Scaling Up Symbolic Analysis by Removing Z-Equivalent States

YUEQI LI and S. C. CHEUNG, Hong Kong University of Science and Technology
XIANGYU ZHANG, Purdue University
YEPANG LIU, Hong Kong University of Science and Technology

Path explosion is a major issue in applying path-sensitive symbolic analysis to large programs. We observe that many symbolic states generated by the symbolic analysis of a procedure are indistinguishable to its callers. It is, therefore, possible to keep only one state from each set of equivalent symbolic states without affecting the analysis result. Based on this observation, we propose an equivalence relation called z-equivalence, which is weaker than logical equivalence, to relate a large number of z-equivalent states. We prove that z-equivalence is strong enough to guarantee that paths to be traversed by the symbolic analysis of two z-equivalent states are identical, giving the same solutions to satisfiability and validity queries. We propose a sound linear algorithm to detect z-equivalence. Our experiments show that the symbolic analysis that leverages z-equivalence is able to achieve more than ten orders of magnitude reduction in terms of search space. The reduction significantly alleviates the path explosion problem, enabling us to apply symbolic analysis in large programs such as Hadoop and Linux Kernel.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Algorithms, Verification, Performance, Experimentation

Additional Key Words and Phrases: Symbolic analysis, path explosion, state equivalence detection

ACM Reference Format:
Yueqi Li, S. C. Cheung, Xiangyu Zhang, and Yepang Liu. 2014. Scaling up symbolic analysis by removing z-equivalent states. ACM Trans. Softw. Eng. Methodol. 23, 4, Article 34 (August 2014), 32 pages.
DOI: http://dx.doi.org/10.1145/2652484

1. INTRODUCTION

Symbolic analysis [King 1976; Clark 1976], such as that in KLEE [Cadar et al. 2008], has a wide range of applications [Cadar et al. 2008; Chandra et al. 2009; Godefroid 2007; Kothari et al. 2008; Person et al. 2008; Xie et al. 2003; Sen et al. 2005; Cui et al. 2013]. It executes a program using symbolic states and may proceed with both branches constrained by different symbolic path conditions, unlike concrete executions that proceed with only one branch upon a conditional statement. In such cases, two subsequent symbolic states are forked from the state before the conditional statement, which can be further explored separately. Essentially, the analysis maintains one symbolic state along each individual path, and users can query the computed symbolic states to check any properties of interest. Compared with constraint-based static analyses that encode abstract states along multiple paths to one logical formulae [Babic et al. 2007; Babic

This research is supported by the Research Grants Council under General Research Fund 611912 of Hong Kong, as well as by the US National Science Foundation (NSF) under grants D917007, 1218993, and 1320326. Any opinions, fundings, conclusions, or recommendations are solely those of the authors.
Y. Li, S. C. Cheung, and Y. Liu are currently affiliated with the Department of Computer Science and Engineering at HKUST. X. Zhang is currently affiliated with the Department of Computer Science at Purdue University.
Corresponding author: Yueqi Li; Hong Kong University of Science and Technology; email: yueqili@cse.ust.hk.

DOI: http://dx.doi.org/10.1145/2652484

```
Input: An initial state (l, true, s),
w: = {(l, true, s)}
while w ≠ ∅
    (l, pc, s): = pick(w)
    switch instruction(l)
      case v: = e                          //assignment
        w: = w ∪ {(succ(l), pc, s[v → eval(s, e)])}
      case if (e), goto l'                 //conditional jump
        if ( follow( pc ∧ eval(s, e)) )
          w: = w ∪ {(l', pc ∧ e, s)}
        if (follow( pc ∧ eval(s, ¬e)) )
          w: = w ∪ {(succ(l), pc ∧ ¬e, s)}
       ...
        ...
  ...
```

Fig. 1. General symbolic analysis algorithm.

and Hu 2008; Dillig et al. 2008], the per-path encoding in symbolic analysis leads to simpler formulae. This is friendlier to the underlying constraint solver and has better precision in reasoning, especially in handling heap-related states. It is hence widely used in test generation [Cadar et al. 2008; Godefroid 2007; Godefroid et al. 2005] and evidence-based bug finding [Chandra et al. 2009; Chaudhuri and Foster 2010].

Figure 1 describes a general algorithm popularly adopted by existing symbolic analysis tools [Cadar et al. 2008; Godefroid et al. 2005]. The algorithm uses a work list w to maintain a set of states (paths) to explore. A symbolic state consists of a program counter, path conditions, and a symbolic store. The state selector function pick() determines the next symbolic state to explore. The branch selector function follow() prunes infeasible paths and sometimes feasible paths that are irrelevant to the properties of interest, to alleviate path explosion. As shown in Figure 1, the algorithm first populates the work list with an initial state and then iteratively selects one state to explore in its main loop. It also handles assignments by updating the corresponding symbolic store. Upon reaching a conditional statement, it may fork into two states if both branches are feasible.

Despite its precision, such path-sensitive symbolic analyses often face the scalability issue due to the exponential growth of states [Kuznetsov et al. 2012]. Hence, even though there is some success in using them to analyze small programs, it is still difficult to apply them to large programs, such as Linux Kernel.

In this article, we address the problem by pruning equivalent states. We observe that large real-world programs such as Tomcat, Hadoop, and Linux Kernel heavily depend on various external entities, including the programs' external libraries, runtime environments, underlying computer networks, and peer processes running on the same computer. These programs interact with their external entities via the corresponding APIs.

To handle these API functions, one approach is to manually construct models. This approach entails substantial human effort and is even infeasible in many cases. For example, the default models provided by KLEE are only sufficient for analyzing coreutil programs. In order to analyze Linux Kernel, we have spent a few months modeling a large set of necessary APIs (e.g., memory management APIs). However, there are still 5,000+ undocumented assembly functions. Yet, even when one can model those assembly functions, it is still hard to model external inputs from unknown servers/devices and external data sources. Analyzing large Java programs presents similar challenges, and they usually rely heavily on Java's standard libraries. Modeling about 5,000 native APIs in JDK is difficult, and the APIs tend to evolve over time, making manual modeling more challenging. Besides, the source code of many libraries is often not available

[Qi et al. 2012]. Even for symbolic execution engines that can handle binaries, such as S2E [Chipounov et al. 2012], the implementations of library calls are difficult to handle as they are often highly optimized, extensively using caching, hashing, and bit level operations [Qi et al. 2012].

Hence, the approach used by many recent static analyses [Babic and Hu 2008; Babic et al. 2007; Chandra et al. 2009; Chaudhuri and Foster 2010; Dillig et al. 2008; Kothari et al. 2008] is to treat these difficult-to-model API functions as uninterpreted functions that can bind to any possible values allowed by their types. However, in the context of analyzing large programs, the sheer number of these functions is one of the main causes of symbolic state explosion.

We observe that such treatment of API functions induces significant state redundancy. When we symbolically analyze a procedure directly or transitively involving uninterpreted functions, we observe that many symbolic states are equivalent in answering path feasibility, property satisfiability, and validity queries because of the flexible binding assumed by the uninterpreted functions. This observation motivates us to define an equivalence relation called z-equivalence between a pair of symbolic states. This relation is weaker than logical equivalence but strong enough to relate two relatively indistinguishable symbolic states when they are indistinguishable with respect to the continuation of the symbolic analysis.

We prove that the subsequent symbolic analysis of two z-equivalent states traverses the same set of paths and gives the same answers to validity and satisfiability queries. Utilizing the results, we only need to analyze one state in a set of z-equivalent states and avoid redundant path exploration from these states. This reduction considerably alleviates the path explosion problem, enabling us to perform symbolic analysis on large programs such as the Hadoop and Linux Kernel. However, the problem of z-equivalence detection is undecidable, and even a less-general form of it is exponential in complexity, as discussed in Section 3.2. Detection of all z-equivalent states is intractable. We therefore propose a linear-complexity algorithm for detecting an effective subset of z-equivalent states. We evaluate the effectiveness by applying our technique to the Ant, Lucene, Hadoop and Linux Kernel. By removing the redundant z-equivalent states detected, we observe up to ten orders of magnitude reduction in terms of search space.

2. MOTIVATING EXAMPLE

We use a motivating example in Figure 2 to illustrate our idea. We consider readUserInput(), *cin*.hasNext(), *cin*.nextInt(), *s*.hasNext(), and *s*.nextInt() in readData() as external APIs. These APIs are highlighted with "*" in Figure 2. We treat them as *uninterpreted functions* such that their return values can bind to any value confined by their types.

If the symbolic analysis algorithm expands the loops in the procedure readData() twice, it explores a total of six paths in readData(). When the algorithm finishes analyzing readData(), each of the six paths leads to a symbolic state, which is characterized by a constraint constructed by the conjunction of the associated path conditions and readData()'s postcondition. In the following constraints, x_n, w_n, y_n, and z_n denote the return values from the nth invocations of *cin*.hasNext(), *s*.hasNext(), *cin*.nextInt(), and *s*.nextInt() made in a path, respectively. The constraints are presented as follows.

$$
\begin{aligned}
&C1\text{: } {}'F' = type \wedge \neg x_1 \wedge RET = 0,\\
&C2\text{: } {}'F' = type \wedge x_1 \wedge \neg x_2 \wedge RET = y_1,\\
&C3\text{: } {}'F' = type \wedge x_1 \wedge x_2 \wedge \neg x_3 \wedge RET = y_1 + y_2,\\
&C4\text{: } {}'F' \neq type \wedge \neg w_1 \wedge RET = 0,\\
&C5\text{: } {}'F' \neq type \wedge w_1 \wedge \neg w_2 \wedge RET = z_1,\\
&C6\text{: } {}'F' \neq type \wedge w_1 \wedge w_2 \wedge \neg w_3 \wedge RET = z_1 + z_2.
\end{aligned}
$$

```
1
2   int readData(char type){
3     int sum=0;
4     if('F'==type){
5       Scanner cin=new Scanner(FileReader("input"));
6       while(cin.hasNext()){     // cin.hasNext()*
7         sum += cin.nextInt();   // cin.nextInt() *
8       }
9       cin.close();
10    }else{
11      Socket s=new Socket("1.1.1.1");
12      while(s.hasNext()){       // hasNext() *
13        sum += s.nextInt();     // nextInt() *
14      }
15      s.close();
16    }
17    return sum;
18  }
19  int readAndNoti(){
20      int type=readUserInput(); //readUserInput() *
21      int s= readData(type);
22      if( s>=0 )
23              print("Zero or Positive");
24       else
25              print("negative");
26      return s;
27  }
28  void main(){
29    int  rawInput=readAndNoti();
30    assert(rawInput>=0);
31    …
32  }
```

Fig. 2. Motivating example.

$C1$–$C3$ encode the paths corresponding to a true branch of line 3, with a different number of iterations of the inside loop. Note that the constraint variables are local to a constraint. Therefore, x_1 may bind to different values in $C2$ and $C3$. A conventional symbolic execution engine would consider these six constraints to be different and would analyze each of them separately. In fact, this is unnecessary if we take advantage of the uninterpreted functions in two ways. First, some clauses in these constraints do not affect the symbolic analysis of the code outside readData(). For example, the conjunction $w_1 \wedge w_2 \wedge \neg w_3$ in the constraint $C6$ does not affect the analysis outside the scope of readData(). This is because the conjunction does not constrain any variables outside the scope of readData() directly, or indirectly, by constraining other correlated variables. Second, some expressions have equivalent effects. For example, the clauses $RET = y_1$ and $RET = y_1 + y_2$ have equivalent effects on the subsequent analysis after a symbolic execution engine finishes analyzing readData(). The reason is that we can always choose some value for y_1 in the constraint $C2$ to make y_1 equal to $y_1' + y_2$ in the constraint $C3$. Here, we rename variable y_1 to y_1' in $C3$ to resolve the name conflict. Similarly, we can also choose values for y_1' and y_2 to make $y_1' + y_2$ in $C3$ equal to y_1 in $C2$. From these observations, we consider $C2$ and $C3$ equivalent. Similarly, $C5$ and $C6$ are equivalent. Thus, upon the return from readData() in Line 20, we only need to continue analyzing paths associated with $C1$, $C2$, $C4$, and $C5$. With these four constraints, we have eight paths to analyze in readAndNoti() and two of them are infeasible. The eight paths are characterized by the following constraints, in which $RET2$ denotes the return from readAndNoti().

$$
\begin{aligned}
C7\text{: } & 'F' = type \wedge \neg x_1 \wedge RET = 0 \wedge RET = RET2 \wedge RET2 < 0,\\
C8\text{: } & 'F' = type \wedge x_1 \wedge \neg x_2 \wedge RET = y_1 \wedge RET = RET2 \wedge RET2 < 0,\\
C9\text{: } & 'F' = type \wedge \neg x_1 \wedge RET = 0 \wedge RET = RET2 \wedge RET2 \geq 0,\\
C10\text{: } & 'F' = type \wedge x_1 \wedge \neg x_2 \wedge RET = y_1 \wedge RET = RET2 \wedge RET2 \geq 0,\\
C11\text{: } & 'F' \neq type \wedge \neg w_1 \wedge RET = 0 \wedge RET = RET2 \wedge RET2 < 0,\\
C12\text{: } & 'F' \neq type \wedge w_1 \wedge \neg w_2 \wedge RET = z_1 \wedge RET = RET2 \wedge RET2 < 0,\\
C13\text{: } & 'F' \neq type \wedge \neg w_1 \wedge RET = 0 \wedge RET = RET2 \wedge RET2 \geq 0,\\
C14\text{: } & 'F' \neq type \wedge w_1 \wedge \neg w_2 \wedge RET = z_1 \wedge RET = RET2 \wedge RET2 \geq 0.
\end{aligned}
$$

Constraints $C7$ and $C11$ are not satisfiable. We can again conduct a similar analysis for readAndNoti() to identify equivalent constraints. Observe that the variable *type* is not used in the caller of readAndNoti(). Nor does it impose any constraints on variables in the scope of readAndNoti()'s caller. Thus, its value is irrelevant to the symbolic analysis of the caller of readAndNoti(). If we ignore clauses *'F'* $=$ *type* and *'F'* $\neq$ *type*, we can consider $C8$ and $C12$, $C9$ and $C13$, and $C10$ and $C14$ equivalent. As a result, it suffices to keep the three paths corresponding to $C8$, $C9$, and $C10$ for further analysis. Note that this step of reduction is dependent on the previous step because variable s in readAndNoti() has different symbolic expressions and can propagate out of the function body. If we only ignore *'F'* $=$ *type* and *'F'* $\neq$ *type* as the existing approaches [Boonstoppel et al. 2008] do, the resulting symbolic states are still different.

