $\Phi \equiv (\langle\gamma, c\rangle = extop(\langle\alpha_1, a_1\rangle, \ldots))$
39:             $S \leftarrow \{(succ(l), pc \wedge \Phi, \Omega')\}$
40:         case $halt$                                                ▷ terminate path
41:             GenerateTest$(l, pc, \Omega, \texttt{PASS})$
42:     $W \leftarrow W \cup S \cup T$                                     ▷ update worklist
43: end while
```

While the worklist is non-empty, it picks a state from the worklist (using heuristics referred to as *search criteria*) and proceeds to handle it depending on the instruction type (lines 7-40).

The inputs to the programs are marked via the `symbolic(v)` instruction: on encountering this instruction, the algorithm binds the input variable v to a fresh symbolic value α, and proceeds to the next location by adding this new state to the worklist (lines 7-8).

For the assignments statement `v:=e`, the algorithm evaluates the expression e on the current symbolic map Ω, and updates the binding of the variable v accordingly (lines 9-10).

For conditional control transfer statements (Line 11), the engine evaluates the branch condition into a symbolic expression e (line 12), and then, uses a constraint (logic) solver to check if both ends of the branch are feasible (line 13): if only one of the branches is feasible, execution proceeds along that direction (line 15 and 21). However, if both the directions are feasible (line 14), the engine *forks* off the execution state—the path conditions for the child states are constructed so as to include additional constraints specifying if the branch condition was `true` ($pc \wedge e$), or `false` ($pc \wedge \neg e$).

The `fail` instruction (line 28) terminates the progress of the current execution state, generating a *failing* test-case. The test-case is synthesized by querying a constraint (logic) solver on the *pc* for satisfiable assignments of the symbolic values. The `halt` statement (line 40) generates a *passing* test-case and, then fetches another state from the worklist, terminating the current state.

When the engine hits an external operation `v:=extop()` (line 30) that could not be handled symbolically by the assignment statement (`v := e`), it performs *concretization* (line 31) via the CONCRETE() function: it searches for concrete values a_i for the arguments w_i that are consistent with the current path condition *pc*; it uses these concrete values to natively execute the external operation, collecting a concrete return value *c*. Finally, it constructs the successor state by binding the variable v to this value *c* in Ω. *Please note that the variable bindings for the parameters are not altered in* Ω.

2.3 Example

Our motivating example (Listing 1) is inspired from the `cut` program in *coreutils-8.29*. Let the string functions `strcmp()` and `strchr()` be external operations. The program starts off by invoking the `getopt()` function on the string `arg`. The `getopt()` function has the following specification: when invoked with an input argument `arg` (say "`-b`") and a colon-separated set of options `optstring` (like `"b:c"`), it returns the character next to the hyphen in `arg` if this character is in the list of characters specified in `optstring`; otherwise, it returns `-1`. The `getopt()` function works as follows:

(1) It checks (line 3) if the parameter starts with a hyphen; if not, it simply returns with an error value (`-1`);
(2) Next, it checks if the option is "`--`" (line 4-5); in this case, it returns a value "?" (some programs use it as an indication to read from `stdin`);
(3) Next, it checks if the argument is just "`-`" without a following character (line 7); in this case it returns "`-`". Note that this check is equivalent to checking if the second character in `arg` is "`\0`" or not, as: (i) the `if`-statement at line 3 ensures that `arg[0]` is '`-`'; (ii) the second character `arg[1]` is '`\0`'. The above two conditions imply that (`strcmp(arg, "-")==0`);

```
1  int getopt(char *arg, const char *optstring) {
2  ✓✓✓ int ch = -1;
3  ✓✓✓ if(arg[0] == '-') {
4  ✓✓✓  r = strcmp(arg,"--");
5  ✓✓✓  if(r == 0)
6  ✗✓✓   ch = '?';
7  ✓✓✓  if(strcmp(arg, "-") == 0) {
8  ✗✓✓   ch = '-';
9  ✗✓✓   return ch;
10       }
11 ✓✓✓  if(arg[1]=='\0') // sanity check
12 ✓✓✗   assert(0);
13 ✓✓✓  char *e = strchr(optstring, arg[1]);
14 ✓✓✓  if(e != NULL)
15 ✗✓✓   ch = e[0];
16      }
17 ✓✓✓ return ch;
18 }
19
20 int main ( ) {
21 ✓✓✓ symbolic(arg);
22 ✓✓✓ optc = getopt(arg, "b:c");
23 ✓✓✓ if(optc != -1) {
24       switch(optc){
25 ✗✓✓     case 'b': ...
26 ✗✓✓     case 'c': ...
27 ✗✓✓     case '-': ...
28 ✓✓✓     case default: ...
29         }
30    // Yet another sanity check
31 ✓✓✓  if (optc!='b' && optc!='c' && optc!='-')
32 ✗✓✗   assert(0);
33   }
34 }
```

Listing 1: Motivating example

(4) Then, it performs a sanity-check on condition (ii) above: test if the second character is '`\0`';
(5) Finally, it checks if the given option (the second character of *arg*) is in the set of options and returns it; else it returns `-1`.

Let us see how Algorithm 1 (without the highlighted lines) operates on this example; the second column in Table 1 shows the path conditions:

(1) Our SE engine commences execution by binding fresh symbolic variables $\alpha_0\alpha_1\alpha_2\alpha_3$ to `arg`, say with an user specified bound of 4 for size of `arg` (Algo 1, line 8);
(2) It, then, calls the function `getopt()`, mapping the actual parameters to the formal parameters and jumping into `getopt()` (not shown in Algorithm 1);
(3) At line 3 of Listing 1, it *forks* the execution (Algo 1, line 14), for the cases where `arg[0]` is equal to '`-`' or not. This can be seen in the symbolic execution tree (Figure 2): the root node denoting the state S1 at line 3 forks off to states S2 and S3, transferring control to lines 4-5 and 17 (respectively) on conditions $\alpha_0 \neq 45$ and its negation (Note: 45 is the ASCII code for '`-`');
(4) With state S2, it encounters the external call `strcmp()` at line 5, thereby, applying *concretization* (Algo 1, line 31):

- solving the symbolic constraint on the path condition, it finds a feasible concrete value for `arg`, say "`-q\0\0`", which is consistent with the path condition $\alpha_0 = 45$;
- it natively executes `strcmp()` on this value, thereby evaluating `r = 5` (say);
- it continues symbolic execution with a concrete value of `r` but discards the concretized values of the argument, continuing with the symbolic value for `arg`.

