Abstract—For software testing, concolic testing reasons about data symbolically but enumerates program paths. The existing concolic technique enumerates paths sequentially, leading to poor branch coverage in limited time. In this paper, we improve concolic testing by bounded model checking (BMC). During concolic testing, we identify program regions that can be encoded by BMC on the fly so that program paths within these regions are checked simultaneously.

We have implemented the new algorithm on top of KLEE and called the new tool LLSPLAT. We have compared LLSPLAT with KLEE using 10 programs from the Windows NT Drivers Simplified and 88 programs from the GNU Coreutils benchmark sets. With 3600 second testing time for each program, LLSPLAT provides on average 13% relative branch coverage improvement on all 10 programs in the Windows drivers set, and on average 16% relative branch coverage improvement on 80 out of 88 programs in the GNU Coreutils set.

I. INTRODUCTION

With the increasing power of computers and advances in constraint solving technologies, an automated dynamic testing technique called concolic testing [1], [2] has received much attention due to its low false positives and high code coverage [3], [4]. Concolic testing runs a program under test with a random input vector. It then generates additional input vectors by analyzing previous execution paths. Specifically, concolic testing selects one of the branches in a previous execution path and generates a new input vector to steer the next execution toward the opposite branch of the selected branch. By carefully selecting branches for the new inputs, concolic testing avoids generating redundant input vectors that execute the same program path, and thus enumerates all non-redundant program paths. In practice, concolic testing suffers from path explosion: it may enumerate exponentially many unique program paths one by one [3]–[6], which often leads to poor branch coverage in limited time.

On the other hand, bounded model checking (BMC) [7]–[10] is a fully symbolic testing technique. Given a program under test and a bound \( k \), BMC unrolls loops and inlines function calls \( k \) times to construct an acyclic program which is an under-approximation of the original program. It then performs verification condition (VC) generation over the acyclic program to obtain a formula which encodes the acyclic program and a property to check. The formula is then fed into a SAT solver. If the formula is proved to be valid by the solver, the property holds. Otherwise, the solver provides a model from which we can extract an execution of the program that violates the property. BMC provides a way to encode and reason about multiple execution paths simultaneously using a single formula, but its scalability is often limited by deterministic dependencies between program paths and data values.

A natural question is whether there is a way to improve concolic testing by BMC to alleviate path explosion and thus improve branch coverage? In this paper, we provide a positive answer and propose a concolic+BMC algorithm. Intuitively, given a program under test, the algorithm starts with the per-path search mode in concolic testing while referring to the control flow graph (CFG) of the program to identify easy-to-analyze portions of code that do not contain loops, recursive function calls, or other instructions that are difficult to generate formulas using BMC. Whenever a concolic execution encounters such a portion, the algorithm switches to the BMC mode and generates a BMC formula for the portion, and identifies a frontier of hard-to-analyze instructions. The BMC formula summarizes the effects of all execution paths through the easy-to-analyze portion up to the hard frontier. When the concolic execution reaches the frontier, the algorithm switches back to the per-path search mode to handle the cases that are difficult to summarize by BMC.

We have implemented the concolic+BMC algorithm on top of KLEE [11] and called the new tool LLSPLAT. We have compared LLSPLAT with KLEE, using 10 programs from the Windows NT Drivers Simplified [12] and 88 programs from the GNU Coreutils used in [11]. With 3600 second testing time for each program, LLSPLAT provides on average 13% relative branch coverage improvement on the programs in the Windows NT drivers simplified set, and on average 16% relative branch coverage improvement on 80 out of 88 programs in the GNU Coreutils set.

The rest of the paper is organised as follows. Section 2 provides a motivating example. Section 3 reviews concolic testing. Section 4 describes the concolic+BMC algorithm. Section 5 presents experimental results. Section 6 shows related work. Section 7 concludes this paper.
II. A MOTIVATING EXAMPLE

We illustrate the inadequacy of concolic testing, and the benefits of using BMC to improve concolic testing, using the function `foo` below. The function runs in an infinite loop, and receives two inputs in each iteration. One input `c` is a character and the other input `s` is a character array. The function `foo` reaches the label `L` if the variable `state` is 9, and the input array `s` holds the string “reset”. From the label `L`, there is a huge chunk of code consisting conditionals, loops, and procedure calls. Similar functions like `foo` are often generated by lexers.

Concolic testing systematically explores all execution paths of the function. Since the function `foo` runs in an infinite loop, the number of distinct feasible executions is infinite. To perform concolic testing we need to bound the number of iterations of the loop if we perform a depth-first search of the execution paths. There are 17 possible choices of values of `c` and `s` that concolic testing would consider, and at least 9 iterations are required to reach the label `L`. Hence, concolic testing will explore about $17^9 \approx 10^{11}$ execution paths. It is unlikely that concolic testing can explore a path that reaches the label `L` and executes the code below the label `L` in a reasonable time budget. We confirm this fact by testing the function `foo` using KLEE [11]. It could not reach the label `L` in a day, which led to poor branch coverage.