In this motivating example, we have intuitively shown that uninterpreted functions (e.g., w_1) and variables dependent on these functions (e.g., *type*) can induce equivalent symbolic states. The identification of these states can significantly reduce the number of paths to explore. Note that such redundancy is present across multiple queries to the solver. It is hence difficult for the solver to leverage its internal optimization mechanism to remove the redundancy. The complex heap behavior of large programs also makes merging multiple paths to a single formula (hoping the redundancy can be suppressed by merging) an impractical design choice. It was observed that such formulae encoding many paths of the entire body of a large program cannot be solved by the underlying solvers [Cadar et al. 2008]. That is also the reason why many symbolic analysis engines that aim to have precise simulation of heap behavior choose to explore individual paths. More discussion can be found in Section 5.5.

In fact, as an on-the-fly constraint optimization method external to the solver, our technique is orthogonal to many existing techniques that optimize path exploration and constraint merging. For example, we can still deploy efficient data structures with copy-on-write techniques to optimize data storage [Cadar et al. 2008], and perform constraint merging [Kuznetsov et al. 2012] to combine constraints (after redundancy removal) from multiple states to one.

As shown in the preceding example, equivalent state detection can be made in three steps. The first step (in Section 3.1) is to encode symbolic states in terms of constraint expressions. The second step (in Section 3.3) is to identify those expressions in constraints that can flexibly bind to arbitrary values, for example, the expressions y_1 and $y_1 + y_2$ in the preceding example can bind to any integer values. We illustrate the challenges of decoupling variables when they can constrain each other in different constraint expressions. For instance, y_1 can be constrained if there is another constraint $y_1 > 0$ which prevents y_1 from binding to negative values. A sound algorithm is devised to accomplish such decoupling efficiently. The third step (in Section 3.4) is to unify the expressions as much as possible, for example, y_1 and $y_1 + y_2$ can be unified.

3. APPROACH

In this section, we first introduce a symbolic analysis algorithm augmented with equivalent state detection. Then we define z-equivalence and explain the detection of z-equivalent states.

3.1. Symbolic Analysis

Like the symbolic analysis engine in KLEE [Cadar et al. 2008] (as shown in Figure 1), our analysis maintains a symbolic state for individual paths. The improvement lies in the fact that we leverage symbolic variables to achieve state reduction at function call boundaries.

$$
\begin{array}{rcl}
v \in SymValue & & \rho \in Constraint \\
\sigma \in SymStore & ::= & Variable \mapsto SymValue \\
\omega \in Worklist & ::= & \mathcal{P}(Stmt \times Constraint \times SymStore) \\
\Omega \in WLStack & ::= & \overline{Worklist} \\
\gamma \in RetConstraint & ::= & \mathcal{P}(Constraint) \\
\Gamma \in RCStack & ::= & \overline{RetConstraint}
\end{array}
$$

$\sigma\downarrow ::= \bigwedge_{x\in\sigma} x = \sigma(x)$, the operator turns a store to a logical conjunction.

$\langle s,\rho,\sigma\rangle \bowtie_x^f \gamma \ ::= \bigcup_{\rho'\in\gamma}\{\langle s, \rho\wedge\rho', \sigma[x\mapsto rt_f]\rangle\}$, the operator propagates the constraints collected in the callee to the caller for an invocation $x=f(e)$; s is the statement right after the invocation. rt_f represents the return value of f.

select(ω) chooses the next symbolic state $\langle s,\rho,\sigma\rangle$ to explore.

follow(ρ) determines the feasibility of a path condition ρ and realizes some user specified path pruning heuristics.

eq_test(γ, B, ρ) takes a set γ of return constraints generated so far, the set of observable variables B, and a newly generated constraint ρ, and determines if ρ is redundant. It returns {} if redundant, { ρ} otherwise.

Fig. 3. Definitions related to the semantic rules.

Let us facilitate our discussion using a small imperative language, which models integer and Boolean types of constants, assignment, conditional statements, function calls, and returns. Our implementation supports complex language features, like heap modifications, pointers, and arrays. External functions are explicitly modeled by unknown().

Program	$p ::= m^*$
Function	$m ::= f(x)\{s\}$
Constant	$c ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid \textbf{true} \mid \textbf{false}$
Expression	$e ::= x \mid c \mid e_1 \textbf{op} e_2$
Statement	$s ::= x = e \mid x = f(e) \mid x = \textbf{unknown}() \mid s_1; s_2 \mid$ $\textbf{if } (e) \textbf{ then } s_1 \textbf{ else } s_2 \mid \textbf{ret}^f x$
Operators	$\textbf{op} ::= + \mid - \mid \ldots \mid > \mid \geq \mid = \mid \neq \mid \wedge \mid \vee \mid \ldots$

Figure 3 presents the definitions related to the semantic rules. In particular, the symbolic store σ maps a variable to a symbolic value; the work list ω contains a set of triples that encodes the symbolic states to explore. Each triple contains the next statement to explore, the path constraint, and the symbolic store for that path. Upon a function call from A() to B(), our analysis proceeds to analyze the paths inside B(), computing the various possible symbolic return values and pruning redundant constraints (or states) before it continues to execute the statements after the call in A(). Hence, we need to use a stack to retain the caller's state when analyzing the callee. In particular, Ω represents a stack of work lists, each of which corresponds to one level of function invocation. This is similar to the call stack in a concrete execution. A return constraint Y is a set of constraints generated for the return value of a function, encoding the symbolic values yielded along different paths in the function. Γ represents a stack of return constraints for nesting functions.

The semantic rules are presented in Figure 4. We have three sets of rules. The expression rules evaluate an expression to a symbolic value. Note that it evaluates a variable directly to its symbolic form instead of acquiring its value from the symbolic store (Rule E-VAR in Figure 4).

EXPRESSION RULES $\boxed{e \twoheadrightarrow v}$

$$\frac{}{c \twoheadrightarrow c} \quad \text{(E-CONST)} \qquad \frac{}{x \twoheadrightarrow x} \quad \text{(E-VAR)}$$

$$\frac{e_1 \twoheadrightarrow v_1 \quad e_2 \twoheadrightarrow v_2}{e_1 \mathbf{op}\ e_2 \twoheadrightarrow v_1 \boldsymbol{op}\ v_2} \quad \text{(E-OP)}$$

STATEMENT RULES $\boxed{\langle s, \rho, \sigma\rangle \rightarrow \mathcal{P}(\langle s, \rho, \sigma\rangle)}$

$$\frac{e \twoheadrightarrow \rho_0}{\langle x = e; s,\ \rho, \sigma\rangle \rightarrow \{\langle s, \rho, \sigma[x \mapsto \rho_0]\rangle\}} \quad \text{(S-ASSIGN)}$$

$$\frac{e \twoheadrightarrow v \qquad \rho_1 = \rho \wedge v \qquad \rho_2 = \rho \wedge \neg v \qquad follow(\rho_1 \wedge \sigma\downarrow) = \mathbf{true} \quad follow(\rho_2 \wedge \sigma\downarrow) = \mathbf{false}}{\langle \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s,\ \rho, \sigma\rangle \rightarrow \{\langle s_1; s, \rho_1, \sigma\rangle\}} \quad \text{(S-IF-T)}$$

$$\frac{e \twoheadrightarrow v \qquad \rho_1 = \rho \wedge v \qquad \rho_2 = \rho \wedge \neg v \qquad follow(\rho_1 \wedge \sigma\downarrow) = \boldsymbol{true} \quad follow(\rho_2 \wedge \sigma\downarrow) = \boldsymbol{true}}{\langle \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s,\ \rho, \pi, \sigma\rangle \rightarrow \{\langle s_1; s, \rho_1, \sigma\rangle, \langle s_2; s, \rho_2, \sigma\rangle\}} \quad \text{(S-IF-BOTH)}$$

$$\frac{r \text{ a fresh symbolic variable}}{\langle x = \mathbf{unknown}(); s,\ \rho, \sigma\rangle \rightarrow \{\langle s, \rho, \sigma[x \mapsto r]\rangle\}} \quad \text{(S-UNINTPRT)}$$

GLOBAL RULES $\boxed{\langle \Omega, \Gamma\rangle \Rightarrow \langle \Omega', \Gamma'\rangle}$

$$\frac{\langle s, \rho, \sigma\rangle = select(\omega) \quad \langle s, \rho, \sigma\rangle \rightarrow t}{\langle \Omega \circ \omega, \Gamma\rangle \Rightarrow \langle \Omega \circ \omega[t/\langle s, \rho, \sigma\rangle], \Gamma\rangle} \quad \text{(G-STMT)}$$

$$\frac{\langle y = f(e); s_0, \rho, \sigma\rangle = select(\omega) \quad f(x)\{s\} \text{ is a method} \quad e \twoheadrightarrow v \qquad \omega' = \{\langle s, \rho, \sigma[x \mapsto v\ ]\rangle\}}{\langle \Omega \circ \omega, \Gamma\rangle \Rightarrow \langle \Omega \circ \omega \circ \omega', \Gamma \circ \{\}\rangle} \quad \text{(G-CALL)}$$

$$\frac{\langle \mathbf{ret}_f\ x, \rho, \sigma\rangle = select(\omega) \quad f(y)\{s\} \text{ is a method} \quad \omega' = \omega - \langle \boldsymbol{ret}_f\ x, \rho, \sigma\rangle \quad B = \{rt_f, y\} \quad \gamma' = \gamma \cup \text{eq\_}test(\gamma, B, \rho \wedge rt_f = \sigma(x))}{\langle \Omega \circ \omega, \Gamma \circ \gamma\rangle \Rightarrow \langle \Omega \circ \omega', \Gamma \circ \gamma'\rangle} \quad \text{(G-RET)}$$

$$\frac{\langle x = f(e); s, \rho, \sigma\rangle = select(\omega) \quad \omega' = \omega[\langle s, \rho, \sigma\rangle \bowtie_x^f \gamma\ /\ \langle x = f(e); s, \rho, \sigma\rangle]}{\langle \Omega \circ \omega \circ \{\}, \Gamma \circ \gamma\rangle \Rightarrow \langle \Omega \circ \omega', \Gamma\rangle} \quad \text{(G-POST-CALL)}$$

Fig. 4. Semantic rules.

Statement rules evaluate a symbolic state $\langle s, \rho, \sigma\rangle$ to a set of symbolic states representing its possible continuations. The assignment evaluation is standard. Rule (S-IF-T) states if the path feasibility test determines that only the true branch is taken, the continuation is the statement in the true branch with the path condition updated. Note that we use a flattening operator $\downarrow$ to flatten the mapping in the symbolic store to the corresponding constraints for feasibility testing. We omit the rule for the case when only the false branch is taken. Rule (S-IF-BOTH) evaluates a conditional statement to two continuations when both branches are feasible. Both of them will be added to the work list and evaluated later independently. Rule (S-UNINPRT) specifies that when an external function is encountered, we introduce a fresh symbolic variable to represent its return value. Note that the variable is not constrained.

The global rules describe top-level evaluation, which is represented as a term rewriting process. It starts with the work list stack Ω containing only the work list for the main function. The list has the initial symbolic state denoting the entry of the main function. The return constraint stack Γ is initially empty. The process terminates when Ω's top-level work list becomes empty, meaning all symbolic states have been evaluated.