(5) As there is only one feasible path in this case (the `false` path), a fork is not required at line 5 of Listing 1;
(6) The check at line 7 also fails, and is handled similarly;
(7) Line 11 branches on the symbolic variable `arg[1]`: as the symbolic argument for the parameter `arg` was retained, the SE engine finds that both outcomes are feasible at this branch, thereby failing at line 12. This is a **false positive**!
At this point, the engine also generates a test case for this failure by solving the current path condition $\alpha_0 = 45 \wedge \alpha_1 = 0$ to produce inputs "`-\0\0\0`". Note that running the program with this input takes the program to a different path that does not cause the intended failure—a frustrating situation for the user! This is referred to as **path divergence** [18]. In this case, the target path was infeasible, but path divergence is possible even when the target path was feasible.
(8) At line 13, it *concretizes* `argv[1]` (as per its *pc*, $\alpha_0 = 45 \wedge \alpha_1 \neq 0$), say getting "`-q\0\0`"; running `strchr()` on it produces `NULL`.
(9) Finally, failing the test at line 14, the function returns `-1` to the parent procedure.

The loss in coverage is not just contained in this function, but has a compounding effect—the SE engine is not able to cover any of the branches of the switch statement in the parent procedure due to the coverage lost to concretization in `getopt()`.

2.4 Discussion

We found three limitations of the baseline algorithm (above):

- **Loss in coverage**: Certain paths of the program were not covered, potentially leading to *loss in coverage* and *missed bugs*. For example, line 8 of Listing 1 was not reached.
- **False-positive**: The algorithm traverses infeasible program paths due to incomplete modeling of the path condition, potentially leading to *false positives* and *path divergence*. For example, line 12 of Listing 1 raises an alarm though the path was infeasible.
- **Failure to reproduce executions**: This is another side-effect of incomplete modeling of the path condition caused by *path divergence*.

Concretization is an underapproximation that attempts to maintain accuracy by trading off coverage; the alternative is an overapproximation (via fresh symbolic variables for the returned values from external operations) that trades off accuracy for coverage.

Overall, we have the following options for handling an external operation of the form $ret = extop(arg_1, arg_2, \ldots)$:

- ovarapprox *ret*, overapprox arguments: ensures coverage, but can cause path explosion due to opening up of an enormous number of infeasible paths; KLEE has the `-make-concrete-symbolic` setting to enable this, and it provides a (hacky) way of taming the path explosion problem by taking a user-provided *probability* of how many of the argument instances to turn symbolic;
- overapprox *ret*, underapprox arguments: this is not an interesting option—why concretize the arguments when the result is not concretized;
- underapprox *ret*, overapprox arguments: this is the default setting for KLEE; it causes loss in coverage, divergence and can lead to irreproducible executions, but has been found to generally work well in practice in terms of gaining coverage;
- underapprox *ret*, underapprox arguments: ensures that there is no divergence, but it can prune large parts of the SE tree, leading to loss in coverage.

Table 1: PC at different program points for the Listing 1

Line	PC		
	Concrete	**Symbolic**	**Colossus**
b1	$\alpha_0 = 45$	$\alpha_0 = 45$	$\alpha_0 = 45$
b2	$\alpha_0 \neq 45$	$\alpha_0 \neq 45$	$\alpha_0 \neq 45$
b3	✗	$\beta_0 = 0$	$\langle\beta_0, 0\rangle = 0$
b4	$113 \neq 0$	$\beta_0 \neq 0$	$\langle\beta_0, 113\rangle \neq 0$
b5	✗	$\beta_1 = 0$	$\langle\beta_1, 0\rangle = 0 \wedge \langle\alpha_1, 0\rangle = 0$
b6	$67 \neq 0$	$\beta_1 \neq 0 \wedge \alpha_1 = 0$	$\langle\beta_1, 67\rangle \neq 0 \wedge \langle\alpha_1, 67\rangle \neq 0$
b7	$\alpha_1 = 0$	$\alpha_1 = 0$	✗
b8	$\alpha_1 \neq 0$	$\alpha_1 \neq 0$	$\langle\alpha_1, 67\rangle \neq 0$
b9	✗	$\beta_2 \neq 0$	$\langle\alpha_1, 98\rangle \neq 0 \wedge \langle\beta_2, 98\rangle \neq 0$
b10	$0 = 0$	$\beta_2 = 0$	$\langle\alpha_1, 100\rangle \neq 0 \wedge \langle\beta_2, 0\rangle = 0$
b11	✗	$\beta_3 \neq -1$	$\langle\beta_2 \neq 98\rangle \neq 1$
b12	✗	$\beta_3 \neq 98 \wedge \beta_3 \neq 99 \wedge \beta_3 \neq 45 \wedge \beta_3 \neq 63$	✗

We ran Listing 1 on KLEE and measured the coverage for the tests generated corresponding to paths that KLEE could execute to completion. The string functions, `strcmp()` and `strchr()`, were treated as external calls. The first row in green shows the lines covered by KLEE: as can be seen, it fails to cover many of the lines (loss in coverage) and also flags a false positive at line 12 (divergence). The second row in blue corresponds to the case where we turn the return value from the external operation as symbolic; in this case, it is able to cover almost all lines; however, for long programs, it can get stuck into exploring infeasible paths. At the same time, it flags two false positives at line 12 and line 32. The last column shows the response from our tool, Colossus: it covers all the feasible lines and does not flag any false positives.

Figure 2 shows the (partial) symbolic execution tree for the program: the dotted nodes refer to infeasible paths. Each node refers a *state* (only states at the forks are shown). The colored dots show if an algorithm is able to visit a given node: green is for the baseline algorithm, blue is for the case when the returned values are marked as symbolic and red is our proposed algorithm. In terms of *state coverage*, only our algorithm is able to cover all feasible states and avoid all the infeasible ones.

3 DEFERRED CONCRETIZATION

Consider Listing 2: why do we lose coverage at the external call $\boxed{\texttt{x = extop(y, z)}}$, even with concretization? Because, we would have generated only one concrete value for `x` at `L1`, while we need *two* values to cover both the arms of the branch at L2! Moreover,

```
L0: y = extop2(w);
L1: x = extop(y, z);
L2: if (x > 42)
L3:   ...
    else
L4:   ...
...
L5: if (x > 84)
L6:   ...
    else
L7:   ...