It is worth mentioning that, if there were buggy code after the label `L`, the situation would get even worse because concolic testing cannot reveal the bugs efficiently.

```c
void foo() {
    char c, s[6];
    int state = 0;

    while(1) {
        // Some dummy code
        c = input(); s = input();

        if (c == 't' && state == 0) state = 1;
        if (c == 't' && state == 1) state = 2;
        if (c == 'a' && state == 2) state = 3;
        if (c == 'a' && state == 3) state = 4;
        if (c == 'x' && state == 4) state = 5;
        if (c == 'x' && state == 5) state = 6;
        if (c == 't' && state == 6) state = 7;
        if (c == 't' && state == 7) state = 8;
    }
    // A large chunk of code below.
}
```

In our concolic+BMC approach, whenever a concolic execution encounters a conditional, it has a choice either to save a predicate representing that a particular branch is taken along the execution as concolic testing does, or to save a BMC formula, for example, that encodes the entire conditional. Which choice is taken depends on whether the conditional is “simple” enough to generate a BMC formula easily. For example, a conditional

```c
P ::= (var g)^∗ · Fn^+
Fn ::= f((var p))^∗ · (var l)^∗ · BB^+
BB ::= Inst · TermInst
Inst ::= x ← e | f(e^∗) | x ← input()
TermInst ::= ret | br e BB1 BB2 | br BB | ERROR
```

is simple if there are no loops and recursive function calls\(^1\) inside it. Since all conditionals are simple in function `foo`, the concolic+BMC approach can easily generate a feasible path that reaches the label `L` and thus greatly shorten the time to reach the subsequent uncovered paths. We validated this fact by using LLSPLAT to test function `foo`. LLSPLAT could generate paths that reached the label `L` in 3s, which led to better branch coverage.

III. CONCOLIC TESTING

We first review the traditional concolic testing algorithm with depth-first path searching strategy, and then describe how LLSPLAT modifies it.

A. Program Model

We describe how concolic testing works on a simple language shown in Figure 1. A program consists of a set of global variables and a set of functions. Each function consists of a name, a sequence of formal parameters, a set of local variables, and a set of basic blocks representing the control flow graph (CFG) of the function. Each basic block consists of a list of instructions followed by a terminating instruction. There are three types of instructions: `x ← e` is an assignment, `f(e^∗)` is a function call, and `x ← input()` indicates that the variable `x` is a program input. There are four types of terminating instructions: `ret` is a return instruction, `br e BB1 BB2` is a conditional branch, `br BB` is an unconditional branch, and `ERROR` indicates program abortion. We omit an explicit syntax of expressions. We assume there is an entry function `main` that is not called anywhere. We also assume each function has an `entry` basic block, and every basic block of the function is reachable from it.

B. The Concolic Testing Algorithm

To test a program `P`, concolic testing tries to explore all execution paths of `P`. It first instruments the program `P` by Algo 1, and outputs an instrumented program `P'`. The red-highlighted lines(gray with monochrome printers) in the algorithms can be ignored for now because they are used in the concolic+BMC approach we describe later. Algo 2 repeatedly runs the instrumented program `P'`. Due to limited space, we omit the instrumentation for function calls, and the code that bounds the search depth in the search strategy — these are identical to previous work [1], [2].

Algo 1 first makes a copy `P''` of the program `P`, and inserts various global variables and function calls which are used\(^1\)The program size after function inlining can be exponentially larger than the size of the original program.

128
for the symbolic execution. It then returns the instrumented program $P'$. Algorithm 3 presents the definitions of the instrumented functions. The expressions enclosed in double quotes ("e") represent syntactic objects. We denote $\&x$ to be the address of a variable $x$.

**Algorithm 1: Instrumentation**

Program instrument($P$):

$P' \leftarrow P$

Add to $P'$ global vars $i \leftarrow 0$, inputNo $\leftarrow 0$, symStore $\leftarrow \emptyset$, pathC $\leftarrow \emptyset$

Gows $\leftarrow \{BB \mid BB$ is a governor in $P\}$

Add to $P'$ global vars

$bmcNo \leftarrow 0$, currGov $\leftarrow None$, init $\leftarrow None$

foreach $BB \in P'$ do

if $BB \in GR(gov)$ for some $gov \in Gows$ then continue

foreach $Inst \in BB$ do

switch $Inst$ do

case $x \leftarrow input()$ do

Replace $Inst$ by $InitInput("x")$

case $x \leftarrow e$ do

Add updateSymStore("x", "e") before $Inst$

case $br \leftarrow BB1$ $BB2$ do

if $BB \in Goes$ then

Add startBMC($BB$) before $Inst$

foreach $d \in Dests(BB)$ do

Add endBMC($BB$, $d$) as the 1st instruction of $d$

else

Add addPathConstraint("e", e) before $Inst$

case $Return$ do

if $Inst$ is in the main function then

Add solveConstraint() before $Inst$

case $ERROR$ do

Add print("ERROR found") before $Inst$

return $P'$


**Algorithm 2: run_llsplat**

void run_llsplat($P$):

$I \leftarrow []$; branch_hist $\leftarrow []$; completed $\leftarrow false$

$CFG_{P} \leftarrow CFGofProgram(P)$

while $\neg completed$ do execute instrument($P$)


**Algorithm 3: Concolic Testing**

void $InitInput("x")$: $inputNo \leftarrow inputNo + 1$

void $addPathConstraint("e", b)$:

if $b$ then

pathC[i] $\leftarrow symexpr("e")$

else

pathC[i] $\leftarrow \neg symexpr("e")$

if $i < \neg br_hist[i]$ then

if $i = \neg br_hist[i] - 1$ then

$br_hist[i].done \leftarrow true$

else

$br_hist[i]$ $\leftarrow BNode(isCovered : b, done : false)$

void $SolveConstraint()$:

while $j \geq 0$ do

if $\neg br_hist[j].done$ then

foreach $d$ such that $\neg br_hist[j].isCovered[d]$ do

if $\forall c \leq i \leq j \exists pathC[k] \land$ $\land rmLastDest(pathC[c]) \land V_{c \in Edges_{d}[j]} \land$ $c$ has a solution $I'$ then

$br_hist \leftarrow \neg br_hist[0..j]

$I \leftarrow I'$

return

else

$br_hist[j].isCovered$ $\leftarrow \neg br_hist[j].isCovered$

$\neg pathC[j]$ $\leftarrow \neg pathC[j]$

if $pathC[0..j]$ has a solution $I'$ then

$br_hist \leftarrow \neg br_hist[0..j]

$I \leftarrow I'$

return

else

$j = j - 1$

if $j < 0$ then completed $\leftarrow true$


**IV. COMBINING CONCOLIC TESTING WITH BMC**

Recall that the goal of our work is that, whenever a concolic run starts with some inputs, instead of using the path constraint to encode a single execution path, we plan to add BMC formulas to the path constraint so that it may encode (potentially exponentially) many execution paths. Note that
the path constraint is used by a constraint solver to decide
new runs in concolic testing. When we attempt to cover an
uncovered instruction in the new run, the path constraint that
encodes multiple execution paths enables the constraint solver
to search for multiple paths leading to the instruction, instead
of one path in the traditional concolic testing.