Rule (G-STMT) describes that the work list selection function chooses a non-call/-return statement to evaluate. The corresponding symbolic state is hence replaced by its resulting continuation state(s). Rule (G-CALL) specifies that if the selected statement is a function invocation, a new work list is pushed to the stack, representing the evaluation context of the callee. The new work list has one single statement denoting the entry of the callee. The new symbolic state is inherited from the caller and updated with the argument mapping. This allows us to test path feasibility inside the callee using information from the caller. Rule (G-RET) specifies that when a return statement is selected for evaluation, the resulting constraint of the return value is tested against the other constraints collected earlier for redundancy. Note that we introduce a special symbolic variable rt_f to denote the return value. The symbolic state of the return statement is removed from the work list, meaning that the evaluation of an intraprocedural path has ended. After all paths within the current function have been evaluated, the work list on top of the stack becomes empty. Rule (G-POSTCALL) specifies when this happens, the constraints of the return value of the current function (in γ) are propagated to its caller. In particular, we use a propagation operator $\bowtie_x^f$ to update the path condition with the constraints we have collected for the return value and insert a mapping from the variable x to rt_f. Note that multiple continuation symbolic states may be yielded by the operation, corresponding to the different paths inside the callee. When the two stacks are popped, we resume evaluating the caller.

Our search strategy explores the paths within a callee upon a method invocation before exploring the paths of the caller. At the call graph level, our search strategy is a depth-first search. Within a function call, the random search strategy is used. Random strategy randomly selects states from a worklist to explore [Cadar et al. 2008]. For example, if function main() calls a(), and a() calls b(), our strategy would randomly explore the paths within b() by randomly selecting states from the worklist. After all paths[1] within b() are explored, the algorithm proceeds to explore the continuation of the paths in a() and the remaining paths in a(). We do not use a global random strategy, which randomly explores all paths. The paths within the same function call share the same pre-condition, that is, path conditions, call stack, parameters, and heap. As such, we can make state comparison based only on the changes made within callees. This reduces the overhead to compare the entire symbolic state.

3.2. Z-equivalence

Here, let us explain the equivalence test, that is, $eq\_test(\gamma, B, \rho)$ in Rule (G-RET) in Figure 4. When our symbolic analysis finishes analyzing a path within a function (i.e., upon reaching the return statement), it calls $eq\_test(\gamma, B, \rho)$ to determine if the constraint ρ computed along the path is redundant by comparing it to the previously computed constraints in γ. We classify constraint variables into two categories: *observable* and *unobservable.* Given a function, observable variables denote those accessible to the statements outside the function. In Rule (G-RET), they are denoted by set B, including the parameter and the return value. In practice, global variables and data structure fields that are reachable through pointer parameters are also considered observable. Unobservable variables are the fresh symbolic variables introduced when analyzing the function (and its callees). They include those representing external APIs and local variables. Our basic idea is to leverage the flexibility of their binding to determine equivalence. Note that we cannot simply ignore unobservable variables and determine z-equivalence, because observable variables may be constrained through unobservable variables.

[1]If there are loops in a function, there can be infinite number of paths in the function. In this case, we set a threshold based on available memory.

Definition 1. Two constraints ϕ and ψ are z-equivalent (denoted as $\phi \simeq \psi$) if $\forall\vec{\beta}\forall\vec{\alpha}_1\exists\vec{\alpha}_2(\phi = \psi)$ and $\forall\vec{\beta}\forall\vec{\alpha}_2\exists\vec{\alpha}_1(\phi = \psi)$, where $\vec{\beta}$ denotes the vector of the union of observable variables in ϕ and ψ; $\overrightarrow{\alpha_1}$ and $\overrightarrow{\alpha_2}$ denote the vector of unobservable variables in ϕ and ψ, respectively; and the variables in $\overrightarrow{\alpha_1}$ and $\overrightarrow{\alpha_2}$ denoting the same program variable are renamed to ensure independence.

A symbolic state is characterized by a state constraint, which involves observable and unobservable variables. Two symbolic states are said to be z-equivalent if their characterizing constraints are z-equivalent. Intuitively, z-equivalence relates indistinguishable states (or constraints) by skipping comparisons of certain sub-expressions that involve unobservable variables. Note that z-equivalent constraints ϕ and ψ may involve two different sets of observable variables. Vector $\vec{\beta}$ is the union of these two sets. There are no common variables between unobservable variable vectors $\vec{\alpha}_1$ and $\vec{\alpha}_2$. In the example in Figure 2, constraints C2 and C3 are z-equivalent, where *type* and *RET* are observable variables and x_i and y_j are unobservable variables. The following theorem shows that z-equivalence observes the three properties of equivalence relation.

THEOREM 1. *The binary relation z-equivalence satisfies the following.*

(*1*) $\forall\phi(\phi \simeq \phi)$ *(reflexive).*
(*2*) $\forall\phi\forall\psi((\phi \simeq \psi) \rightarrow (\psi \simeq \phi))$ *(symmetric).*
(*3*) $\forall\phi\forall\psi\forall\varphi((\phi \simeq \psi) \wedge (\psi \simeq \delta) \rightarrow (\phi \simeq \delta))$ *(transitive).*

ϕ, ψ, and φ are constraints.

These three properties avoid pairwise z-equivalence comparison among constraints. For example, we can conclude that a constraint is z-equivalent to any members of a z-equivalence set if it is z-equivalent to a member of the set. Please refer to our technical report [Li et al. 2013] for the theorems' proofs in this article.

THEOREM 2. *$(\phi \equiv \psi) \models (\phi \simeq \psi)$, where ϕ and ψ are predicate formulae and $\equiv$ denotes logical equivalence.*

Theorem 2 states that logical equivalence is not weaker than z-equivalence. Note that z-equivalence does not necessarily imply logical equivalence. For example, formulae $a_1 > 1$ and $a_2 = 1$ are z-equivalent, where a_1 and a_2 are *unobservable variables*. This is because whether $a_1 > 1$ is *true* or *false*, the second formula $a_2 = 1$ can also be *true* or *false* accordingly by choosing appropriate values for a_2. Similarly, formula $a_1 > 1$ can bind to *true* or *false* regardless of the outcome of $a_2 = 1$. However, they are not logically equivalent. Since z-equivalence is weaker than logical equivalence, we establish its soundness and completeness regarding satisfiability and validity queries by Theorems 3 to 6. A satisfiability query checks if a given property can ever be true. A validity query checks if a property always holds.

THEOREM 3. $(\phi \simeq \psi) \wedge SAT(\psi \wedge \varphi) \models SAT(\phi \wedge \varphi)$.

THEOREM 4. $(\phi \simeq \psi) \wedge \neg SAT(\psi \wedge \varphi) \models \neg SAT(\phi \wedge \varphi)$.

SAT(x) denotes that x is satisfiable. The sets of unobservable variables in ϕ and ψ cannot be referenced in φ.

Theorems 3 and 4 state the soundness and completeness of z-equivalence for satisfiability queries. We use $\wedge\varphi$ in the theorems to denote the constraint generated by analyzing a subsequent path (from ϕ or ψ). We can bind φ to true if we do not further analyze from states ϕ or ψ. Theorem 3 asserts that if a state $(\psi \wedge \varphi)$ originating from state ψ is feasible, then the state $(\phi \wedge \varphi)$ originating from ψ's z-equivalent state ϕ is also feasible. Theorem 4 asserts infeasibility. The two theorems hence assert that

the symbolic analysis explores the same set of paths (or states) starting from the z-equivalent states ϕ and ψ.[2]

THEOREM 5. *If* $\phi \wedge \varphi \mid\!\!\vdash \lambda$ *and* $\phi \simeq \psi$, *then* $\psi \wedge \varphi \mid\!\!\vdash \lambda$.

THEOREM 6. *If* $\phi \wedge \varphi \mid\!\!\nvdash \lambda$ *and* $\phi \simeq \psi$, *then* $\psi \wedge \varphi \mid\!\!\nvdash \lambda$.

Theorems 5 and 6 assert the soundness and completeness for validity queries.

The preceding discussion shows that the proposed z-equivalence has two nice features. First, it can relate more equivalent states by being weaker than logical equivalence. Second, it is strong enough to guarantee the soundness and completeness of path reduction based on z-equivalence. However, deciding z-equivalence is in general undecidable due to the undecidability of first order logic [Huth and Ryan 2004]. Z-equivalence, unlike the format $\exists v_1 \exists v_2 \ldots \exists v_n \phi$ accepted by constraint solvers, involves both universal and existential quantifiers. It is also intractable for first-order logic theorem provers because the computation involves infinite domains, for example, integer domains. Even a formula with only boolean operations would require an exponential algorithm for solving [Reiter and Clayton 2013]. These two factors make deciding z-equivalence difficult.

The key contribution of our work is a linear algorithm that can identify a subset of z-equivalence states effectively. In the remainder of the section, we discuss the algorithm.

3.3. Identifying Unconstrained Expressions

This section introduces the technique we use to identify a set of expressions in a constraint such that the outcomes of these expressions can be arbitrary values confined by their data types, namely, integer or Boolean. We call such a set of expressions *unconstrained*. For example, we consider a singleton expression set $\{a = b\}$ *unconstrained* when variable a represents an unobservable variable, because the Boolean outcome of the expression $a = b$ can be *true* or *false*.

Before presenting our algorithm for unconstrained expression identification, let us illustrate its underlying intuition using three examples. We use β to denote an observable variable and α to denote an unobservable variable representing an un-interpreted function.

Example 1. $\beta_1 = \alpha + \beta_2$.

The expression in Example 1 is unconstrained because α is unconstrained and we can always choose a value of α to make the constraint whether satisfied or not.

Example 2. $(\beta_1 = \alpha_1 + \alpha_2) \wedge (\beta_2 = \alpha_1 + \alpha_2)$.

The addition expression in Example 2 is not unconstrained. Although both clauses in the constraint comprise the operator $+$ and they look similar to the one in Example 1, we cannot find appropriate values for α_1 and α_2 to satisfy the constraint when $\beta_1 \neq \beta_2$, even though α_1 and α_2 are unobservable. This is largely because α_1 and α_2 are shared by the two clauses and they constrain each other. Thus, our algorithm needs to consider how variables constrain each other. A naïve approach could easily get unsound conclusions in cases similar to Example 2.

Example 3. $(\alpha_1 = \beta_2) \wedge (\beta_1 = \alpha_1 + \alpha_2)$.

We can deduce in two steps that the additional expression in Example 3 is unconstrained. First, we can always choose a value for α_1 to control any possible outcome of $(\alpha_1 = \beta_2)$. After that, we can always choose a value for α_2 to control any possible

[2]Assume that symbolic analysis uses the same path exploration strategy for both symbolic states.

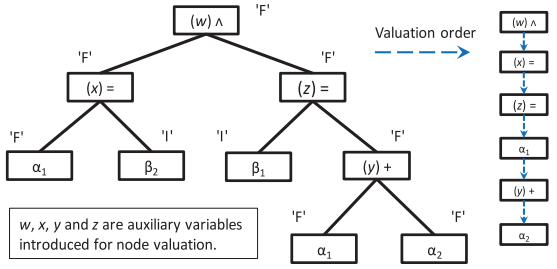

Fig. 5. Abstract syntax tree of the expression in Example 3.

outcome of $(\beta_1 = \alpha_1 + \alpha_2)$. As we can see, a good algorithm should avoid getting unsound conclusions in cases like Example 2 and be precise enough to identify unconstrained expressions as in Example 3.

In light of the three examples, let us examine the 'unconstrained' notion over a set of expressions with unobservable variables that may occur in more than one expression. Our examination focuses mainly on unobservable variables because observable variables and constants are constrained by nature. Suppose S is a set of *non-overlapping* expressions in a constraint under analysis (i.e., disjoint subtrees in the abstract syntax tree of a constraint). We call the unobservable variables that occur only in S the *local unobservable variables* of S, and the other unobservable variables, which occur in S as well as expressions outside S, the *non-local unobservable variables* of S. The bindings of local unobservable variables do not affect expressions outside S. An important criterion for S to be unconstrained is whether the outcomes of its expressions in S assume any outcomes without affecting expressions outside.

Definition 2 (*Unconstrained Expression Set*). Suppose S is a set of expressions $\{s_1, \ldots, s_n\}$, which consists of a set of observable variables, a set of local unobservable variables, and a set of non-local unobservable variables. Let $\overrightarrow{\beta}$, $\overrightarrow{\alpha_l}$, and $\overrightarrow{\alpha_n}$ be a binding of these three sets of variables, respectively. Let $\overrightarrow{\delta}$ be a vector of values $<\delta_1, \ldots, \delta_n>$, where δ_i is an arbitrary value allowed by type s_i. A set of expressions $S = \{s_1, \ldots, s_n\}$ in a constraint is unconstrained if $\forall \overrightarrow{\delta} \forall \overrightarrow{\beta} \forall \overrightarrow{\alpha_n} \exists \overrightarrow{\alpha_l} (\overrightarrow{v_s} = \overrightarrow{\delta})$, where $\overrightarrow{v_s}$ is a vector of values given by $s_1, \ldots, s_n$.

Essentially, the definition means that after binding observable and non-local unobservable variables to some specific values, if we can still ensure that all expressions in S can have arbitrary values by having some solution to local unobservable variables, then S is unconstrained. The establishment of such an unconstrained set of expressions is usually supported by an order of valuations of the Abstract Syntax Tree (AST) nodes that represent these expressions. Here we call assigning a value to an AST node the *valuation of the node*. Constraint solving is thus essentially a process of finding a consistent way of valuating all nodes inside an AST [Huth and Ryan 2004]. For illustration, we use the following concrete example.