```

Listing 2: Requirement of *symcret*

```
int main() {
  char arg[3];
  char *x11,x10,x12;
  read(arg);
  x10 = (45==arg[0]);
  if(x10) {
    x11 = "--";
    x12=strcmp(arg,x1);
    if(!x12) assert(0);
  }
}
```

Listing 3: Snippet generated for Node S5

as x now binds to a concrete value, even at all subsequent branches that involve the variable x, the symbolic execution will be able to follow only one of the branch outcomes. Hence, we end up losing the entire symbolic execution tree corresponding to the other arm of the branch at L1.

Why not create two such values? Or, four? It is not possible to answer these questions at the location where the external call is invoked as we do not *yet* have access to the following information:

- **What should be the constraints on the concrete values?** At the location where the external call is invoked, the required constraints on x depends on the branch conditions—*that are yet to be visited!* For example, in Listing 2, though *concretization* due to the external call happens at L1, it is only at line L2 that one gets to know that x needs concrete values in the ranges $(-\infty, 42]$ and $[43, \infty)$ to cover both L3 and L4.
- **How many concrete values to generate?** Again, while the external call is executed at L1, one only discovers later that two concrete values are needed to cover both outcomes of the branch L2, and subsequently, two additional concrete values for each of the executions through branch L5—that is, a total of four concrete values for complete path coverage of the program. This information is not available at L1.

Also, the concretized values of y and z that drive the native execution of extop() in search for a concrete value of z must form a *consistent tuple* with x under extop(): hence, whenever another concretization attempt is needed in search for a different value of x, we must appropriately update the values of y and z as well. Further, in this case, the value of variable y is fetched from another external call extop2(); any change in y should transitively lead to an update of w. In summary, any concretization attempt must be applied together on this set of variables $\{w, y, x, z\}$ corresponding to a *consistent concretization set.*

The above problems occur only in EGT-style symbolic execution engines as they have to maintain active states corresponding to multiple (partial) executions.

The problem is not just about the return value but also the values of the arguments: the arguments and the return values together form a *consistent tuple* bound by the semantics of the external operation; for example, for z=sum(x,y), (z=5,x=2,y=3) is a consistent tuple but (z=4,x=2,y=3) is not! Hence, one needs to consider over/under approximation choices for the arguments as well.

Definition 3.1. We say $(r, c_1, c_2, \ldots, c_n)$ is a **consistent tuple** *under f* if executing $f(c_1, c_2, \ldots, c_n)$ returns r.

Definition 3.2. We say that set of *symcrete* variable mappings $\xi = \{x_1 \mapsto \langle \alpha_1, c_1 \rangle, x_2 \mapsto \langle \alpha_2, c_2 \rangle, \ldots, x_n \mapsto \langle \alpha_n, c_n \rangle\}$ is a **consistent concretization set** under an execution Δ (or path condition *pc*) if $\{x_1, \ldots, x_n\}$ is the set of all the variables corresponding to the arguments and return values of external operations and $c_1, \ldots, c_n$ are their corresponding concrete values along the execution Δ (or *pc*). It can be obtained by taking a closure over the consistent tuples under all external operations on the execution Δ (or *pc*).

3.1 The Notion of *symcrete* Values

We handle the above problems by introducing a new category of values, *symcrete* values. A *symcrete* (**sym**bolic-con**crete**) value is essentially a symbolic value for which we also maintain (or "hide") a concrete witness. *Symcrete* values are notated as a tuple $\langle \alpha, c \rangle$ over a symbolic value α and a concrete value (witness) c. The variable map, Ω, maps each program variable $v \in \mathcal{V}$ to a symbolic, concrete or *symcrete* value. Such a *symcrete* variable map (and the corresponding *symcrete* state) is valid only if it constitutes a *consistent concretization set.*

3.2 The *Deferred Concretization* Algorithm

The *Deferred Concretization* algorithm uses two solvers:

- **Logic Solver:** This is an SMT solver that is used by the baseline symbolic execution engine (like STP [15]); we show calls to this solver via isSat().
- **Fuzz Solver:** Handling *symcrete* values requires us to extend reasoning over executable interpretations of external operations for which logical interpretations are not available. Hence, we design a *fuzz-based constraint solver* (or simply *fuzz solver*), to solve such constraints.
  Given a path condition, *pc*, the *fuzz solver* routine, Fuzz(), searches for concrete values c_i such that the variable map $x_i \mapsto \langle \alpha_i, c_i \rangle$ forms a *consistent concretization set* under the abstract execution represented by *pc*; note that the *pc* constraints also contain the executable interpretations of the external calls. $\boxed{\forall_{\langle \alpha_i, * \rangle \in pc} \exists_{c_i} . \, pc[\langle \alpha_i, * \rangle \rightarrow c_i]}$

The modifications to the baseline algorithm to implement *Deferred Concretization* is highlighted in Algorithm 1. Our algorithm uses a new set, T, to accumulate the states added due to fuzzing (line 19, 25). The statements that need to be handled differently are conditional branching and external operations.

3.2.1 External operation. For $v := extop(w_1, \ldots)$, instead of binding v to the result obtained by invoking *extop*() (say c), we bind v to a *symcrete* value $\langle \gamma, c \rangle$. This achieves two goals:

- we overapproximate the return from the external operation via a fresh symbolic variable γ (regaining coverage);
- we retain the concrete return value from the operation as a witness from a native execution, "hiding" it in the *symcrete* value (to maintain the same path).

Further, we also *upgrade* (any) symbolic arguments of *extop*() to *symcrete*, thereby recording the concrete parameter values used to

$$0 \geq -10 \land 0 = 3 * 0 \land 0 \geq 0 \land 3 = 3 + 0 \land 27 = 216$$
$$(Unsatisfiable!)$$
$$\Uparrow \text{ Logic Solver}$$
$$\langle\alpha_1, 0\rangle \geq -10 \land \langle\alpha_2, 0\rangle = 3 * \langle\alpha_1, 0\rangle \land \langle\alpha_3, 0\rangle = ceil(\langle\alpha_2, 0\rangle) \land \langle\alpha_3, 0\rangle \geq 0 \land$$
$$\langle\alpha_4, 3\rangle = 3 + \langle\alpha_3, 0\rangle \land \langle\alpha_5, 27\rangle = pow(\langle\alpha_4, 3\rangle, 3) \land \langle\alpha_5, 27\rangle = 216$$
$$\Downarrow \text{ Fuzz Solver}$$
$$\alpha_1 \geq -10 \land \alpha_2 = 3 * \alpha_1 \land \alpha_3 = ceil(\alpha_2) \land \alpha_3 \geq 0 \land$$
$$\alpha_4 = 3 + \alpha_3 \land \alpha_5 = pow(\alpha_4, 3) \land \alpha_5 = 216$$
$$(Solution : \alpha_1 = 1, \alpha_2 = 3, \alpha_3 = 3, \alpha_4 = 6, \alpha_5 = 216)$$

Figure 1: Handling of *pc* by solvers

fire the external operation; all concrete arguments remain unaltered (line 34-38). This is required to maintain that the variable map always corresponds to a *consistent concretization set*, and hence, the respective state remains valid.