To achieve the goal, given a program under test, we identify
regions of the program for generating BMC formulas. A region
of a program is a subgraph of the control flow graph of the
program. We list the following requirements for identifying
regions.

1) A region must be acyclic. It is required by any BMC
procedure.

2) A region should not have function calls. It is desired
because function inlining is required before generating
BMC formula but the resulting BMC formula may be
exponentially large in the size of the input program, which
we want to avoid.

3) A region should be sufficiently large so that the cor-
responding BMC formula covers more paths and fully
exploits the power of a modern constraint solver. Most
of the path constraints in those multiple paths are the
same except for the constraints within the BMC regions,
thus solving one concolic+BMC constraints representing
multiple paths may take less time than solving multiple
independent paths.

In addition, given a region, it is also required that a desired
BMC generation procedure be compatible with concolic testing.
Unlike generating a formula for an entire program in existing
BMC tools, we need to generate ones for regions of a program
which lead to some specific issues, which BMC procedures in
existing BMC tools need not and cannot handle. For example,
a natural question would be that, after adding a BMC formula
to the path constraint, how the symbolic store needs to be
updated so that concolic testing proceeds.

We now present the concolic+BMC algorithm. The key ob-
ervation is that given a program \( P \) under test, the instrumented
program for \( P \) can additionally refer to the (static) CFG of
\( P \) and perform static analysis at run time. Section IV-A describes
how to identify program portions for BMC formula generation.
Section IV-B describes the BMC formula generation algorithm. Section IV-C integrates this with concolic testing.

A. Identifying Program Portions for BMC

  a) Preliminaries: Given a CFG, a basic block \( m \) domi-
nates a basic block \( n \) if every path from the entry basic block
of the CFG to \( n \) goes through \( m \). We denote \( \text{Dom}(m) \) to
be the set of basic blocks which \( m \) dominates. A depth-first
search of the CFG forms a depth-first spanning tree (DFST).
There are edges in CFG that go from a basic block \( m \) to an
ancestor of \( n \) in DFST (possibly to \( m \) itself). We call these
dependent paths. By [13], a directed graph is acyclic if a
depth-first search yields no back edge.

  b) Governors, Governed Regions, and Destinations:
Given a basic block \( m \), a basic block \( n \in \text{Dom}(m) \) is polluted
in \( \text{Dom}(m) \) in the following four cases: (1) \( n \) contains function
call instructions, (2) \( n \) has no successors, (3) \( n \) is the source
or the target of a back edge, or (4) \( n \) is reachable from a
polluted basic block \( k \in \text{Dom}(m) \). A basic block \( m \) effectively
dominaates a basic block \( n \) if \( n \in \text{Dom}(m) \) and \( n \) is not
polluted in \( \text{Dom}(m) \). We denote \( \text{Edom}(m) \) to be the set of basic
blocks that \( m \) effectively dominates.

A basic block \( m \) is called a governor candidate if (1) the
terminating instruction of \( m \) is of the form \( \text{br} \in B1 B2 \),
(2) \( m \) dominates both \( B1 \) and \( B2 \), and (3) \( \text{Edom}(B1) \)
and \( \text{Edom}(B2) \) are not empty. Given a governor candidate
\( m \) with its two successors \( B1 \) and \( B2 \), the governed region
of \( m \), denoted by \( GR(m) \), is \( \text{Edom}(B1) \cup \text{Edom}(B2) \).
A basic block \( n \) is a destination of \( GR(m) \) if \( n \notin GR(m) \)
and \( n \) is a successor of some basic block \( k \in GR(m) \). Let the set
\( \text{Dests}(m) \) be all destinations of \( GR(m) \). A basic block \( gov \)
is a governor if \( gov \) is a governor candidate, and there is no
governor candidate \( m \) with \( gov \in GR(m) \). For any governor
\( gov \), \( GR(gov) \) is acyclic and does not have function calls, and
gov dominates every basic block \( BB \in GR(gov) \).

c) Example: Consider the program in Fig 2a. \( BB0 \) is a
governor. Its governed region \( GR(BB0) \) includes \( BB1 \), \( BB2 \),
\( BB4 \), \( BB5 \), and \( BB6 \), which are inside the red dash circle. \( BB3 \)
and \( BB7 \) are the destinations in \( \text{Dests}(BB0) \). Though \( BB2 \)
is a governor candidate, it is not a governor because it is
in \( GR(BB0) \).

![Fig. 2: An Example](image)

(a) A program under test
(b) Variable renaming

B. Translating Governed Regions to BMC Formulas

A governed region is ideal for generating a BMC formula
because it is acyclic, does not have function calls, and is
“sufficiently” large in the sense that it includes as many
(unpolluted) basic blocks as its governor governs. We present
our algorithm that translates a governed region to a BMC
formula, and provide an example.