Take the singleton expression set $\{(\alpha_1 = \beta_2) \wedge (\beta_1 = \alpha_1 + \alpha_2)\}$ in Example 3 for instance. In the expression, β_1 and β_2 are observable variables, while $\alpha 1$ and $\alpha 2$ are unobservable variables. Figure 5 presents the AST of the expression. We show that we can always follow the valuation sequence $<w = x \wedge z; x = (\alpha_1 = \beta_2); z = (\beta_1 = y); y = \alpha_1 + \alpha_2>$ to find an appropriate binding of α_1 and α_2 to satisfy any valuation of w after β_1 and β_2 are bound. For example, suppose β_1 and β_2 are set to 1 and 2, respectively. Assume that we want w to be false. According to the previously mentioned valuation sequence, we can first bind x and z in $w = x \wedge z$ to $false$ and $true$, respectively. Then we

can bind α_1 to 0, y to 2, and finally α_2 to 2. Similarly, if we want w to be true, we can also find such a valuation order. As a result, we can conclude that this singleton expression set is unconstrained. We note that not all expressions in a constraint necessarily form an unconstrained set (e.g., the two clauses in Example 2).

Before discussing our identification algorithm, we introduce three types of operators that are important for the algorithm.

Definition 3. A Type I operator $\otimes$ is a commutative operator that satisfies $\forall x \forall a \exists b$ $(x = (a \otimes b))$ *and* $\forall x \forall b \exists a (x = (a \otimes b))$, where the values of x, a and b are only bounded by their types.

In this work, we consider $=, \neq, +, -$ as Type I operators. Besides binary operators, the unary operator can also be modeled as 0-a and considered as a Type I operator. The unary operation $\neg x$ can be converted to $false = x$ so that $\neg$ can be considered as a Type I operator.

Definition 4. A Type II operator $\otimes$ is a commutative operator that satisfies $\forall x \exists a \exists b (x = (a \otimes b))$, where the values of x, a and b are bounded by their types.

In this work, we consider $>, <, \leq, \geq, \times, \div, \vee$, and $\wedge$ as Type II operators. We also consider all bit operators as Type II operators, where unary bitwise complement $\sim x$ can be modeled as -1 *xor* x. We consider operator $\geq$ as Type II instead of Type I because we cannot make $\alpha \geq \beta$ *false* when β binds to the minimal machine representable integer, even if α is a free variable. The same applies to $>$, $<$, and $\leq$.

Definition 5. An operator, which is neither Type I nor Type II, is a Type III operator.

Type III operators induce expressions that are not flexible, that is, we may not be able to make such an expression have a desired value. Type III operators include arithmetic *mod* and operators for heap object manipulation. We classify the *mod* operator as Type III because the outcome of a *mod* operation cannot be the largest representable integer.

We propose six rules in Figure 6 to transform the AST of a given constraint into a tree annotated by tags and arcs. The set of unconstrained expressions can be derived from the tags. The arcs compose a partial valuation order that serves as the witness. There are three types of annotations for an AST node, namely, 'Flexible (F)', 'In-flexible (I)', and 'Undecided (?)'. An 'F'-annotated node u means that u can flexibly bind to any value. Before applying the six rules, nodes that represent operators or occurrences of unobservable constraint variables are initialized with '?' and the remaining nodes with 'I'. The 'I'-annotated nodes are either occurrences of observable constraint variables or constants, which cannot be bound to arbitrary values. The rules explore the structure of an AST and iteratively transform some '?'-annotated nodes to 'F'/'I' nodes. An arc from node u to v indicates that u should be valuated before v.

Rule 0 is the preprocessing rule. A parent node is annotated with 'I' if both children have been annotated as 'I', as its value must not be flexible. It is applied iteratively before the other rules until it cannot be applied further. This is to identify all the nodes that are absolutely not flexible.

Rule 1 is for initialization. It annotates the nodes that represent unobservable variables with a single occurrence in the AST by 'F'. These variables must be local unobservable variables of their enclosing expressions. They can be bound to arbitrary values. Rules 2–5 are applied iteratively until they cannot be applied further. In each step, one node changes from '?' to 'F'.

Node w in Rule 2 denotes a Type I operator with two operands s and c, that is, $w = c \otimes s$, where nodes c, s, and w denote the outcomes of the expressions whose ASTs are rooted at c, s, and w, respectively. Rule 2 reflects the following situation: if w is a

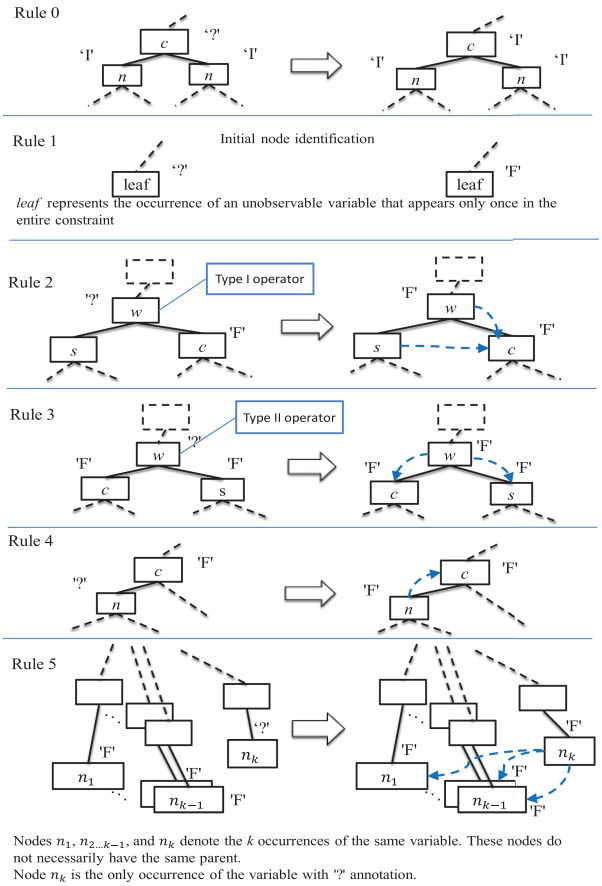

Fig. 6. Partial-order construction rules.

Type I operator and the operand c can be bound to an arbitrary value, then w can also be bound to an arbitrary value. Rule 2 also identifies a partial order that requires w and s to be valuated before c. It is justified by the fact that a binding of c always exists for any specific bindings of w and s according to the definition of Type I operators.

Similarly, Rule 3 includes w in the valuation sequence and gives a partial order by which we can always bind c and s to some values after w is valuated, where $w = c \otimes s$ and $\otimes$ is a Type II operator. This can be justified in a similar way to that of Rule 2.

Rule 4 states that if we have identified that the parent node c can be bound to an arbitrary value (through previous rule applications), the child node n is flexible if it has not been annotated, regardless of the type of the operator $\otimes$. This is because the flexibility of c allows us to bind it to any value even after we have bound n. The arc indicates such a valuation order. Rule 5 is applied to a scenario where an unobservable variable has k occurrences in the entire AST, and $(k-1)$ of the occurrences have been annotated by 'F', while the last occurrence is annotated by '?'. The nodes $n_1, n_2, \ldots, n_k$ in Rule 5 refer to the k occurrences of the unobservable variable n. The arcs in Rule 5 indicate the partial order to valuate n_k before $n_1, \ldots, n_{k-1}$. In such scenarios, Rule 5 will annotate the node n_k by 'F'. After that, we can further apply Rules 2 and 3 to annotate more nodes that are ancestors of n_k.

A fixed point is reached when further exercise of Rules 2–5 do not result in additional 'F'-annotated nodes. To study the properties of the annotated AST at a fixed point, let us first introduce the concept of *Flexible-Abstract-Syntax-subTree* (*FAST*).

Definition 6 (*Flexible-Abstract-Syntax-subTree*). Suppose TS is the set of all annotated ASTs rooted with an 'F'-annotated node when a fixed point is reached. An AST in TS is a *Flexible-Abstract-Syntax-subTree* (FAST) if it is not a proper subtree of any ASTs in TS.

For example, the entire annotated AST in Figure 5 is an FAST. However, the AST rooted at node z is not an FAST, because z has an 'F'-annotated ancestor node w. Theorem 7 asserts that the set of expressions represented by the FASTs at the fixed point of AST transformation is unconstrained. Intuitively, the outcome of an expression corresponding to a FAST can be flexibly bound to any value.

LEMMA 1. *A valuation sequence that conforms to the partial order constructed by the rules in Figure 6 always exist in the set of FASTs.*

This can be proved by showing that our partial order construction rules never introduce cycles into the graphs that comprise the AST nodes and the generated arcs at fixed points (i.e., the graph is a directed acyclic graph). Please refer to our technical report [Li et al. 2013] for the proof.

THEOREM 7. *If $S = \{s_1, \ldots, s_n\}$ is the set of FASTs in an annotated AST when a fixed point has been reached, the set of expressions represented by these FASTs is unconstrained.*

We can show that we can always construct a valuation sequence from our partial orders to help establish the fact that the concerned expression set is unconstrained.

THEOREM 8. *If S_p and S_q are the sets of FASTs obtained at the two fixed points reached by applying the rules to an AST in two different orders p and q, S_p equals S_q.*

The theorem dictates that the order of the rules that are applied is irrelevant. We can prove Theorem 8 by modeling the AST transformation process using Petri nets [Murata 1989]. We show that the applications of our rules can be simulated using transition firings in Petri Nets. The problem of deciding whether an AST node is 'F'-annotated at a fixed point can be reduced to the classic reachability problem in Petri nets. Then we can prove that the sets of FASTs obtained at different fixed points are identical and independent of rule application orders by showing that the behavioral property "reachability" of Petri nets is not affected by the transition firing orders [Li et al. 2013].

The six rules in Figure 6 are implemented by an algorithm that uses a *worklist* to track the process of annotating nodes. The *worklist* is initialized by Rule 1. The algorithm removes one node from the *worklist* each time. Any rule(s) that can be applied to the node are applied. A node is added to the *worklist* if it changes from '?' to 'F', which may lead to further rule application. We illustrate the execution of such an algorithm in Figure 7. Tree T1 in Figure 7 represents the AST of Example 3 with initial annotations, that is, the 'I' and '?' annotations. Since the unobservable variable α_2 has only one occurrence in the entire constraint, we add it to the *worklist* and annotate it by 'F' according to Rule 1. Tree T2 gives the result after this step. Note that we use α_1 and α_1' in Figure 7 to denote two different occurrences of the same variable α_1 in the example constraint. In T3, the algorithm removes α_2 from the *worklist*, annotates node y by 'F' (Rule 2), and then adds y to the *worklist*. In T4, the algorithm removes node y from *worklist* and annotates nodes z, α_1', and α_1 by 'F' using Rule 1, Rule 4, and Rule 5, respectively. Nodes α_1 and z are then added to *worklist*. In T5, the algorithm removes α_1 from the *worklist*, annotates x by 'F' (Rule 1), and adds x to the *worklist*. In T6, the

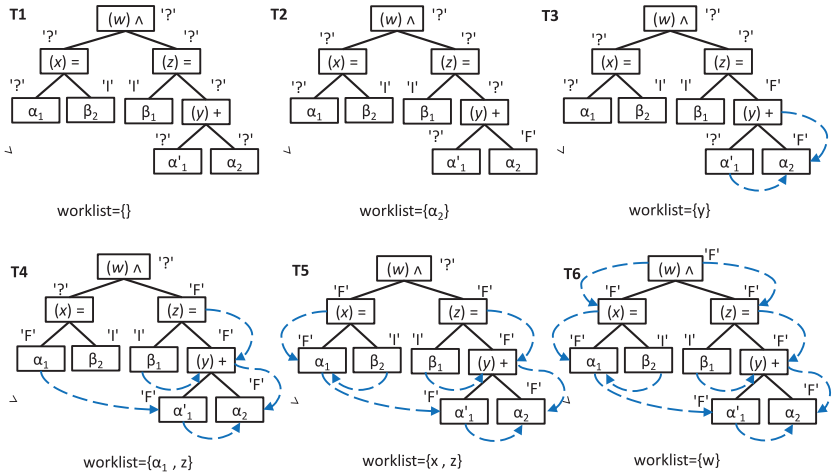

Fig. 7. Stepwise illustration of Example 3.

Table I. Unification Rules

	Case	Unification substitution	Explanation
1	$\beta_i = \beta_i$	$\{\}$	Two identical observable variables can be unified
2	$\alpha_i = a_j$	$\{a_i \rightarrow a_j\}$	Two unobservable variables can be unified by renaming.
3	$FAST_i = FAST_j$	$\{\}$	Two FASTs can be unified.
4	$\otimes_i = \otimes_i$	$\{\}$	Identical operators can be unified.
5	$c_i = c_i$	$\{\}$	Identical constants can be unified.

algorithm annotates w by 'F' using Rule 2. Now we have established that the singleton set of expressions $\{(\alpha_1 = \beta_2) \wedge (\beta_1 = \alpha_1 + \alpha_2)\}$ is unconstrained. The valuation sequence is also illustrated in Figure 7.