The consistent tuple corresponding to the return value $\langle\gamma, c\rangle$ and the parameter values $\langle\alpha_1, a_1\rangle, \ldots$ is recorded in the path condition via the constraint Φ. It stays within the path condition as an uninterpreted function (the "executable" interpretations of each external call is available only to the *fuzz solver*).

3.2.2 *Conditional Branching.*

For conditional branches, the engine uses `sevel()` to symbolically evaluate the branch condition; it, then, uses the *logic solver* to check the feasibility of the path condition for both the `true` ($pc \land e$) and `false` ($pc \land \neg e$) outcomes of the branch condition e. The `isSat()` function transforms the pc in the following way before submitting it to the *logic solver*:

(1) All *symcrete* values $\langle\alpha_i, c_i\rangle$ are replaced by their concrete values c_i: this ensures that the formula is evaluated on a *consistent concretization set* on this path;
(2) All terms that contain external operations (appearing as uninterpreted functions) are dropped from the formula as no interpretation of these functions is available to the *logic solver*.

The *logic solver* is, thus, fed an underapproximation of the path condition (to prevent divergence). If the *logic solver* fails to satisfy for any of the branch outcomes, we use the *fuzz solver* to test the feasibility of the path condition. For example, if `isSat()` fails for the `false` side (line 17), we run the *fuzz solver* on ($pc \land \neg e$) to find a chain of new witnesses, i.e. a *consistent concretization set* ξ for this path. If `Fuzz()` is successful in finding such a set of witnesses ξ, the *symcrete* values in the current variable map is updated to hold these new witnesses; else the path is considered infeasible.

Figure 1 shows how the *logic solver* and *fuzz solver* view a *pc*; also, in this case though the *logic solver* finds the *pc* unsatisfiable, the *fuzz solver* could find a solution.

3.3 Fuzz-based Constraint Solver *(Fuzz solver)*

As *logic solvers* cannot handle external functions for which no logical interpretations are available (only executable interpretations are available via native calls), we build a *fuzz-based constraint solver* (or simply *fuzz solver*) to solve these path constraints. Our *fuzz solver* transforms a satisfiability query on a logical formula (queries on path conditions) to a reachability query in a program. It then uses an off-the-shelf fuzzer (AFL [2]) to solve the reachability query.

Figure 3 shows the design of our *fuzz solver*: a query from the SE engine is first filtered through a *unsat predictor* that attempts to answer the query from past history (discussed below). When the predictor guesses the query to be satisfiable, the query passes on to the *constraint compiler*. The *constraint compiler* translates the formula (query) into a C program such that satisfiability on the constraints is answered by a reachability check (simulated by an assertion failure). The generated program is linked with the external library and passed on to a state-of-the-art graybox fuzzer (AFL [2]) to search for the assertion failure. If a failure is found (within a timeout), the *fuzz solver* declares the formula satisfiable, returning the failing test case as the model; otherwise, the formula is declared unsatisfiable. For example, for the following query: $\exists_{c,\alpha_0,\alpha_1,\alpha_2,\alpha_3} \alpha_0 = 45 \land c = strcmp(\alpha_0\alpha_1\alpha_2\alpha_3, \text{``}--\text{"}) \land c = 0$, the *constraint compiler* generates a C code snippet as shown in Listing 3. The fuzzer (AFL) finds a failing test case $[c = 0, \alpha_0 = 45, \alpha_1 = 45, \alpha_2 = 0, \alpha_3 = 0]$, which is returned as a model.
We use some of our domain knowledge to guide the fuzzers, like:

- *(spatial locality) both directions of a branch solve similar constraints:* the (concrete) values on the concretized variables from one arm of a branch is passed on as seed values on the other arm to initiate the search on the *fuzz solver*;
- *(temporal locality) many branch locations have similar outcomes across multiple queries:* we exploit this knowledge to build our *unsat-predictor* that uses the history of *fuzz solver* outcomes for the current branch to *guess* the new outcome (sat/unsat). Our *unsat predictor* is designed similar to a 2-bit branch predictor: it uses a finite-state automata (Figure 4) for each program location. The current outcome is decided by the current state of the predictor; on satisfiable outcomes, the result is validated on the fuzzer, and the fuzzer output is used to update the state machine.
- *there are a large number of potential paths to be explored:* we run the fuzz solver on a tight timeout and use the above *unsat predictor* to return `unsat` quickly. These heuristics reduce the *fuzz solver* times at the cost of missing some paths (introducing loss in coverage); however, we do not lose accuracy as all *sat* outcomes from the predictor are validated by fuzzing.
- *most branches may not require decision on concretized (symcrete) values:* We found that only a few queries require reasoning on values from external calls, and even when it is required, often the current *consistent concretized set* is enough to answer the query. Hence, we use the *logic solver* first, and fall-back to the *fuzz solver* only when it fails.

3.4 Example

Let us run our algorithm on Listing 1 (the path conditions are provided in Table 1). The execution of the program on our algorithm is the same till the external function is hit at line 5. At this point, the engine calls the *logic solver* to get a consistent value for `arg`, say "`-q\0\0`". Then, it invokes the external call `strcmp()` on this value, returning a non-zero value, say `67`. Accordingly, in the variable mapping Ω, it creates *symcrete* values for the return and as well as the parameters, and creates the following bindings:

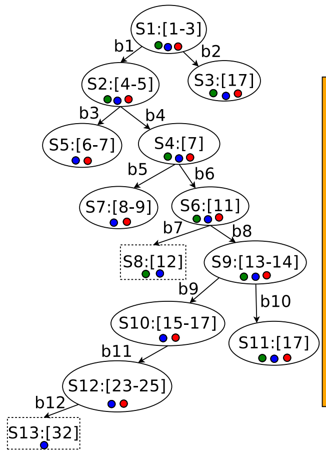


Figure 2: SE tree (incomplete) for Listing 1

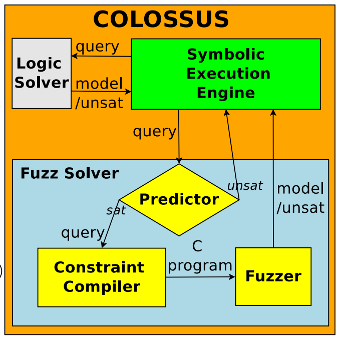


Figure 3: Colossus

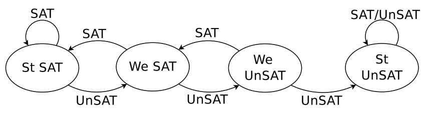


Figure 4: States in fuzz predictor

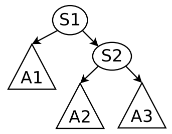


Figure 5: States missed

[`arg[0]` $\rightarrow \langle\alpha_0, 45\rangle$, `arg[1]` $\rightarrow \langle\alpha_1, 113\rangle$, `arg[2]` $\rightarrow \langle\alpha_2, 0\rangle$, `arg[3]` $\rightarrow \langle\alpha_3, 0\rangle$, `r` $\rightarrow \langle\beta_0, 67\rangle$] (46, 113 are ASCII codes of '-', 'q').