1) The BMC Formula Generation Algorithm: Given a
governor \( gov \), we construct a BMC formula \( \phi \) for \( GR(gov) \)
in five steps:

1) Renaming variables in \( GR(gov) \) into an SSA-form. Since
\( GR(gov) \) is acyclic, there exists a topological ordering
over the basic blocks in \( GR(gov) \). Let \( \text{AccVars} \) be the
set of variables accessed by the instructions in \( GR(gov) \),
and let a version map \( V \) be a map from each variable
Let \( x \in AccVars \) to a variable \( x_\alpha \) with a version \( \alpha \in \mathbb{N} \). We assign the version number \( \alpha \) to each variable in \( GR(gov) \) for renaming according to the topological order.

2) Create a boolean variable \( g_{BB} \) for each basic block \( BB \in GR(gov) \), which is called an execution guard. Our intention is that, if \( g_{BB} \) is true, then an execution represented by a model of the final BMC formula \( \phi \) goes through \( BB \). Otherwise, it does not goes through \( BB \).

3) Compute an edge map \( Edges \) that maps each basic block \( BB \in GR(gov) \cup Dests(gov) \) to a list of edge formulas. Each entry in the list represents the condition of hitting an incoming edge of a basic block. For each \( BB \in GR(gov) \), if its terminating instruction is \( br eBB1 BB2 \), then we add \( g_{BB} \land e \) to \( Edges[BB1] \), and add \( g_{BB} \land \neg e \) to \( Edges[BB2] \); if it is \( br BB1 \), then we add \( g_{BB} \) to \( Edges[BB1] \). If the terminating instruction belongs to a governor, we insert the condition pair \( e \) and \( \neg e \) to the corresponding \( Edges \) map directly. Let the governor's terminating instruction be \( br eBB1 BB2 \). Let \( e_0 \) be an expression obtained by replacing each variable \( x \) in \( e \) with \( x_0 \). We set \( Edges[BB1] = e_0 \) and \( Edges[BB2] = \neg e_0 \).

4) Compute a block map \( Blks \) that maps each basic block \( BB \in GR(gov) \) to a block formula. For each \( BB \in GR(gov) \), let \( I_1, I_2, \ldots, I_k \) be the non-terminating instructions in \( BB \). For each \( 1 \leq i \leq k \), if \( I_i \) is \( x_\alpha \leftarrow e \), we define an instruction formula \( c_i \) to be \( x_\alpha = \tau e(g_{BB}, e, x_\alpha - 1) \). We set \( Blks[BB] = \bigwedge_{1 \leq i \leq k} c_i \).

5) Create the final BMC formula \( \phi \), defined as follows:

\[
\bigwedge_{BB \in GR(gov)} \left( g_{BB} = \bigvee_{c \in Edges[BB]} c \right) \land Blks[BB]
\]

Intuitively, \( \phi \) claims that for each basic block \( BB \in GR(gov) \), (1) \( BB \) is taken (i.e., \( g_{BB} \) is true) if one of its predecessor is taken, and (2) the block formula of \( BB \) must hold.

Our BMC formula generation algorithm has the following important property. The proof is in our technical report\(^2\).

**Theorem IV.1.** Let \( gov \) be a governor and \( T \) be an arbitrary topological ordering over \( GR(gov) \). After the BMC algorithm is done w.r.t. \( T \), for any destination \( d \in Dests(gov) \), (1) the formula \( \phi \land \bigvee_{c \in Edges[BB]} c \) encodes all executions from \( gov \) to \( d \), and (2) for every execution \( \rho \) from \( gov \) to \( d \), the final version of each variable \( x \) in \( \phi \) represents the value of \( x \) when \( \rho \) enters \( d \).

\(^2\)http://zilongwang.github.io/papers/scam16_tr.pdf

When \( \rho \) enters the destination \( BB_3 \), the largest version of \( x \) and \( y \) along \( \rho \) is \( x_3 \) and \( y_0 \), but their final versions in \( \phi \) are \( x_4 \) and \( y_1 \). However, since \( BB_2, BB_4, BB_5 \) and \( BB_6 \) are not taken along \( \rho \), we have \( x_4 = x_3, x_2 = x_1 = x_0 \), and \( y_1 = y_0 \). Since \( BB_1 \) is taken, we have \( x_3 = x_2 - y_0 \). Thus \( x_4 = x_0 - y_0 \) and \( y_1 = y_0 \). We conclude that \( x_4 \) and \( y_1 \) represent the values of \( x \) and \( y \) when \( \rho \) enters the destination \( BB_3 \).

C. Integrating BMC Formulas with Concolic Testing

To integrate BMC with concolic testing, we need to instrument function calls for BMC encoding, obtain program CFG for performing static analysis, and consider the way to update concolic data structures such as the worklist, the symbolic store and the branch history. Thus we add the red lines in Algo 1, 2, and 3 to reflect the integration.

Once we meet a branch instruction, we need to check whether this instruction is a good candidate to perform BMC. During the instantiation in Algo 1, we first compute a set \( Govs \) of all governors of the program \( P \). Since basic blocks in governed regions are used to generate BMC formulas, we skip instrumenting them. When a basic block \( BB \) has two successors, if \( BB \) is a governor, we instrument a function call \( startBMC(BB) \) before \( BB \)’s terminating instruction, and for each destination \( d \) in \( Dests(BB) \), we instrument a function call \( endBMC(BB, d) \) as the first instruction of \( d \). If \( BB \) is not a governor, we perform the old instrumentation in concolic testing.

BMC formulas generation in our algorithm requires static analysis result of a program under test. In Algo 2, we read the CFG of the uninstrumented program \( P \). This CFG is used to generate BMC formulas along concolic executions.