Complexity of the Annotation Algorithm. Before an operator node is added to the *worklist*, the algorithm always checks if this node has a '?' annotation and then annotates it with 'F'. Therefore, both leaf nodes and operator nodes can be added to the *worklist* at most once. It takes constant time to process the removal of a node from the *worklist*. Therefore, the complexity is linear to the size of AST.

3.4. Equivalent Symbolic State Detection

To determine z-equivalence of two symbolic states, we apply our algorithm to identify unconstrained sets of expressions in the ASTs of the two constraints that encode the two symbolic states. We adapt the unification algorithm of first-order logic [Russell and Norvig 2003] to check if two constraints are unifiable using the rules in Table I. The unification algorithm takes the ASTs of two constraints as input and performs top-down comparison using the unification rules in Table I. If two constraints are unifiable, they are z-equivalent. The only special rule in Table I is that two FASTs can be unified even they are syntactically different. For example, two constraints $c1$: $\beta_1 \times (\alpha_1 + 34) = \alpha_2 + \alpha_2$ and $c2$: $\beta_1 \times (\alpha_3 - \alpha_4) = \alpha_5 + \alpha_5$ can be unified as follows. We first identify $(\alpha_1 + 34)$ in $c1$ and $(\alpha_3 - \alpha_4)$ in $c2$ as FASTs using our algorithm. Then $(\alpha_1 + 34)$ and $(\alpha_3 - \alpha_4)$ can be unified using Rule 3 in Table I. In the first step, $c1$ is rewritten to $\beta_1 \times (\alpha_3 - \alpha_4) = \alpha_2 + \alpha_2$. Variables α_2 and α_5 can be unified by renaming α_2 to α_5 according to Rule 2 in Table I. In the second step, $c2$ is rewritten to $\beta_1 \times (\alpha_3 - \alpha_4) = \alpha_5 + \alpha_5$. The remaining variables and operators of $c1$ and $c2$ are identical and can be unified by

other rules in Table I: as such, constraints $c1$ and $c2$ are unifiable and z-equivalent. Once two state constraints are z-equivalent, their corresponding symbolic states are z-equivalent. We only need to symbolically analyze one of these states. Note that we only perform z-equivalence detection for the set of states S upon the return of a function invocation (Section 3.1).

4. IMPLEMENTATION

4.1. Tools

We have implemented our algorithm on top of two symbolic analysis engines, namely, KLEE [Cadar et al. 2008] and Sym-JVM. KLEE is a symbolic execution engine for C programs. Sym-JVM is our own Java symbolic analysis engine. Sym-JVM adopts similar optimizations of KLEE by leveraging copy-on-write and symbolic object lazy initialization techniques to reduce state forking and memory overheads. These optimizations are essential to applying symbolic analysis to large programs.

4.2. Encoding of Heap Constraints

We introduce *obj* and *val* operators to model heaps. If a reference r points to an object o, we express this relation as $obj(r) = o$. If the field f of an object o has a value of 5, we express it as $val(o, f) = 5$. Arrays can be modeled in a similar way. For example, our algorithm uses a constraint $obj(arr) = a \wedge val(a, 5) = 4$ to model $arr[5] = 4$, where a denotes an array object. For languages like C, we use offsets instead of field names. For example, our algorithm generates a constraint $obj(p) = m \wedge val(m, 0) = \alpha$ for the following code with pointer p, memory region m, and index 0 as observable *variables* in procedure foo().

$$\text{void foo(int } {}^{*}\text{p)}\{{}^{*}\text{p} = \text{networkRead()};\}.$$

Similar ideas have been adopted by earlier work [Clarke et al. 2003] to handle heap constraints. More complex data structures can be represented using these primitive operators. The difference is that we only use such heap encoding for equivalence detection purposes. Our algorithm considers the two heap operators obj and val as Type III operators. This dictates that the constraints related to these operators must have isomorphic syntactic structures in order to be z-equivalent. Although treating heap operators as Type III operators appears to affect the degree of reduction, an earlier study [Hackett and Aiken 2006] shows that large applications mostly use only a small proportion of global pointers. For non-global pointers, another study [Dillig et al. 2008] finds that the call chains of pointer propagation are typically short with an average length of less than six. Many data structures are, therefore, subject to only a few scopes and can be pruned once they are out of those scopes and become unobservable. As a result, a lot of states with various heap structures can still be z-equivalent.

4.3. Complex Returns from External Functions

We found that there are a very large number of external function calls in both Java subjects and Kernel modules. For example, the standard library of JDK 1.7 contains about 5,000 native methods with 500,000 lines of native code. It is impractical to model all these functions. Many of these methods return objects of complex data structures. Hence, simply introducing a fresh symbolic variable is not sufficient. To address this issue, we allow lazy initialization of unknown data structures [Khurshid et al. 2003]. For example, a program invokes a native function and that function returns an object o. Our tool first marks the object to be an unknown object. When the subsequent code accesses a field f in o, our tool would initialize field f to point to a new object x and mark x to be an unknown object.

Table II. Java Subject Information

Name	Version No.	Lines of Code	Entry Classes	
Apache Ant	1.8.4	128,133	taskdefs.optional.ejb.IPlaneEjbc	1
			taskdefs.KeySubst	2
Apache Hadoop	0.19.2	203,190	fs.FsShell	3
			mapred.TaskTracker	4
			hdfs.server.datanode.DataNode	5
			hdfs.server.namenode.NameNode	6
			hdfs.tools.DFSck	7
			io.compress.CompressionCodeFactory	8
			mapred.JobClient	9
			mapred.JobShell	10
			mapred.lib.aggregate.ValueAggregatorJob	11
			mapred.lib.InputSampler	12
			mapred.JobTracker	13
			util.RunJar	14
Apache Lucene	4.0.0-BETA	392,065	demo.IndexFiles	15
			demo.SeachFiles	16
			facet.example.multiCL.MultiCLIndexer	17
			misc.HighFreqTerms	18

4.4. Equivalence Checks

Our existing implementation uses functions as boundaries to distinguish observable and unobservable variables. We perform equivalence detection at the end of a function call. We note that more variables can be identified as unobservable by performing a forward static analysis to identify the variables that are not directly referenced by name in the subsequent analysis. We will improve our approach by incorporating such analysis in our future work.

5. EVALUATION

To evaluate our approach, we ran KLEE and Sym-JVM augmented with our technique to analyze four large open-source subjects, namely, Ant, Lucene, Hadoop, and Linux Kernel. To evaluate the scalability of our technique, the Java subjects Ant, Lucene, and Hadoop we used are much larger than conventional Java benchmark suites (e.g., Dacapo 9.12 has 41,593 SLOCs). Since the Java subjects Ant, Lucene, and Hadoop contain multiple main methods, we performed experiments on the main methods listed in Table II after excluding small main classes with less than 500 reachable methods. The C subject Linux kernel consists of many components and more than 9.9 million lines of code. The components are invoked via interrupts and there is no calling relationship between them. We selected 20 top-level methods in these modules as analysis entry functions for our experiments. These top-level methods were selected in the following way. During the compilation phase, we performed call graph traversal and ranked the top-level functions (i.e., methods with no direct callers in the call graph) according to the number of object code files reachable from them. Then we selected the top 20 methods. For each of those, we linked the dependent object code into a single "module". These modules are about 10–30MB in terms of LLVM bitcode, as shown in Table III. Note that the bitcode size of the subjects used in the earlier study of KLEE [Cadar et al. 2008; Kuznetsov et al. 2012] were usually less than 500KB. We ran KLEE to analyze each of the 20 selected methods. Our experiments were run on Linux machines (CentOS 5, x86_64 edition) with two dual core 2.4GHz AMD Opteron processors and 16GB RAM. We set the maximum allowed memory to 5.5GB.

Table III. Linux Kernel (3.4.4) Entry Function Information

Entry Function		Size	Entry Function		Size
acpi_battery_notify	1	8.3M	_handle_hotplug_event_fune	11	10M
acpi_cpu_soft_notify	2	9.2M	nouveau_fbeon_find_or_ereate_single	12	21M
acpi_pci_root_add	3	8.7M	nouveau_load	13	37M
acpi_processor_add	4	8.8M	nv50_display_ereate	14	22M
cayman_init	5	20M	r520_init	15	18M
check_sub_bridges	6	10M	r600_init	16	20M
disable_slot	7	11M	radeon_driver_load_kms	17	21M
evergreen_init	8	20M	rv515_init	18	18M
_handle_hotplug_event_bridge	9	10M	rv770_init	19	20M
shpc_probe	10	9.6M	vmw_driver_load	20	22M

For Java programs, we introduced fresh symbolic variables to denote return values from native calls, I/O APIs, encryption /hashing APIs, and certain string operations that the underlying Choco constraint solver cannot solve. For Kernel, we considered return values from functions in the assembly code and the functions that DragonEgg[3] failed to convert to LLVM bitcode as external functions. The setups are similar to earlier studies [Babic and Hu 2008; Chandra et al. 2009].

5.1. Effectiveness of State Reduction

Our first experiment focuses on evaluating the effectiveness of state reduction. For each entry function, we ran KLEE/Sym-JVM for 24 hours. During the analysis, we recorded the number of symbolic states in the memory. We also calculated the *multiplicities* of symbolic states. Intuitively, the multiplicity of a symbolic state is the number of paths denoted by the state before reduction. As we discussed earlier, two z-equivalent states would lead to the traversal of the same set of consequent paths (recall Theorems 3 and 4). Therefore, we can leverage the multiplicity metric to measure the reduction of states as follows. We denote the multiplicity of a symbolic state s as $m(s)$. Initially, the multiplicity of a state is one. If we find that one state s_1 is z-equivalent to another state s_2, our approach would remove state s_2. Then, the updated multiplicity $m(s_1)$' of s_1 is calculated as: $m(s_1)' = m(s_1) + m(s_2)$. If state s_1 later forks into two states, both of the forked states would inherit the multiplicity $m(s_1)$'.

Using this metric, Figure 8 shows the reduction in the number of states in the memory. X-axis corresponds to each entry function of our Java and C subjects. The Y-axis indicates the reduction in log scale. These reduction results were obtained from snapshots at the end of the 24 hour period. We can observe from Figure 8 that our approach can achieve at least a few orders of magnitude reduction in most cases. On average, our approach achieved 15 orders of magnitude reduction for Java subjects, and 10 orders of magnitude reduction for the Linux kernel. We know that the amount of computational resources (e.g., RAM) that a symbolic analysis engine requires at any time is roughly dependent on the number of symbolic states in the memory at that time, as shown in Appendix A. As such, the very large numbers of states indicate the symbolic analysis engine could require an unrealistically large amount of resources to analyze a large program. Therefore, the memory needing to be consumed for the analysis on the same set of states without reduction is unavailable. The reduction achieved by our technique would enable scaling to large programs.

We also studied whether our approach can explore more states given the same time budget (24 hours). To answer this question, we ran Sym-JVM and KLEE to analyze the

[3]DragonEgg is a GCC plugin for converting GCC IR to LLVM bitcode. It is available at `http://dragonegg.llvm.org/`.

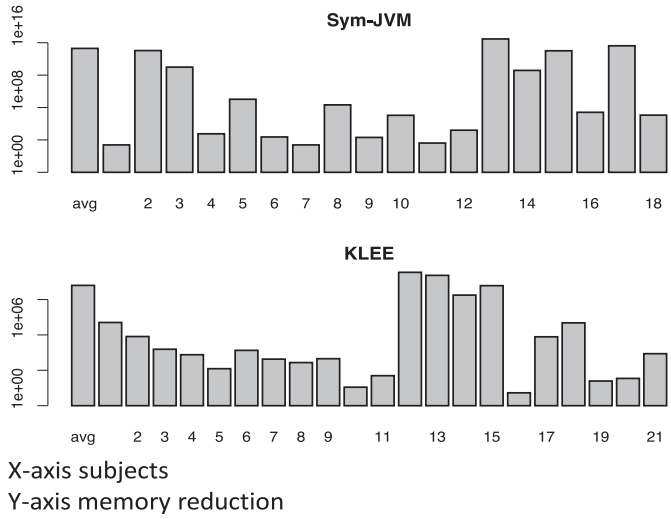

Fig. 8. State reduction in the analysis of each subject.

Java subjects and 20 modules of Linux kernel with three different state exploration strategies. For KLEE, the first strategy is random, which explores states in a random manner. The second strategy uses the distance to uncovered statements as a heuristic to guide the symbolic analysis engine to cover as many uncovered statements as possible. These two are the existing state exploration strategies used in KLEE studies [Cadar et al. 2008; Kuznetsov et al. 2012]. We use them as a comparison baseline to study our state exploration strategy based on z-equivalence detection. Our state exploration strategy, namely z-equivalence strategy, works as follows. The state exploration along a path of a function is paused upon reaching return instructions and waits till all other paths of the function are explored. When this occurs, we check z-equivalence among the states collected at return instructions. The base-line strategy of Sym-JVM is random. Each of these strategies assumes the same set of external functions, which are considered as uninterpreted functions. Figures 9 and 10 show the comparison. In the figure, we refer to the previously-mentioned three state exploration strategies as "Random", "CoverNew", and "Z-equivalence", respectively. Observe that our approach can explore on average about seven orders of magnitude more states.