In the next line (Line 5), when the engine hits the branching on `r`, it calls the *logic solver* (with the *symcrete* values replaced by their respective witnesses), checking for feasibility [Algo 1, line 15]; for the `true` outcome, it appends the new branch condition to the existing *pc* to form the updated *pc* ($\langle\alpha_0, 45\rangle = 45 \wedge \langle\beta_0, 67\rangle$=0). This *pc* is sent to the *logic solver* (*symcrete* values substituted with their witnesses), thereby creating the constraint ($45 = 45 \wedge 67 = 0$). This turns out to be `false` (indicating that the `false` path must be feasible) [Algo 1, line 21], thereby adding ($\langle\alpha_0, 45\rangle = 45 \wedge \langle\beta_0, 67\rangle \neq 0$, Ω) as a new active state.

Now, it employs *deferred concretization* for the `true` outcome via a call to the *fuzz solver* to solve the following constraint:

$\exists_{c,\alpha_0,\alpha_1,\alpha_2,\alpha_3} \alpha_0 = 45 \wedge c = strcmp(\alpha_0\alpha_1\alpha_2\alpha_3, "--") \wedge c = 0.$

In this case, it should find a possible assignment [$c = 0, \alpha_0 = 45, \alpha_1 = 45, \alpha_2 = 0, \alpha_3 = 0$] that satisfies this constraint. So, it creates a new state by binding the respective variables to *symcrete* values with the corresponding constants, creating a state (6, pc', [$(r = \langle\beta_0, 0\rangle)$, $(arg[0] = \langle\alpha_0, 45\rangle)$, $(arg[1] = \langle\alpha_1, 45\rangle)$, $(arg[2] = \langle\alpha_2, 0\rangle)$, $(arg[3] = \langle\alpha_3, 0\rangle)$])

Our algorithm will also *not raise an false alarm* at line 12 (we omit the detailed analysis for want of space).

4 IMPLEMENTATION AND EVALUATION

Colossus is built on KLEE version 1.3.0 running with STP 2.1.2. Colossus can operate in two modes: *coverage* and *divergence*. The *divergence* mode is described in Algorithm 1. In the *coverage mode*, the returned values from external operations are made *symcrete*, but the argument bindings remain unaltered in the variable map (i.e. symbolic variables are not changed to *symcrete*). The coverage mode avoids calling the fuzz solver on branches involving arguments to external calls; hence, this mode may exhibit false positives. Symbolic execution engines are used both as bug finding and test-generation tools: the *coverage* mode is advisable for bug finding applications where one would like to gain coverage quickly; the *divergence* mode is useful for generating test-suites with low divergence.

We evaluate Colossus on 40 programs from GNU Coreutils-8.29. The experiments were performed on a 3.4 GHz 12 core machine with 32 GB RAM. Each program was run with a 2 hour timeout both for KLEE and Colossus. The fuzz solver was invoked with a 6s timeout.

We compiled the benchmarks with *uclibc* support while holding back the definitions of the string functions. Coverage was computed using *gcov* for the paths on which the tools could complete their execution. We trigger *deferred concretization* only for the C *string library functions* for the following reasons:

- To clearly define the set of functions on which *deferred concretization* was enabled, facilitating future comparisons;
- String functions often produce interesting challenges, like modification of the heap;
- The string library is extensively used in Coreutils.

Note that enabling Colossus for more external functions can only improve our coverage numbers as more paths will be enabled.

We also handle impure functions, like ones that modify the heap (eg. `strcat`, `strcpy`, `stpncpy`, `strchr`, `strncat`, `strncpy`, `strpbrk`, `strrchr`, `strstr`): a wrapper function captures the reachable-heap, passes it as an additional argument to the fuzzer, and patches the changed heap (argument) back (similar to closure-conversion in functional-programs). However, functions (eg. system calls) that may modify the operating system state (eg. signal masks) are beyond our current implementation.

Our experiments were designed to answer the following:

RQ1 Does Colossus derive better coverage than KLEE?
RQ2 Is making the return values symbolic a good solution?
RQ3 What percentage of the symbolic execution tree does KLEE miss due to concretization that Colossus could recover?
RQ4 Were the optimizations on the fuzz solver helpful?
RQ5 Is Colossus able to reduce divergence?
RQ6 What is the tradeoff between improved coverage and reduced divergence?
RQ7 How does the rate of increase in coverage with time for Colossus compare with KLEE?

We use the *coverage* mode of Colossus for RQ1-4 and the *divergence* mode for RQ5-7.