Since a BMC region may contain multiple destinations, we need to instrument one \( startBMC(gov) \) call and multiple \( endBMC(gov, d) \) calls into the program to define a BMC governed region.

The definition of \( startBMC(gov) \) is given in Algo 4. It saves the governor \( gov \) that will be used to generate a BMC formula using \( currGov \). Then it increments \( bmcNo \), which records the number of BMC formulas that have been generated so far along the concolic execution. It then uses \( init \) to “glue” the execution before entering \( GR(gov) \) with the BMC formula for \( GR(gov) \). More concretely, for each variable \( x \in AccVars(gov) \), an equation \( x_{bmcNo} = symStore[\&x] \) is created, and \( init \) is the conjunction of all such equations.

The definition of \( endBMC(gov, d) \) is also given in Algo 4. If the passed-in governor \( gov \) is the one saved in \( currGov \), it performs the BMC generation algorithm described in

### TABLE I: Edge formulas and block formulas

<table>
<thead>
<tr>
<th>BB</th>
<th>Edges[BB]</th>
<th>Blks[BB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB1</td>
<td>{( x_0 &gt; y_0 )}</td>
<td>( x_0 = \tau e(g_{BB1}, x_2 = y_0, x_2) )</td>
</tr>
<tr>
<td>BB2</td>
<td>{( \neg(x_0 &gt; y_0))}</td>
<td>( x_1 = \tau e(g_{BB2}, y_0 - x_0, x_0) )</td>
</tr>
<tr>
<td>BB3</td>
<td>{( g_{BB1}, g_{BB6} \land y_1 \neq 9 } }</td>
<td>( x_4 = \tau e(g_{BB4}, y_0 - x_3, x_3) )</td>
</tr>
<tr>
<td>BB4</td>
<td>{( g_{BB2} \land x_1 &gt; y_0 )}</td>
<td>( x_2 = \tau e(g_{BB5}, y_0 - x_1, x_1) )</td>
</tr>
<tr>
<td>BB5</td>
<td>{( g_{BB2} \land \neg(x_1 &gt; y_0))}</td>
<td>( y_1 = \tau e(g_{BB6}, x_4, y_4) )</td>
</tr>
<tr>
<td>BB6</td>
<td>{( g_{BB4}, g_{BB5} \land x_2 \neq 0 )}</td>
<td>( y_2 = \tau e(g_{BB5}, y_0 - x_1, x_1) )</td>
</tr>
<tr>
<td>BB7</td>
<td>{( g_{BB5} \land \neg(x_2 \neq 0), g_{BB6} \land \neg(y_1 \neq 9) }</td>
<td>( y_3 = \tau e(g_{BB6}, x_4, y_4) )</td>
</tr>
</tbody>
</table>
Section IV-B to obtain a BMC formula $\phi$ for the governed region $GR(gov)$, the final version map $V_{final}$, and the edge map $Edges$.

The coverage history $branch\_hist$ is updated in $endBMC(gov, d)$. We extend $branch\_hist$ to be a list of $BranchNode \cup BmcNode$. A BmcNode has three fields: $is\_covered$ records which destinations have been covered in prior runs, $Edges\_d$ maps each destination to its edge formulas, and $done$ records whether all destinations have been covered in prior runs.

We then start to integrate BMC formulas into concolic testing. Given a formula $\psi$ and a number $j$, we denote $addSup(\psi, j)$ to be the formula obtained by replacing each variable $x$ in $\psi$ with a new variable $x'$. We first create a formula $\phi \land \bigwedge_{c \in Ends[dest]} c, bmcNo$ which represents all executions from the governor $gov$ to the destination $d$ by Theorem IV.1. Since the governed region may be reached multiple times along an execution, we compute a formula $\psi \equiv addSup(\phi \land \bigwedge_{c \in Ends[dest]} c, bmcNo)$ which specifies that $\psi$ is the $bmcNo$-th BMC formula along the execution. We then add $init \land \psi$ to the path constraint. Finally, to let the concolic execution proceed, for each variable $x \in AccVars(gov)$, we update the symbolic store so that $symStore[kx]$ represents the value of $x$ when the execution enters the destination $d$.

The function $SolveConstraint$ is extended as shown in Alg 3. If the node $branch\_hist[i]$ is a BmcNode, we find an uncovered destination $d$, and asks if there is an execution that goes to $d$. The formula $rmLastDest(pathC[i])$ is defined by removing the disjunction of edge formulas of $d'$ from pathC[i] where $d'$ is the destination covered by the just terminating execution. If there are new inputs $I'$ for such an execution to $d$, a new run is started with inputs $I'$.

1) Example: We again reuse the example in Fig 2a. Suppose LLSPLAT randomly generates $x = 10$ and $y = 5$ in the first run. When the run terminates, the path constraint is of size 1, and $pathC[0] = init \land \phi \land \psi$, defined as follows. Note that the superscript 1 of the variables in $pathC[0]$ represents that it is the first BMC formula generated along the run. The symbolic variables $sym1$ and $sym2$ are created for $x$ and $y$ when $InitInput("x")$ and $InitInput("y")$ are called.

$$init \equiv x_0^1 = sym1 \land y_0^1 = sym2$$

$$\phi \equiv \left[ \begin{array}{c} g_{BB1} = x_0^1 > y_0^1 \land \\ g_{BB2} = \neg (x_0^1 > y_0^1) \land \\ g_{BB4} = (g_{BB2} \land x_0^1 > y_0^1) \land \\ g_{BB5} = (g_{BB2} \land \neg (x_0^1 > y_0^1)) \land \\ g_{BB6} = (g_{BB4} \lor (g_{BB5} \land g_{BB2} \neq 0)) \\ x_0^1 = ite(g_{BB2}, x_0^1 > y_0^1, x_0^1) \land \\ x_0^1 = ite(g_{BB2}, y_0^1 \land \neg (x_0^1 > y_0^1), x_0^1) \land \\ y_0^1 \equiv \neg (g_{BB6} \land y_0^1 \neq 9) \\ \right] \land \neg \psi_d$$