Discussion. While our technique is effective in practice, it does not guarantee it will detect all unconstrained expression sets. Omission can occur when unobservable variables are tightly coupled. For example, our rules fail to detect the unconstrained expression $(\alpha_1 + \alpha_2 = \beta_1) \wedge (\alpha_2 + \alpha_3 = \beta_2) \wedge (\alpha_1 + \alpha_3 = \beta_2 + \alpha_1)$, because each occurrence of an unobservable variable in a subexpression is coupled with another one in a different subexpression. We speculate an effective solution to the problem may require using nonlinear algorithms. We leave it to our future work.

5.2. Performance

To study the overhead of our approach (mainly the equivalence detection overhead), we ran Sym-JVM and KLEE with the different state exploration strategies for one hour to analyze the subject programs and modules. The breakdown of the execution time is shown in Figure 11. Observe that equivalence detection is only 24% and 6% of the execution time on average for Sym-JVM and KLEE, respectively. The detection time overhead is not significant because of the linear complexity of our algorithm. The

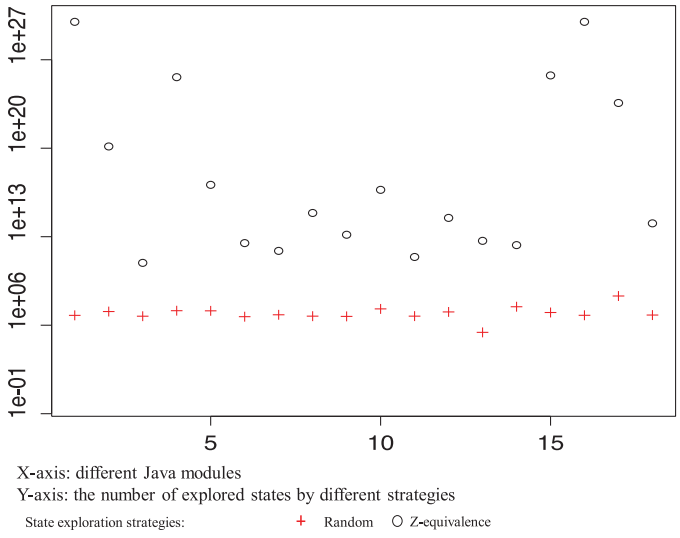

Fig. 9. Number of states explored (Sym-JVM).

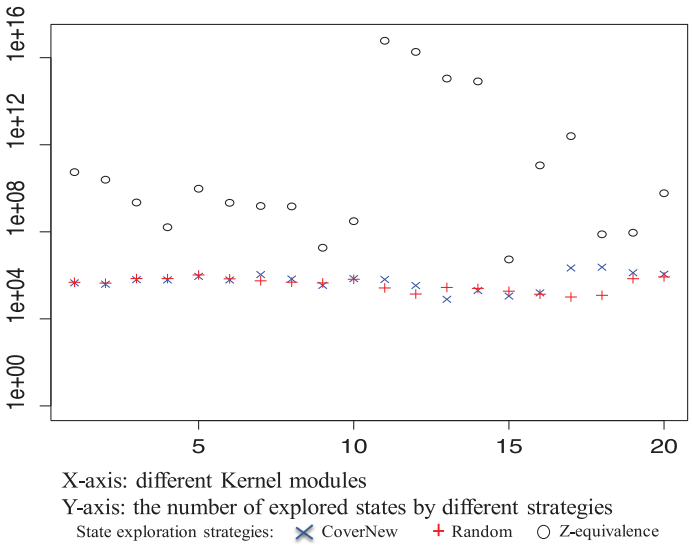

Fig. 10. Number of states explored (KLEE).

equivalence detection is much more lightweight than other dominant factors, such as constraint solving. The overhead is higher for Java subjects because Java programs tend to have a larger number of small functions so that equivalence detection has to be performed more often. Consequently, the time spent on z-equivalence detection is relatively more significant.

We further compare the number of analyzed instructions in one hour for the three strategies to observe if the overhead of the z-equivalence strategy leads to performance degradation when compared with the random strategy on Sym-JVM (and the random strategy and CoverNew strategy on KLEE). For the performance comparison of z-equivalence, we used the actual number of instructions analyzed by symbolic

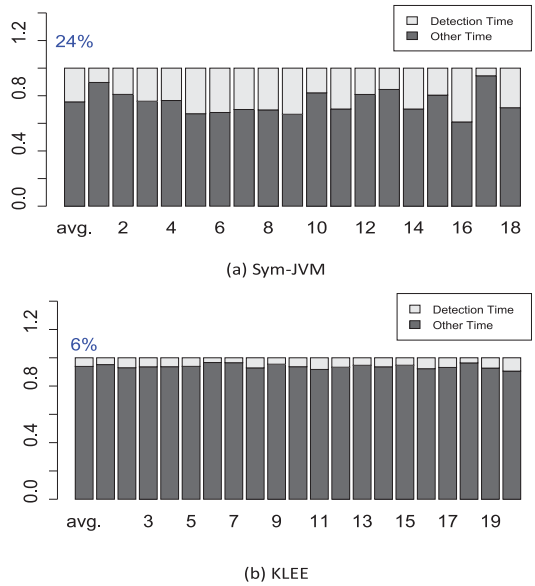

Fig. 11. Runtime overhead.

execution engines without multiplying it with the multiplicity just discussed. The instructions here refer to LLVM bitcode for KLEE and Java bytecode for Sym-JVM. To compare the relative performance, we choose the performance of random strategy as the basis of comparison. The relative performance of z-equivalence or CoverNew can thus be measured using a *normalized number of instructions* given by

$$\frac{\#instructions\ analyzed\ by\ the\ strategy\ under\ comparison}{\#instructions\ analyzed\ by\ random\ strategy} - 1.$$

Figure 12 presents the experimental results of the performance comparison. Observe that z-equivalence is subject to minor (on average 4.1%) performance degradation on Sym-JVM, while there is no obvious (on average 1%) performance degradation on KLEE. In some cases, using the z-equivalence strategy can even process more instructions (positive bars). We believe that this is because of the benefits derived from constraint caching. Recall that we analyze all paths within a callee before going back to the caller, which may allow more cache hits.

5.3. Code Coverage

In the next experiment, we study whether our approach is able to increase code coverage given the same time budget (24 hours).

We present the experimental results in Figure 13. Figures 13(a) and 13(c) give the results for function coverage, and Figures 13(b) and 13(d) give the results for basic block coverage. The X-axis represents the analysis of each subject, and the Y-axis represents the code coverage (e.g., 1.0 means 100%). Observe that our technique significantly outperforms the other strategies. On average, by adopting our strategy, the symbolic analysis engine KLEE can cover 58% functions and 55% basic blocks of the studied Linux kernel modules. For some subjects, our approach is able to get about 80%

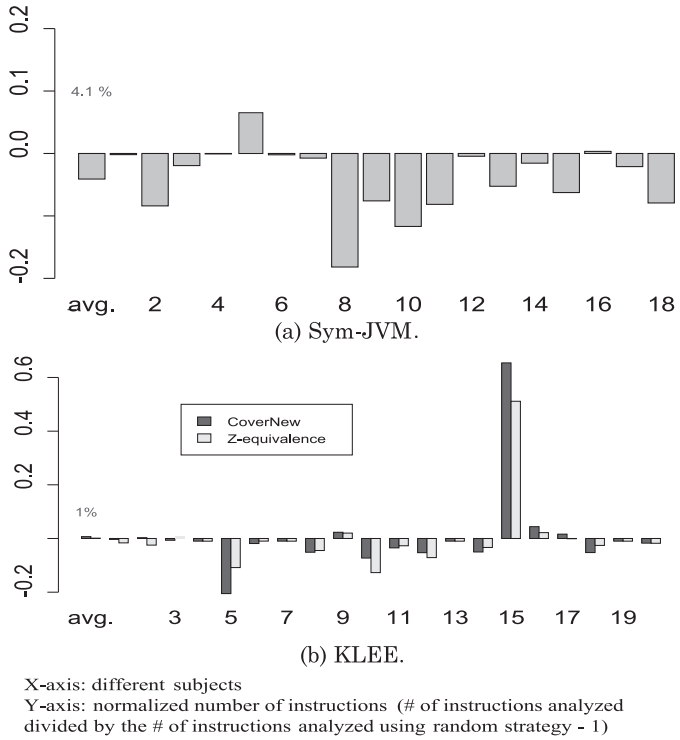

Fig. 12. Runtime performance comparison.

function and block coverage. In contrast, the random strategy can only cover 10% functions and 6% basic blocks, and the heuristic-based "CoverNew" strategy only performs marginally better than the random strategy. Note that with infinite resource (i.e., time and memory), the two existing strategies would be able to reach the same coverage as ours. The advantage of our technique lies in its ability to scale up symbolic analysis with limited resources. Similarly, Sym-JVM with z-equivalence detection improves the function coverage and basic block coverage by 40% and 32%, respectively.

5.4. Bug Detection Capability

Better coverage suggests that we can potentially expose more software defects. In the next experiment, we compare the bug detection capability of the three strategies. In our experiment, KLEE is used to detect memory errors, while Sym-JVM is configured to detect null point exceptions, assertion errors, and exceptions thrown out of the main methods. Figure 14 shows the results. For Sym-JVM, our approach is able to detect more errors in one third of subjects. For KLEE, our approach can detect one order of magnitude more errors given the same budget (24 hours). The bugs detected by our strategy are a super set of those detected by the other two, illustrating the scaling effect of our technique. All three strategies generate inputs that can be used to confirm the bugs. Next we show one case of bugs detected by our technique, which was not detected by the other two.

It is a Linux Kernel bug. Figure 15 presents the relevant code snippets. In snippet A, variable *node* is found to be NULL along some paths. Variable *node* is the alias of a field called *resolved_node* of the *info* struct. Our technique analyzes how *info* is initialized in Line 185 of nsxfeval.c in Snippet D. The initialization of *info* involves

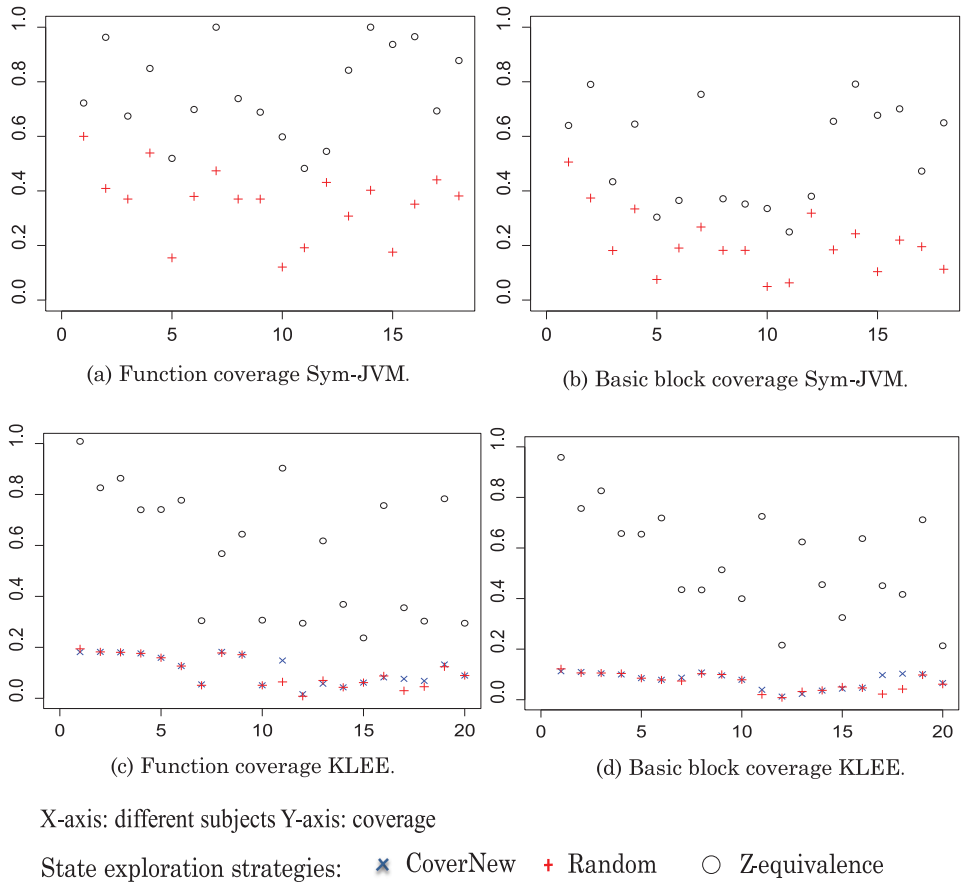

Fig. 13. Code coverage comparison.

multiple functions, inlined functions, and macros defined in *slab_def.h*, *aclinux.h*, *utalloc.c*, *acacros.h*, *ctype.h*, and *nspredef.c*. Our technique decides that the *info->resolved_node* is initialized to NULL. In the third branch, info is passed to acpi_ns_evaluate() in Line 264 *nsxfeval.c*. In Snippet C, *info->resolved_node* is passed to acpi_ns_get_node(). Our technique determines the path that does not initialize *resolved_node* in Snippet B is feasible. The NULL *resolved_node* is hence passed to acpi_ns_check_for_predefined_name() and triggers a null pointer de-reference error in Line 383 of *nspredef.c* (Snippet A). Our technique also produces the failure inducing input, by assigning NULL to *handle*, not-NULL to *pathname*, and "x" to the first element of *pathname*. Such a bug can take a long time to find and verify manually because the relevant code snippets distribute in a few source files. The other two strategies were not able to explore the faulty path due to the large search space.