4.1 RQ1: Coverage

Figure 6 shows the comparison of Colossus against KLEE for branch coverage. The red bars refer to unmodified KLEE with default settings. Colossus (the blue bars) improves the coverage significantly for many programs; for instance, it increases the coverage in `cut` from a mere 5.37% to 71.81%; many other programs like `date`,`mkfifo`, `split`, `tr` exhibit an increased coverage by over 27%—improvement by 115%. Overall, Colossus increases average coverage by 15.54% (an improvement of 66.94%) over KLEE across all the programs. In many of the functions where the coverage of

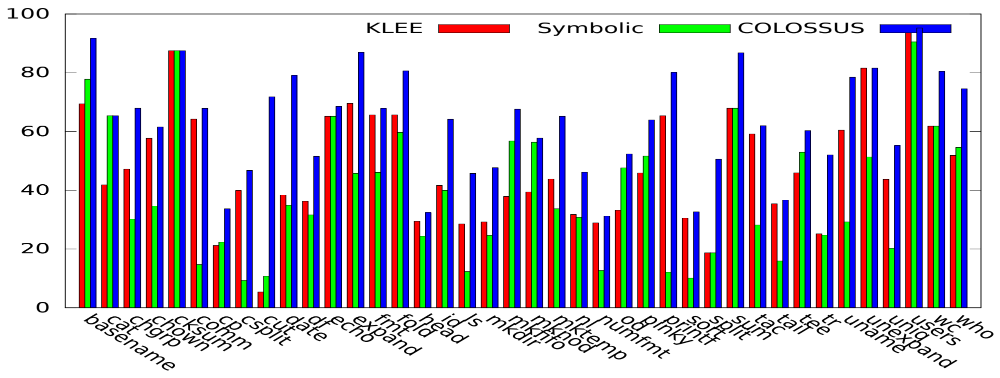


Figure 6: Branch coverage (y-axis) for KLEE, by making return value of missing function symbolic (Symbolic) and Colossus.

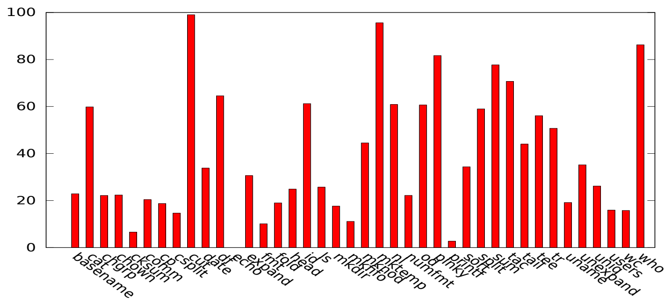


Figure 7: SE tree missed due to concretization (in percent)

KLEE is already high (like `users`) are programs that did not have many string function calls.

Of 40 benchmarks, 12 complete within our 2-hours timeout: average coverage increase of Colossus across these programs is 21% (improvement of 145%); here, the early termination of KLEE shows its inability at exploring available feasible paths. On the rest 28 programs average coverage increases by 13%, showing that the coverage heuristics of symbolic execution engine perform better as Colossus gives them a larger set of paths to pick from.

We have run additional experiments on *ffbench* [14] and *Video-SIMDBench* [30] to support our claim. We summarize the results for each source of concretization:

- external calls and heap-mutation side-effects (66.94% improved coverage in Coreutils [16]),
- non-linear and floating point computations (increases coverage in *ffbench* [14] from 76% to 90%)[1],
- vector instructions (increases coverage in *Video-SIMDBench* [30] from 60.87 to 85.63%).

4.2 RQ2: Comparison with Symbolic Returns

In Figure 6, the green bars (*Symbolic*) refer to a modified version of KLEE where return values from the external operations are overapproximated by fresh symbolic variables. Over *Symbolic*, Colossus increases coverage by as much as 67.97% for `printf`, with an average increase of 24.11% (improvement of 115%) across all the programs. Due to overapproximations, *Symbolic* yields better coverage than KLEE in some cases (like `cat,mkfifo`), but wastes time on infeasible paths in most of the other cases (like `comm,printf`), hence losing out. Colossus beats both these tools, showcasing its ability of *deferred concretization.*

[1]At high coverage levels, it is more challenging to gain incremental coverage

4.3 RQ3: SE Tree Missed by Concretization

We try to estimate the *relative loss in state coverage* for KLEE with respect to Colossus, i.e. in the limit, how much of the complete symbolic execution tree KLEE would miss due to concretizations that Colossus can cover.

For example, say in Figure 5, the complete symbolic execution tree covered by Colossus is $A_1 + A_2 + A_3$ states, and KLEE misses the subtree with A_2 nodes, then the relative loss in state coverage is $\frac{A_2}{A_1+A_2+A_3}$. We measured it by marking all states that appeared on a path that contained at least one branch that opens up via the fuzz solver; these are the states that KLEE would never be able to reach even in the limit. However, note that, within a time budget, loss in state coverage does not directly translate to lower branch coverage because KLEE would be busy exploring other paths in lieu of the states it misses.

Figure 7 shows the percentage loss in state coverage of KLEE: out of 40 programs, 14 have a relative loss in state coverage in excess of 50%. The average relative loss in state coverage in KLEE is 38.60% across all the benchmarks. For the programs `cksum, echo` and `printf` KLEE does not seem to miss much of the symbolic execution tree that Colossus is able to cover. This is because these functions did not have many calls to string functions that Colossus could exploit. One can also see that for these programs, the coverage of both the tools (Figure 6) is almost equivalent.

4.4 RQ4: Fuzz Solver Optimizations

Figure 8 demonstrates the importance of *unsat prediction*: coverage increases by 12.07% on an average over all programs. Our design that filters the queries through the (faster) logic solver before routing it to the fuzz solver also works well: we found that only about 17% of the queries need to be handled by the fuzz solver.

4.5 RQ5: Divergence

For this experiment, we selected programs where KLEE has a coverage of more than 60% in Figure 6. Figure 9(a) shows that, except for `cksum`, Colossus was able to reduce divergence ($\frac{\#\text{paths that diverge}}{\#\text{total paths}}$) to less than 15% in all other cases. For `cksum`, we are still investigating the reason for high divergence; we speculate that the culprit

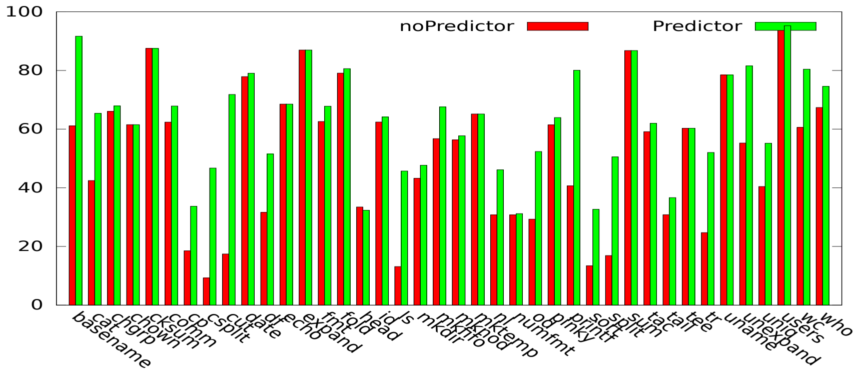
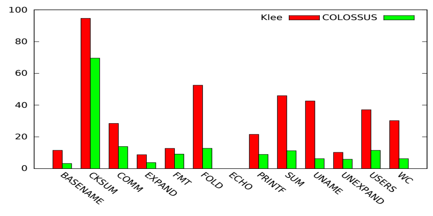


Figure 8: Effect of UnSAT predictor on Branch coverage


(a) Divergence

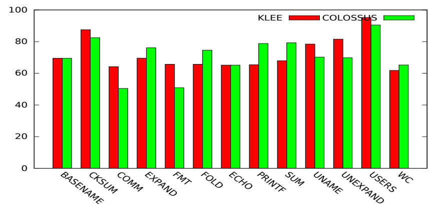

(b) Coverage

Figure 9: Comparison for divergence-mode

for high divergence is some system calls (like `fadvise`()). Overall, Colossus reduces the rate of divergence by more than 18% (improvement of over 55%) over KLEE. However, Figure 9(b) shows that taming divergence is not free, and the tool tends to get sluggish leading to some loss of coverage within a time budget.