The coverage history $branch\_hist$ is of size one. $branch\_hist[0]$ is a BmcNode defined below:

$$branch\_hist[0].is\_covered = [BB3 \Leftarrow true, BB7 \Leftarrow false]$$
$$branch\_hist[0].done = false$$
$$branch\_hist[0].Edges\_d = [BB3 \Rightarrow \{g_{BB1}, g_{BB6} \land y_0^1 \neq 9\}, BB7 \Rightarrow \{g_{BB5} \land \neg (x_0^1 \neq 0), g_{BB6} \land \neg (y_0^1 \neq 9)\}]$$

Now LLSPLAT searches for new inputs for the next run. Since $BB7$ is the only uncovered destination based on $branch\_hist[0].is\_covered$, LLSPLAT solves the formula $init \land \phi \land \bigwedge_{c \in branch\_hist[0].Edges\_d[BB7]} c$, that is, LLSPLAT tries to find a feasible execution path that leads to $BB7$ containing ERROR. Note that there are three execution paths to $BB7$, and the formula encodes all.

V. Experiments

We have built our tool LLSPLAT to implement the concolic+BMC algorithm on top of KLEE (LLVM version 3.4). To verify whether the algorithm can increase branch coverage of concolic testing in practice, we designed our experiments to compare the branch coverage between LLSPLAT and KLEE.

A. Experiment Settings

We chose two sets of benchmarks to perform the comparison. The first benchmark set is the Windows NT Drivers Simplified set containing 10 C programs from [12], and the second benchmark set is the GNU Coreutils tested in [11] that contains 88 C programs. Each program in Windows NT Drivers Simplified set ranges between 2344 and 6444 lines of LLVM-IR code, and that of the GNU Coreutils set contains approximately 200,000 lines of code in LLVM-IR level per benchmark including library code. All the experiments were
performed on a 2 core Intel Xeon E5-2667 v2 CPU machine with 256GB memory and 64-bit Linux (Debian/Jessie). Both LLSPAT and KLEE ran with original KLEE arguments. We conducted the experiments 10 times and then calculated the average coverage because the coverage depends on the initial random input vector.

All the benchmarks in the Windows NT Drivers Simplified set are tested by calling LLSPAT and KLEE with maximum run time of 3600s and maximum memory of 1600MB. All the benchmarks in the GNU Coreutils set are tested using the following command:

```
./<tool-name> --libc=uclibc
--posix-runtime
--no-output
--max-memory=1600
--max-time=3600
./file_under_test
--sym-args 1 10 2
--sym-files 2 8
```

Using these options, we ran each benchmark that contained a minimum of 1 argument, and a maximum 10 of arguments with each argument containing at most 2 characters for 3600s.

### B. Experimental Results

Table II shows the branch coverage of all 10 benchmarks in the Windows NT Drivers Simplified benchmark set. For example, KLEE covered 270 branches while LLSPAT covered 319 branches with relative branch coverage improvement of 18.14% on `cdaudio_s1_f`. Relative branch coverage improvement is defined by the difference between the branch coverage of LLSPAT and the branch coverage of KLEE divided by the branch coverage of KLEE.

The results in Table II show that LLSPAT achieves better branch coverage than KLEE for all 10 benchmarks in the set with 13% relative branch coverage improvement. We conclude that the concolic+BMC algorithm improves the branch coverage of this benchmark set.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>KLEE Branch covered</th>
<th>KLEE Time(s)</th>
<th>LLSPAT Branch covered</th>
<th>LLSPAT Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cdaudio_s1_f</code></td>
<td>270</td>
<td>2.26</td>
<td>319</td>
<td>61.06</td>
</tr>
<tr>
<td><code>cdaudio_s1_t</code></td>
<td>268</td>
<td>2.47</td>
<td>317</td>
<td>61.46</td>
</tr>
<tr>
<td><code>diskperf_s1_t</code></td>
<td>114</td>
<td>3614.35</td>
<td>132</td>
<td>3602.78</td>
</tr>
<tr>
<td><code>floppy_s3_f</code></td>
<td>142</td>
<td>1.55</td>
<td>156 (+9.85%)</td>
<td>20.92</td>
</tr>
<tr>
<td><code>floppy_s3_t</code></td>
<td>142</td>
<td>1.57</td>
<td>155 (+9.15%)</td>
<td>22.36</td>
</tr>
<tr>
<td><code>floppy_s4_f</code></td>
<td>224</td>
<td>3.69</td>
<td>238 (+6.25%)</td>
<td>31.99</td>
</tr>
<tr>
<td><code>floppy_s4_t</code></td>
<td>224</td>
<td>3.57</td>
<td>236 (+5.35%)</td>
<td>33.80</td>
</tr>
<tr>
<td><code>kblitrl_s1_t</code></td>
<td>93</td>
<td>0.42</td>
<td>111 (+19.35%)</td>
<td>1.62</td>
</tr>
<tr>
<td><code>kblitrl_s2_f</code></td>
<td>159</td>
<td>1.11</td>
<td>181 (+13.83%)</td>
<td>4.98</td>
</tr>
<tr>
<td><code>kblitrl_s2_t</code></td>
<td>159</td>
<td>1.11</td>
<td>182 (+14.47%)</td>
<td>4.58</td>
</tr>
</tbody>
</table>

**TABLE II**: Branch coverage comparison between LLSPAT and KLEE on the Windows NT Drivers Simplified

The programs in the GNU Coreutils benchmark set are larger than that of the Windows NT Driver Simplified benchmark set. We would like to evaluate them to check whether LLSPAT can still achieve higher branch coverage than KLEE.