We want to point out that our contribution is not to address precision issues caused by API-approximation, which is a universal issue of many existing static analyses [Babic and Hu 2008; Babic et al. 2007; Chandra et al. 2009; Dillig et al. 2008], but rather improving scalability of symbolic analysis. Our reduction does not introduce any additional imprecision. Given infinite resources, the original KLEE and Sym-JVM will eventually reach the same coverage, report the same set of warnings, and have the same true and false positives as our technique, as proven by Theorems 3–6. As

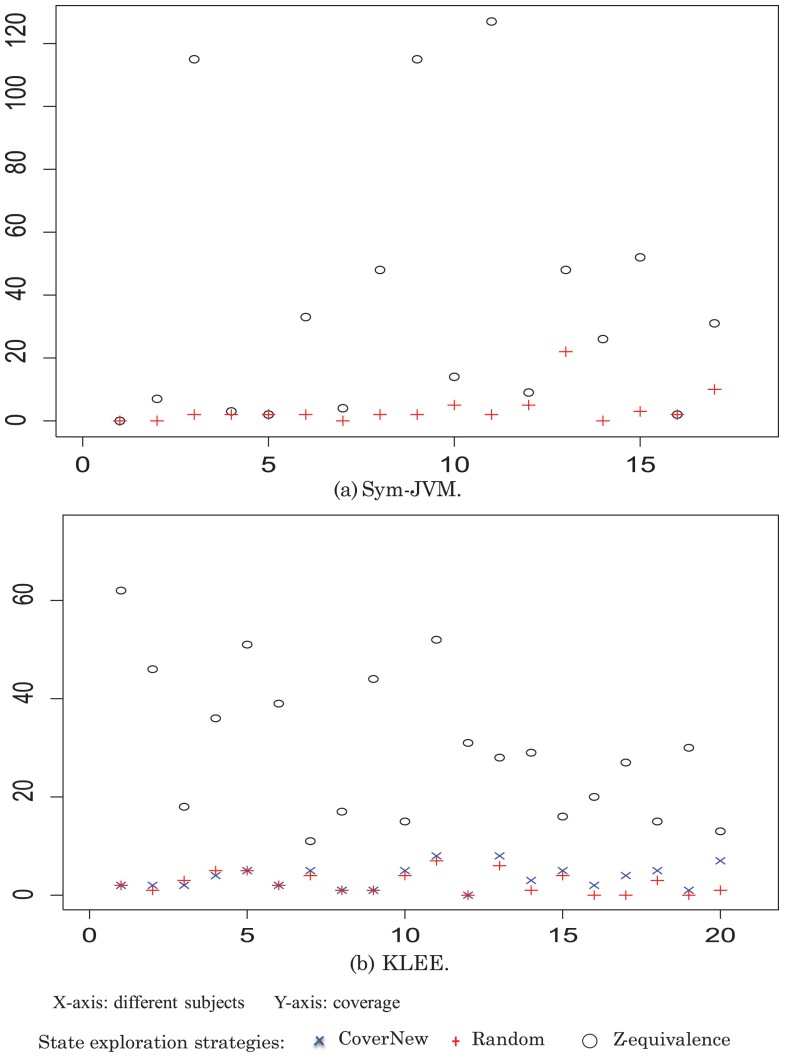

Fig. 14. Runtime errors detected.

mentioned in the introduction section, API over-approximation is a necessary practical compromise for large real-world programs due to the presence of native methods and assembly functions. It is also undesirable to apply concolic testing because repeatedly booting large programs like Kernel is very time consuming.

Our proposed technique enables symbolic analysis to be conducted over those APIs whose precise models are unavailable, by approximating their behaviors using uninterpreted functions. Our experimental results based on real subjects show that such API-approximation is able to provide useful analysis. For example, Sym-JVM finds an assertion violation in the Lucene3xSegmentInfoReader class in the Lucene project. The assertion is given in Example 1 in Figure 16. The violation indicates that the program has not taken appropriate actions to handle malicious inputs. Symbolic analysis can be used to check these user-provided assertions, including the assertions that are implemented by invoking a user-specified function to check complex

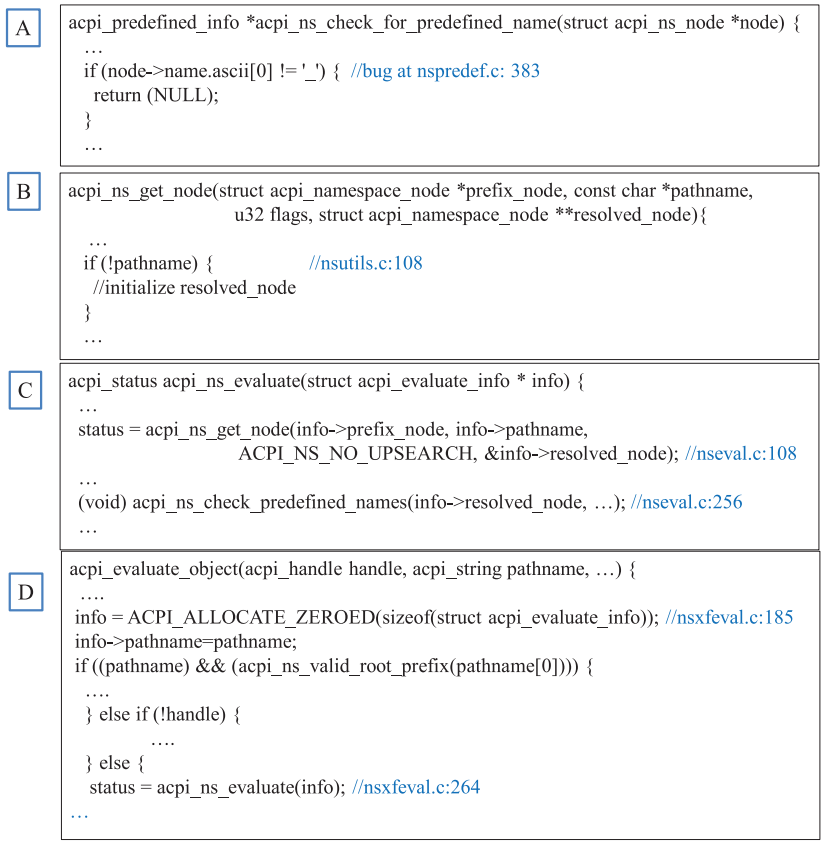

Fig. 15. One bug example in Linux kernel.

properties. In Example 2 shown in Figure 16, Sym-JVM finds that the exception in Line 14 is thrown out of the main method without being properly handled. Sym-JVM also found other true warnings[4] with test cases in the experiment.

Static analysis is generally subject to various sources of imprecision, including points-to analysis, arithmetic, value over-approximation caused by merging, path/context-sensitivity, and external API approximation, and so on. In view of the unavailability of a precise model for many native APIs [Păsăreanu et al. 2008], API approximation is a pragmatic trade-off to enable the use of symbolic analysis to many large open-source projects as well as those developed over a relatively volatile set of APIs, such as Android.

5.5. Comparison with Merging Techniques

Another approach to reducing the number of paths explored in symbolic analysis is to merge the constraints arising from different paths [Kuznetsov et al. 2012]. For example, two paths from code snippets: **if** $(a > 0)x = 2$ else $x = 34$ can be encoded by a single constraint $(a > 0 \rightarrow x = 2) \vee (a \leq 0 \rightarrow x = 34)$. A key issue in applying this approach to real-world large programs is to merge different program states with various heap structures. One solution is to use constraint solvers to resolve all store

[4] `http://sccpu2.cse.ust.hk/symjvm/testcases_for_java.zip`. User name: symjvm, password: password.

```
1  int docCount = input.readInt();
2  ....
3  final int delCount = input.readInt();
4  assert delCount <= docCount;
```

Example 1 Lucene3xSegmentInfoReader

```
1       long genA = -1;
2       ...
3       long genB = -1;
4       if (version == FORMAT_SEGMENTS_GEN_CURRENT) {
5          long gen0 = genInput.readLong();
6          long gen1 = genInput.readLong();
7          if (gen0 == gen1) {
8            genB = gen0;
9          }
10      }
11      gen = Math.max(genA, genB);
12      if (gen == -1) {
13        // Neither approach found a generation
14        throw new IndexNotFoundException("no segments* file found in " + directory);
15      }
```

Example 2 Segmentinfos

Fig. 16. Examples from java subjects.

and load operations by treating the entire address space as a single flat array. However, Cadar et al. [2008] found that no known constraint solvers are able to solve the resultant complex constraints generated from large programs. This explains why existing symbolic analysis engines like KLEE [Cadar et al. 2008] do not fully rely on constraint solvers to reason about heap-related behavior. Instead, they decompose memory into smaller objects and resolve objects by the engine itself with some support of constraint solvers. Due to the complexity of heap constraints, existing constraint merging techniques [Cadar et al. 2008; Kuznetsov et al. 2012] for symbolic analysis engines require merging constraints to have compatible heap structures. Even so, the constraints generated by merging may increase the constraint solving time significantly [Ganesh and Dill 2007; Kuznetsov et al. 2012]. The state of the art is to use query count estimation to selectively merge constraints [Kuznetsov et al. 2012]. In particular, it was found that merging frequently queried constraints will substantially increase the computation time for constraint solving. As a result, only those that are infrequently queried are subject to merging.

In this experiment, we adapt the implementation in Kuznetsov et al. [2012] on KLEE and compare its performance with our approach on the Linux Kernel modules. We use the same configuration (e.g., query count thresholds) as in Kuznetsov et al. [2012] for a fair comparison. The results are shown in Figures 17 and 18. In Figure 19, we show that our technique is able to achieve higher coverage than that of merging based on 24 hours of computation.

The better performance of our technique can be attributed to the following three reasons. First, large subject programs lead to complex formulas after constraint merging (as illustrated in Figure 17). These formulas are difficult to solve. As a result, a lot of execution time was spent on constraint solving (15 times more on average according

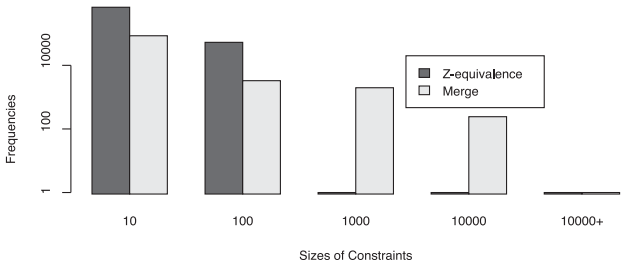

Fig. 17. Constraint size distribution (r600_init).

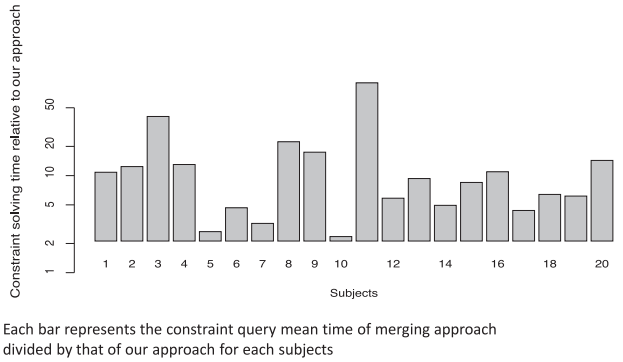

Fig. 18. Constraint query mean time.

to Figure 18). Second, these programs tend to have different heap structures along different paths such that their constraints cannot be merged.

Third, state merging based on query-count-estimation cannot resolve the path explosion issue when the constraints are indeed queried frequently, which tends to be true in complex programs. The code snippet in Figure 20 is from one of our subject modules. The variable *pr* is returned from per_cpu(). Multiple states are forked from the loop inside per_cpu(). Since *pr* is queried intensively, the states of *pr* are not merged. In contrast, our technique can help traverse such complex program blocks where constraint merging shows no advantages in this case. In this example, our technique would conclude that there are only two cases, that is, *pr* is NULL or *pr* point to one *acpi_processor* structure, since the rest of non-NULL structures are z-equivalent.

In fact, merging and z-equivalent state removal are orthogonal to each other. State removal can reduce the number of states to be merged so that the resulting constraints can be simplified.