4.6 RQ6: Divergence versus Coverage

Figure 10 shows the tradeoff between reduced divergence and improved coverage: at every branch involving *symcrete* values that are bound to arguments of external operations, we choose to invoke the fuzz solver probabilistically, sampling from a Bernoulli distribution with bias 0.0, 0.25, 0.5, 0.75 and 1.0. The plots show the effect of this bias on coverage and divergence: at higher bias values (more fuzzing) we have much less divergence, but the tool moves slower due to the high cost of the fuzz solver, thereby fetching less coverage (within a time budget).

4.7 RQ7: Trend in Coverage

To study the performance in divergence mode, we select programs where KLEE could attain at least 60% coverage (in Figure 6). We only get 12 such programs (Figure 9), of which we randomly show five in Figure 10 and Figure 11; other programs show similar trend.

Figure 11 shows the trend for increase in coverage with time for KLEE and Colossus. These plots attempt to estimate the *coverage space* of the tools and also answer why some of the benchmarks in Figure 9(b) fetch lower coverage. In some cases, though the coverage may be inferior to KLEE at the 2 hr timeout stage (dotted line), on running for another 2 hrs, we find that the coverage increases quickly and generally reach a higher coverage than what KLEE could achieve. In most cases, Colossus continues to gain coverage long after KLEE saturates, clearly showing that Colossus is traversing a much larger coverage space.

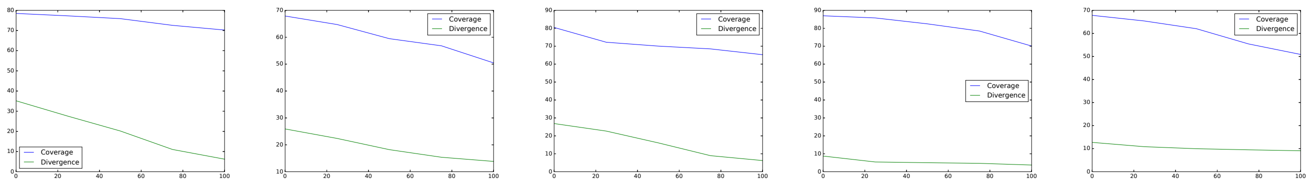
5 RELATED WORK

For concolic execution, Godefroid [17] proposed to solve the problem of incomplete modeling of the path conditions in presence of external operations using uninterpreted functions to represent external operations. They used tests from validity proofs of first-order logic formulas rather than from satisfiability assignments. In the absence of good validity proof generators, they pose their work as a requirement specification for such saturation based solvers. Mixed Concrete-Symbolic Solving [26] use similar ideas for forming the path condition but solve "complicated" path condition by first solving the "simpler" segments and then using iterative solving for repeatedly concretizing to multiple values or making use of user provided (`@Partition`) annotations to generate concretizations.

Dinges et al. [13] propose a solver for solving complex arithmetic path conditions: they define a polytope using the linear constraints, and then sample this polytope using a biased random-walk, guided by a fitness function, to pick the "best" neighbors to visit in the next step of the walk—in a search for a point that satisfies all the non-linear constraints. Unlike our *fuzz solver*, their approach is limited only to concretizations due to (non-linear) arithmetic operations. Further, this technique does not cache the concrete values from previous invocations to the non-linear solver, and thus, often invoke the expensive non-linear solver even when the previous witnesses would have been enough to answer satisfiability. This weakness stems from a weak coupling between the symbolic execution engine and the non-linear solver; we achieve a stronger coupling via the use of *symcrete* values. Finally, the use of off-the-shelf fuzzers (in contrast to developing specialized solvers) allows Colossus to exploit future innovations in fuzzing.

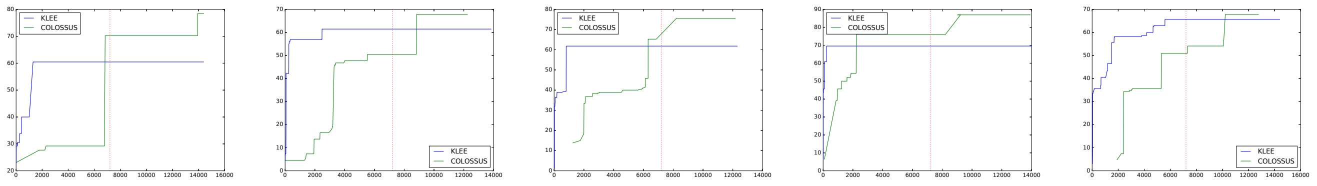
SE engines have seen multiple proposals for coverage heuristics to gain faster coverage. Colossus solves a fundamental problem of symbolic execution (*loss of symbolic states* due to concretization) while a coverage heuristic only prioritizes exploration of available paths. *Deferred concretization* benefits all (EGT-style) SE engines (like Mayhem [9], EXE [7], & S2E [11]) across all heuristics.

Mechtaev et al. [21] specify and solve existential second-order constraints in symbolic execution by posing it as a syntax-guided synthesis [1] problem. Instead of resorting to concretization, the external operations are posed as second-order variables allowing for infeasibility proofs. Though an exciting proposal, it has certain limitations: firstly, it resorts to full-blown synthesis during symbolic execution, questioning scalability at the face of a large number of external operations (not an uncommon scenario). Secondly, the



(a) UNAME (b) COMM (c) WC (d) EXPAND (e) FMT

Figure 10: Divergence and Coverage tradeoffs (y-axis) with *syncrete* arguments fuzzed probabilistically with bias (x-axis)



(a) UNAME (b) COMM (c) WC (d) EXPAND (e) FMT

Figure 11: Trend in increase of coverage (y-axis) with time (x-axis, in seconds) of KLEE and Colossus-*divergence*

infeasibility proofs depend heavily on the user's ability to provide a well-crafted grammar for every second-order variable; a very small grammar can cause loss in coverage and a large grammar will lead to path divergence. Finally, the synthesized specification is still an overapproximation, making the engine traverse multiple infeasible paths and leading to false positives.