Figure 3 shows the result of relative branch coverage improvement for all 88 programs from the GNU Coreutils benchmark set where each bar represents relative branch coverage improvement of one benchmark. A bar above zero indicates by how much LLSPAT wins over KLEE; a bar below shows the opposite. Bars are sorted in ascending order.

The result shows that LLSPAT outperforms KLEE in terms of branch coverage on most of the benchmarks: 80 out of 88 benchmarks tested with LLSPAT have higher branch coverage than KLEE. Average improvement for all benchmarks is 13% and 16% for the 80 improved benchmarks. Among all the benchmarks 65 have more than 10% increases. Table III provides detailed branch coverage information of 10 randomly selected benchmarks. The number of encoded BMC governed regions is also provided in Table III to show the involvement of BMC encoding in this benchmark set. Note that a governed region can be encoded multiple times since it can be reached multiple times during one single execution path.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>KLEE</th>
<th>LLSPAT</th>
<th>Number of governed regions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cspill</code></td>
<td>1015</td>
<td>955 (+12.78%)</td>
<td>54059</td>
</tr>
<tr>
<td><code>chown</code></td>
<td>883</td>
<td>1000 (+13.25%)</td>
<td>55485</td>
</tr>
<tr>
<td><code>shred</code></td>
<td>734</td>
<td>762 (+3.81%)</td>
<td>85194</td>
</tr>
<tr>
<td><code>dd</code></td>
<td>647</td>
<td>811 (+25.34%)</td>
<td>38815</td>
</tr>
<tr>
<td><code>cut</code></td>
<td>651</td>
<td>728 (+11.82%)</td>
<td>39396</td>
</tr>
<tr>
<td><code>echo</code></td>
<td>258</td>
<td>301 (+16.67%)</td>
<td>108079</td>
</tr>
<tr>
<td><code>uniq</code></td>
<td>686</td>
<td>700 (+12.4%)</td>
<td>44827</td>
</tr>
<tr>
<td><code>link</code></td>
<td>431</td>
<td>518 (+20.1%)</td>
<td>79621</td>
</tr>
<tr>
<td><code>nice</code></td>
<td>481</td>
<td>581 (+20.7%)</td>
<td>94622</td>
</tr>
<tr>
<td><code>df</code></td>
<td>844</td>
<td>907 (+7.46%)</td>
<td>81981</td>
</tr>
</tbody>
</table>

**TABLE III**: Branch coverage comparison between LLSPAT and KLEE and the number of encoded governed regions on 10 randomly selected benchmarks from the GNU Coreutils

A natural question that arises is when LLSPAT’s branch coverage exceeds KLEE’s result. We compute the crossing time from which LLSPAT always outperforms KLEE on
the improved benchmarks from the GNU Coreutils. More concretely, for each benchmark, we analyzed a graph that describes how branch coverage evolves in 3600s using LSPLAT and KLEE, and recorded the time when the branch coverage reported by LSPLAT is always higher than the one reported by KLEE. Table IV shows the result statistics. The crossing time of 46 and 68 benchmarks is below 60s and 120s, respectively. This fact implies that our method is preferable over KLEE on these benchmarks given a tight time budget.

<table>
<thead>
<tr>
<th>Crossing Time(s)</th>
<th>0-60</th>
<th>60-120</th>
<th>120-180</th>
<th>&gt;180</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Benchmarks</td>
<td>46</td>
<td>22</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

TABLE IV: Crossing time statistics of improved benchmarks on the GNU Coreutils

In conclusion, the concolic+BMC algorithm has a benefit of increasing branch coverage of all 10 programs in the Windows NT Drivers Simplified benchmark set, and 80 out of 88 programs in the GNU Coreutils benchmark set.

C. Threats to Experiment Validity

We identified the following threats to the validity of our experiment:

- **Test benchmarks used in the experiment may not be representative of all programs.** We chose the Windows NT Drivers Simplified set from [12] since three programs in this benchmark set are used in [14] to evaluate context-guided concolic testing. Thus we evaluated all programs in this benchmark set. To make our benchmark selection more unbiased, we chose 88 programs from the GNU Coreutils benchmark set used in [11] to evaluate KLEE. Even though we used 98 C programs with diverse sources, they may not be representative of all programs.
- **More LLVM-IR types may yield different results.** The current implementation of LSPLAT only deals with LLVM-IR of non-complex constant expression type and non-floating type during the BMC encoding procedure. The experimental results might be different with more other type of LLVM-IR instructions.
- **Large governed region size may yield different results.** In the GNU coreutils experiment, there are 8 benchmarks where LSPLAT performed worse than KLEE. We observe the BMC formulas in those benchmarks are complex. Thus the STP solver used in LSPLAT and KLEE might not be smart enough to solve them efficiently. Under such cases avoiding to encode complex governed regions or breaking complex regions into a few small regions could help improve the result. It remains as our future work to identify BMC formulas of acceptable complexity to avoid solving hard formulas.

VI. RELATED WORK

a) **Concolic Testing:** Several approaches analyze states (i.e., path constraint and symbolic store) maintained by concolic testing so as to explore the search space efficiently. Godefroid [5] introduced compositional concolic testing. The work was later expended to do compositional concolic testing on demand [15]. The main idea is to generate function summaries for an analyzed function based on the path constraint, and to reuse them if the function is called again with similar arguments. Instead of computing dynamic underapproximations of summaries, we compute exact summaries of governed regions using the static representation of the CFG. Kuznetsov et al. [16] introduced the dynamic state-merging (DSM) technique. DSM maintains a history queue of states. Two states may merge (depending on a separate and independent heuristic for SMT query difficulty) if they coincide in the history queue. Our concolic+BMC approach is different because we do not analyze the states to merge execution paths.