6. RELATED WORK

A wide range of program analyses can be modeled as constraint solving problems [Gulwani et al. 2008]. Dillig et al. proposed a constraint based static analysis [2008], in which programs are modeled as a quantified and recursive formula, allowing for precise representation of path-sensitive and context-sensitive properties. They proposed the idea of separating variables to the observable and unobservable classes to prevent modeling unnecessary low-level complexity. Our technique was originally inspired by this idea. In comparison, their technique is driven by the syntactic structure of a program. Their constraints encode multiple paths in one formula. Our technique maintains multiple symbolic states or constraints, each of which corresponds to a single path. In other

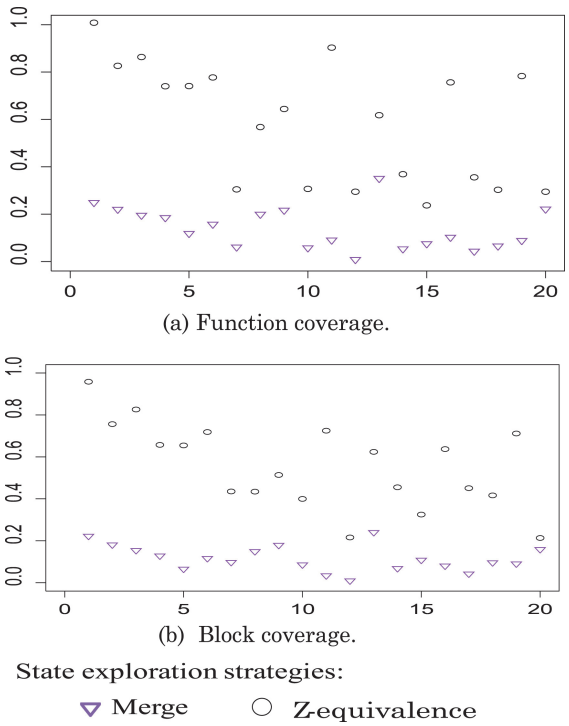

Fig. 19. Coverage comparison between merge/z-equivalence.

```
static int acpi_cpu_soft_notify(struct notifier_block *nfb,
 unsigned long action, void *hcpu)
{
unsigned int cpu = (unsigned long)hcpu;
struct acpi_processor *pr = per_cpu(processors, cpu);
if (action == CPU_ONLINE && pr) {
  if (pr->flags.need_hotplug_init) {
    //intensive query about pr
  }
 } else {
    //intensive query about pr
 }
}
if (action == CPU_DEAD && pr) {
    //intensive query about pr
}
return NOTIFY_OK;
}
```

Fig. 20. Example code.

words, our constraints are generated in a fashion similar to program execution, and the entailed z-equivalent detection is hence unique. Our technique can be applied to test generation, whereas that of Dillig et al. [2008] is mainly for static verification. Another key difference is that Dillig et al. [2008] assumes that variables have a finite domain such that each variable can only bind to a few abstract values. Finite domain assumption allows them to eliminate unobservable variables by enumerating their values to eliminate quantifiers. However, quantifier elimination for quantified Boolean formulae

is PSPACE-complete and requires an exponential complexity algorithm [Reiter and Clayton 2013]. Our approach does not assume a finite domain and allows arbitrary arithmetic operations, which makes z-equivalence widely applicable to a large set of programs. At the same time, such equivalence detection is challenging.

Symbolic analysis has aroused a lot of research interest [Cadar et al. 2008; Chandra et al. 2009; Godefroid 2007; Kothari et al. 2008; Person et al. 2008; Xie et al. 2003; Cui et al. 2013]. Babic et al. proposed improving the scalability of symbolic analysis by using structural abstraction and refinement [Babic and Hu 2008; Babic et al. 2007]. The technique achieves scalability by analyzing portions of a program, while our approach achieves scalability by detecting equivalent symbolic states. The mechanism of Babic et al.'s techique for analyzing a path segment instead of a path from a program entry makes it difficult to construct test cases that validate warnings.

Symbolic analysis has been used in test-case generation [Cadar et al. 2008; Godefroid 2007; Godefroid et al. 2005; King 1976; Kuznetsov et al. 2012; Sen et al. 2005]. The very large search space due to path explosion is a challenge when performing symbolic analysis on large programs. One approach is to distribute the task of path exploration to different machines by partitioning paths [Siddiqui and Khurshid 2012]. An alternative approach to addressing path explosion is to study the trade-off between exploring simpler execution states and fewer complex merged symbolic states [Godefroid 2007; Kuznetsov et al. 2012]. Godefroid [2007] proposed using method summaries to encode multiple program input and output mapping using disjunction of constraints. In doing so, symbolic analysis can help avoid exploring so many simple paths. However, Kuznetsov et al. [2012] pointed out that the resulting merged constraints become more complex and more difficult to solve. They proposed selective merging techniques using cost/benefit analysis to merge only some selected states, such as those that are infrequently queried. The principle of many of these of approaches [Cadar et al. 2008; Kuznetsov et al. 2012] is essentially to shifting the search space burden to the underlying solver. As program sizes become large, the resulting merged constraints can be too expensive for the solver to solve. This is one reason why existing tools [Cadar et al. 2008; Kuznetsov et al. 2012] have mostly been used for subjects with less than 10,000 lines of code, for example, Linux coreutils. The difference between these approaches and ours is that our approach removes states instead of merging them. This allows our approach to scale well with respect to increased program sizes. Boonstoppel et al. [2008] proposed a state reduction technique by discarding the comparison of those variables that are not read in the subsequent analysis. Our approach is more general: since our approach can still detect equivalence, even a symbolic variable is read in the subsequent analysis and achieves better effectiveness in terms of state reduction. Bugrara and Engler [2013] proposed using dynamic slicing to reduce symbolic states such that the technique would not explore uncovered statements. Their reduction does not guarantee soundness or completeness regarding to answering validity and satisfiability queries. Anand et al. [2009] proposed using shape analysis to perform abstraction of data structures, and their abstraction can be used as a subsumption analysis. Their approach does not guarantee soundness of reduction because their abstraction is an approximation.

In the context of regression testing, researchers try to reduce unnecessary exploration of paths that have the same effect on the modified program statements [Person et al. 2008, 2011; Santelices and Harrold 2010]. They require "sink" statements as inputs such that redundant exploration of paths that have equivalent effects to these "sink" statements are avoided. They analyze data/control dependences statically to achieve the goal.

A state matching technique has been used in Java PathFinder (JPF) [Visser et al. 2006]. Visser et al. proposed using explicit state matching and abstract state matching

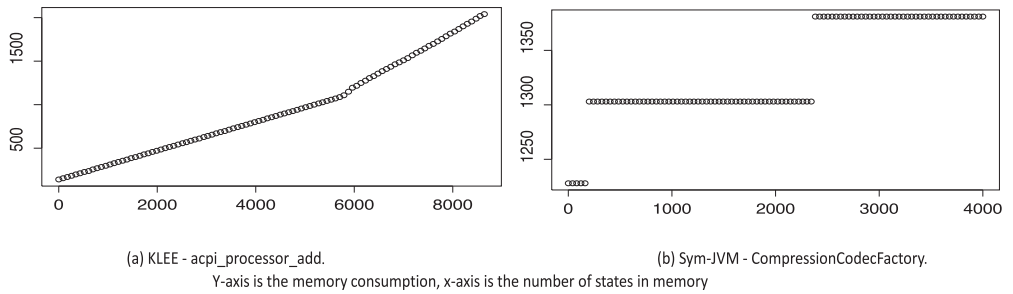

Fig. 21. Relation between number of states in memory and memory consumption in MBs.

to test containers. Explicit state matching is different from symbolic state matching proposed by our approach in that one symbolic state can represent many concrete states. Abstract state matching [Visser et al. 2006] requires domain knowledge about applications under analysis to appropriately encode such abstract states, which is not general to all applications. Our approach is general and does not require application-specific domain knowledge.

Our z-equivalence detection technique can be considered as a technique for detecting redundant constraints in the constraint satisfaction literature. However, the complexity of the general-purpose redundant constraint detection algorithm is $O(na^3)$, where n is the number of variables and a is the domain size [Tsang 1993]: note that a can be as large as 2^{32} for a 32-bit integer. Our technique achieves linear-time complexity by exploiting two types of domain knowledge: (1) uninterpreted functions and (2) classification of operators (Type I, Type II, and Type III operators).

7. CONCLUSION

We propose z-equivalence, a kind of equivalence correlation between symbolic states. It is weaker than logical equivalence and can capture a significant number of z-equivalent states in real-world applications, enabling substantial search space reduction. We prove that z-equivalence is strong enough to ensure that two z-equivalent states in the subsequent symbolic analysis must traverse the same set of paths and give exactly the same answers to the same satisfiability or validity queries. Although deciding z-equivalence is in general undecidable, we propose a linear algorithm to give a sound answer to decide z-equivalence such that our algorithm will never classify two non-z-equivalent states as equivalent. The empirical study shows that our approach is effective for reducing a significant number of z-equivalent states and allows symbolic analysis to scale up to large applications like Hadoop and Linux Kernel.

APPENDIX

A. NUMBER OF STATES IN MEMORY AND MEMORY CONSUMPTION

We recorded the number of states in memory and memory consumption for both engines. In Figure 21(a), the memory consumption for KLEE is proportional to the number of states. In Figure 21(b), the memory consumption for Sym-JVM is also related to the number of states. The memory consumption for Sym-JVM is segmented because JVM always pre-allocates memory from the OS and manages the allocation itself.

ACKNOWLEDGMENTS

The authors would like to thank Tao He for his help in conducting an initial study on KLEE.

REFERENCES

Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2009. Symbolic execution with abstraction. *Int. J. Softw. Tools Technol. Transf.* 11, 1 (2009), 53–67. DOI=10.1007/s10009-008-0090-1 http://dx.doi.org/10.1007/s10009-008-0090-1

Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. ACM, New York, NY, 211–220.

Domagoj Babic and Alan J. Hu. 2007. Structural abstraction of software verification conditions. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*. Springer, Berlin. 371–383.

Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking path explosion in constraint-based test generation. In *Proceedings of the European Joint Conference on Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. C. R. Ramakrishnan and Jakob Rehof (Eds.), Springer-Verlag, Berlin, Heidelberg, 351–366.

Suhabe Bugrara and Dawson Engler. 2013. Redundant state detection for dynamic symbolic execution. In *Proceedings of the USENIX Annual Technical Conference (ATC'13)*. USENIX Association, Berkeley, CA, 199–212.

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, 209–224.

Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. SnuggleBug: A powerful approach to weakest preconditions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, New York, NY, 363–374.

Avik Chaudhuri and Jeffrey S. Foster. 2010. Symbolic security analysis of Ruby-on-Rails Web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, New York, NY, 585–594.

Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.* 30, 1, Article 2 (2012).

L. A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.* 2, 3 (1976), 215–222.

Edmund Clarke, Daniel Kroening, and Karen Yorav. 2003. Behavioral consistency of C and verilog programs using bounded model checking. In *Proceedings of the 40th Annual Design Automation Conference (DAC'03)*. ACM, New York, NY, 368–371.

Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. 2013. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 329–342.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 270–280.

Vijay Ganesh and David L. Dill. 2007. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*. Werner Damm and Holger Hermanns (Eds.), Springer-Verlag, Berlin, Heidelberg, 519–531.

Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. ACM, New York, NY, 47–54.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 213–223.

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program analysis as constraint solving. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 281–292.

Brian Hackett and Alex Aiken. 2006. How is aliasing used in systems software? In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06)*. ACM, New York, NY, 69–80.

Michael Huth and Mark Ryan. 2004. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press.

James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*. Hubert Garavel and John Hatcliff (Eds.), Springer-Verlag, Berlin, Heidelberg, 553–568.

Nupur Kothari, Todd Millstein, and Ramesh Govindan. 2008. Deriving state machines from TinyOS programs using symbolic execution. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN'08)*. IEEE Computer Society, Los Alamitos, CA, 271–282.

Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, New York, NY, 193–204.

Yueqi Li, S. C. Cheung, Xiangxu Zhang, and Yepang Liu. 2013. Scaling up symbolic analysis by removing Beta-equivalent states. Technical Report HKUST-CS13-06. Hong Kong University of Science and Technology.

T. Murata. Petri nets: Properties, analysis and applications. 1989. *Proc. IEEE*, 77, 4, 541–580.

Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. 2012. Abstractions from tests. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM, New York, NY, 373–386.

Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08)*. ACM, New York, NY, 226–237.

Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, 504–515.

Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'08)*. ACM, New York, NY, 15–26.

Dawei Qi, William N. Sumner, Feng Qin, Mai Zheng, Xiangyu Zhang, and Abhik Roychoudhury. 2012. Modeling software execution environment. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*. IEEE Computer Society, Los Alamitos, CA, 415–424.

Edna E. Reiter and Matthew Johnson Clayton. 2013. *Limits of Computation: An Introduction to the Undecidable and the Intractable*. CRC Press

Stuart Russell and Peter Norvig. 2003. Inference in first order logic. In *Artificial Intelligence, A Modern Approach*, Prentice Hall.

Raul Santelices and Mary Jean Harrold. 2010. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*. ACM, New York, NY, 195–206.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'05)*. ACM, New York, NY, 263–272.

Junaid Haroon Siddiqui and Sarfraz Khurshid. 2012. Scaling symbolic execution using ranged analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. ACM, New York, NY, 523–536.

Edward Tsang. 1993. Problem redunction by removing redundant constraints. In *Foundations of Constraint Satisfaction*, Computation in Cognitive Science Series, Academic Pr.

Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. 2006. Test input generation for Java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'06)*. ACM, New York, NY, 37–48.

Yichen Xie, Andy Chou, and Dawson Engler. 2003. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'03)*. ACM, New York, NY, 327–336.

Received July 2013; revised February 2014; accepted March 2014