The advancement of fuzzing technology and the success of graybox fuzzing tools like AFL [2] has elicited a large number of research proposals attempting to combine the benefits of symbolic execution and fuzzing. Hybrid concolic testing [19] was perhaps the earliest work in this direction: in this technique, the program undergoes random testing till it stops making enough progress (in terms of hitting new coverage goals); it, then, switches to symbolic execution. Once a new uncovered goal is reached, random testing is switched back. The proposal attempts to combine the "deep" coverage of random testing (discover a large number of long program paths quickly) with "wide" coverage of symbolic execution (capture a large variety of program behaviors). Zhang et al. [31] improve upon the idea using the popular graybox fuzzer, AFL, instead of random testing. They use symbolic execution to discover interesting seed inputs for the fuzzer, and the paths explored by the fuzzer that improved coverage are fed back to the symbolic execution engine to explore other, potentially difficult to enter, branches. Driller [29] uses the *angr* [28] SE engine to perform testing of binaries. Driller again attempts to use fuzzing as the main testing machinery and offloads the job to the symbolic execution engine when it is tested with complex reasoning to enter a branch. In contrast to all these work, we attempt to improve symbolic execution by using fuzzing to explore constraints that logic solvers cannot handle. The modern graybox fuzzers are quite powerful and new innovations like AFL-fast [4] are making them more competitive.

6 DISCUSSION

In contrast to concolic testers, EGT engines pose additional challenges as they maintain multiple executions simultaneously. In addition to algorithmic challenges (§2.4), they also pose some engineering challenges:

- EGT engines invoke the constraint solver much larger (potentially an exponentially more) number of times. We apply a number of optimizations (§3.3) to make our solver faster.
- The symbolic and concrete executions are not separated well in EGT engines (in contrast to concolic testers); for instance, EGT engines use a unified variable map for both symbolic and concrete values; we solve this by using *symcrete* values to maintain multiple avatars for concretized variables.

Our algorithm for fuzz-based constraint solving can be seen as the dual of *verification condition generation* [12] wherein programs are translated into logical constraints (handled via SMT solvers). In Colossus, the fuzzer is the *only* obstacle to coverage and cause of divergence; it implies that future improvements in fuzzing will immediately improve Colossus. In other words, the *deferred concretization* algorithm is optimal (in terms of attaining coverage).

Though symbolic execution is sound in the limit, it is generally used as a testing tool to find bugs. The possibility of false positives and diverging test-cases, on the other hand, hurts the usability of such tools for practical applications—*deferred concretization* via fuzzing is a step in the direction towards higher coverage, reduced divergence and improved reproducibility. There do exist threats to validity: as all the results were based on the KLEE infrastructure, our results depend on the precision and correctness of this infrastructure. In particular, the statistics about path divergence needed us to replay the executions back to check its correspondence with an earlier run. In many cases, paths may diverge due to reasons other than concretizations, like change in the environment leading to system calls returning different values, use of random numbers etc. Though we were careful, many of these scenarios were beyond our control. Finally, in terms of choice of benchmarks, we were careful to choose a large set of programs and evaluate multiple aspects of the proposed idea; nevertheless more extensive experiments can be conducted.

REFERENCES

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided Synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE, 1–8.

[2] American Fuzzy Lop (AFL) Fuzzer. (accessed 21-Jan-2018). http://lcamtuf.coredump.cx/afl.

[3] Rohan Bavishi, Awanish Pandey, and Subhajit Roy. 2016. To Be Precise: Regression Aware Debugging. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 897–915. https://doi.org/10.1145/2983990.2984014

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[5] J. Burnim and K. Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, Washington, DC, USA, 443–446. https://doi.org/10.1109/ASE.2008.69

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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)*. USENIX Association, Berkeley, CA, USA, 209–224.

[7] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 10 (Dec. 2008), 38 pages. https://doi.org/10.1145/1455518.1455522

[8] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

[9] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 380–394. https://doi.org/10.1109/SP.2012.31

[10] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 121–130. https://doi.org/10.1145/1985793.1985811

[11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[12] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[13] Peter Dinges and Gul Agha. 2014. Solving complex Path Conditions through Heuristic Search on Induced Polytopes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 425–436.

[14] Floating Point Benchmarks. (accessed 18-Mar-2019). https://www.fourmilab.ch/fbench.

[15] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. 519–531. https://doi.org/10.1007/978-3-540-73368-3_52

[16] GNU Coreutils Program. (accessed 15-Sep-2018). http://sir.unl.edu/portal/index.php.

[17] Patrice Godefroid. 2011. Higher-order Test Generation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 258–269. https://doi.org/10.1145/1993498.1993529

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[19] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 416–426. https://doi.org/10.1109/ICSE.2007.41

[20] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage Testing of Software Patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/2491411.2491438

[21] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic Execution with Existential Second-Order Constraints. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM.

[22] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[23] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781. http://dl.acm.org/citation.cfm?id=2486788.2486890

[24] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. 2017. Bucketing Failing Tests via Symbolic Analysis. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering - Volume 10202*. Springer-Verlag New York, Inc., New York, NY, USA, 43–59. https://doi.org/10.1007/978-3-662-54494-5-3

[25] Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM, New York, NY, USA, 179–180. https://doi.org/10.1145/1858996.1859035

[26] Corina S. Păsăreanu, Neha Rungta, and Willem Visser. 2011. Symbolic Execution with Mixed Concrete-symbolic Solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 34–44. https://doi.org/10.1145/2001420.2001425

[27] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-finding Tools with Deep Faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 224–234. https://doi.org/10.1145/3236024.3236084

[28] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[29] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[30] Video-SIMDBench. (accessed 18-Mar-2019). https://github.com/malvanos/Video-SIMDBench.

[31] Li Zhang and Vrizlynn LL Thing. 2017. A hybrid symbolic execution assisted fuzzing method. In *Region 10 Conference, TENCON 2017-2017 IEEE*. IEEE, 822–825.