Moreover, several approaches combine other testing techniques with concolic testing together. Majumdar and Sen introduced hybrid concolic testing [17] that combines random testing and concolic testing. Boonstoppel et al. proposed RWSet [18], a path pruning technique identifying redundant execution paths based on similarity of their live variables. Jaffar et al. [19] used interpolation to subsume execution paths that are guaranteed not to hit a buggy location. Agerino et al. [20] combined static data-flow program analysis techniques with concolic testing. Santelices et al. [21] introduced a technique that merges multiple execution paths based on the control dependency graph of a program.

Recently there are some other trials on the combination use of concolic testing and model checking. Daca et al. [22] proposed an approach combining model checking and concolic testing whose framework is similar to [17]. It ran concolic testing first to generate test cases. When it failed to meet certain goals, it switched to model checking to prove path feasibility to reduce searching space. Jaffar et al. [23] used conditional model checking to construct a residual program that is fed into a concolic testing tool to reduce testing effort. Gonzalez-de-Aledo et al. [24] exploited static analysis to zoom into potential bug candidates and concolic execution to confirm these bugs. Our method is different from these methods because our model checking targets at the loop-free fragment of the code such that we can encode multiple paths on the fly to alleviate path explosion and increase branch coverage.

Several heuristic-based approaches have been proposed to guide an execution toward a specific branch. CREST [25] introduced four search strategies, as already shown in the experiments. SAGE [26] introduced generational search that selects all the branches in an execution path and generates a set of inputs. Xie et al. [27] proposed a fitness-guided search that calculates fitness values of execution paths to guide the next execution towards a specific branch. Li et al. [28] proposed a subpath-guided search which steers symbolic execution to less traveled paths. Seo and Kim [14] introduced context-guided search that selects branches in a new context to help prevent the continuous selecting of the same branch. KLEE [11] used a meta-strategy which combines several search strategies in a round robin fashion to avoid cases where one strategy gets stuck. Conceptually, our concolic+BMC approach is compatible with the above search heuristics. In essence, concolic testing is
to cover both branches of a conditional, which can be regarded as two destinations. Then the goal of concolic+BMC is to cover all destinations in a program.

b) Bounded Model Checking: VC generation approaches in modern BMC tools can be classified into two categories. The first one is based on weakest preconditions [29] by performing a demand-driven backward analysis from the points of interest [30]–[33]. The other one encodes a program in a forward manner, such as CBMC [2], ESBMC [2], and LLBMC [11]. We are inspired by the VC generation algorithm of CBMC, and thus conceptually it is the closest work to our BMC algorithm. The VC generation of CBMC differs from ours in four ways. First, though CBMC also does variable renaming, it does it using a fixed order of basic blocks. We relax this requirement and prove that any topological order works for variable renaming. This is important to us, because we do not have to follow the fixed order CBMC uses. In fact, we use the reverse post order of a governed region as our topological order for variable renaming because it has been computed during the construction of depth first spanning tree which identifies back edges. We save the computation time in this way. Secondly, though the VC generation of CBMC also computes edge formulas for each basic block in a given acyclic program, all predecessors of the basic block contribute to deriving edge formulas. However, this is not the case in ours. For example, suppose that gov is governor, d is a destination of GR(gov), and there is a predecessor BB ∈ GR(gov) of d. This case may happen because BB is polluted. Then our BMC algorithm does not derive an edge formula from BB for d. Thirdly, CBMC does not have the notion of destinations. Since a governed region may have multiple destinations, it is not clear that no matter which destination is chosen, whether the final version of variables in the formula \( \phi \) that encodes the governed region always represents the value of the variables when the destination is reached. We prove this fact in our paper. Lastly, since CBMC encodes the entire program, it does not identify acyclic portions of a program using the notions such as governors. It also does function inlining and loop unrolling, which we do not.

ESBMC follows the VC generation algorithm of CBMC. It extends BMC to check concurrent programs. LLBMC explicitly models the memory as a variable representing an array of bytes, which requires LLBMC to distinguish if a little-endian or big-endian architecture is analyzed. They are orthogonal to LLSPLAT.

c) Software Model Checking: Large-block encoding [34] is widely used in software model checkers. It encodes control flow edges into one formula, for computing the abstract successor during predicate abstraction. Selective enumeration using SAT solvers [35] and symbolic encodings for program regions, e.g., to summarize loops [36], have been successfully exploited in software model checking.

VII. CONCLUSION

Since concolic testing suffers from path explosion, we introduce the concolic+BMC algorithm that applies BMC locally targeting at loop-free code fragment during concolic testing to alleviate path explosion, and thus improve branch coverage. Our experiments show that the concolic+BMC algorithm increases branch coverage of the two test benchmark sets. It is also worth mentioning that, if we notice that there is a BMC region containing a potential error after concolic+BMC, and we want to check exactly which branch in the region leads to the error, we can always set that region to do concolic execution while keeping the remaining regions performing BMC.

We believe some future work can be achieved on top of our algorithm. Specifically, the implementation of LLSPLAT performs BMC encoding if there exists an acyclic graph containing a few merging basic blocks without function calls inside of the graph. We avoid encoding function calls because it may incur exponential blowup in the BMC formula generation. We would like to come up with a clever evaluation procedure that identifies “cheap” function calls that can be encoded. In addition, solving functions of large scale may yield long time being spent in the SMT solver. We would also like to investigate whether there exists a low cost governed region overhead estimation method to make the selection of BMC regions more intelligent.

REFERENCES


