MultiSE: Multi-path Symbolic Execution using Value Summaries

Koushik Sen, George Necula, Liang Gong, Wontae Choi
EECS Department, University of California, Berkeley, USA
{ksen, necula, gongliang13, wtchoi}@cs.berkeley.edu

ABSTRACT

Dynamic symbolic execution (DSE) has been proposed to effectively generate test inputs for real-world programs. Unfortunately, DSE techniques do not scale well for large realistic programs, because often the number of feasible execution paths of a program increases exponentially with the increase in the length of an execution path.

In this paper, we propose MultiSE, a new technique for merging states *incrementally* during symbolic execution, *without using auxiliary variables*. The key idea of MultiSE is based on an alternative representation of the state, where we map each variable, including the program counter, to a set of guarded symbolic expressions called a *value summary*. MultiSE has several advantages over conventional DSE and conventional state merging techniques: value summaries enable sharing of symbolic expressions and path constraints along multiple paths and thus avoid redundant execution. MultiSE does not introduce auxiliary symbolic variables, which enables it to 1) make progress even when merging values not supported by the constraint solver, 2) avoid expensive constraint solver calls when resolving function calls and jumps, and 3) carry out most operations concretely. Moreover, MultiSE updates value summaries incrementally at every assignment instruction, which makes it unnecessary to identify the join points and to keep track of variables to merge at join points.

We have implemented MultiSE for JavaScript programs in a publicly available open-source tool. Our evaluation of MultiSE on several programs shows that 1) value summaries are an effective technique to take advantage of the sharing of value along multiple execution path, that 2) MultiSE can run significantly faster than traditional dynamic symbolic execution and, 3) MultiSE saves a substantial number of state merges compared to conventional state-merging techniques.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Symbolic execution, Testing tools, Debugging aids

Keywords

symbolic execution, MultiSE, value summary, JavaScript, test generation, concolic testing, Jalangi

1. INTRODUCTION

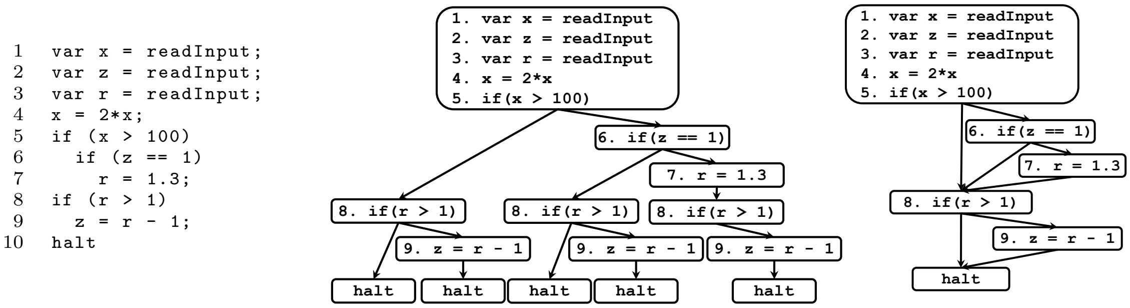
Symbolic execution is a technique for automatically generating a symbolic model from a program. It has been used successfully as a key component in a variety of applications, including generating high-coverage tests for C [20, 44, 11, 10, 16], C++ [33], C# [48], Java [3, 36, 2, 29, 42], PHP [4], JavaScript [41, 43], x86-binaries [22, 47, 5].

The key idea behind symbolic execution was introduced almost 40 years ago [30, 17]. In this paper we consider the dynamic variant of symbolic execution (DSE), in which a program is executed using symbolic values in place of concrete values for inputs.

Symbolic execution techniques do not scale for large realistic programs because often the number of feasible execution paths of a program increases exponentially with the length of an execution path. Since symbolic execution needs to explore multiple paths of a program it is crucial to attempt to mitigate this path-explosion problem. The state-of-the-art in this regard is a technique called *state merging* [19, 1, 23, 32, 5, 49], whereby the symbolic states obtained from multiple paths converging at a join point are merged. For example, consider the conditional statement "`if p then x =` v_1 `else x =` v_2". State merging would execute symbolically the two branches and at the join point would introduce a new symbolic value for the merged value of `x`, say x_0. (Such values are called *auxiliary variables* [5, 32], to differentiate them from the input symbolic values.) The symbolic state would essentially be $(\mathtt{x} = x_0) \wedge ((p \ \wedge \ x_0 = v_1) \vee (\neg p \wedge x_0 = v_2))$.

Part of the motivation for our work is that auxiliary variables introduce several problems. The most significant one is that this technique cannot be used if the merged values (e.g. v_1 and v_2) are outside the domain of the constraint solver, such as floating point values, functions as first-class values, or objects, because the logical formula mentioned above would not be legal in those cases. For example, the statement "`if p then x = 1.2 else x = function(y) { return y+1}`" in JavaScript would generate the state $(\mathtt{x} = x_0) \wedge ((p \ \wedge \ x_0 = 1.2) \vee (\neg p \wedge x_0 =$`function(y) { return y+1}`$))$ which is not a legal formula if our constraint solver does not support floating-point number or function datatypes. Existing symbolic execution techniques often

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786830



```
1   var x = readInput;
2   var z = readInput;
3   var r = readInput;
4   x = 2*x;
5   if (x > 100)
6     if (z == 1)
7       r = 1.3;
8   if (r > 1)
9     z = r - 1;
10  halt
```

Figure 1: (a) A simple program to illustrate MultiSE; (b) Conventional symbolic execution tree; (c) MultiSE execution DAG.

deal with such situations by avoiding merging and discarding one of the paths. Furthermore, in the case when the values being merged are functions (treated as first-class values), or computed jump labels, and the merged value is invoked, the symbolic execution needs to determine where to continue the execution. These kinds of operations are quite common in dynamically typed programming languages such as JavaScript, Python, and Ruby. Existing techniques use an SMT solver to try to determine what is the set of possible functions to be invoked.

In this paper we propose MultiSE, a new symbolic execution framework that achieves state merging *without using auxiliary variables*. MultiSE is based on a new representation of the state where we map each variable, *including the program counter*, to a set of guarded symbolic expressions called a *value summary*. For our example, the value summary representation of the example state above would be: $\texttt{x} \mapsto \{(p, v_1), (\neg p, v_2)\}$.

Essentially, this representation encodes the fact that the value of `x` is equal to v_1 if the predicate p holds, and to v_2 otherwise. The key idea is to separate the path constraints from the values of the variables. The path constraint part (e.g. p and $\neg p$) of a value summary is restricted to formulas within the domain of a constraint solver; however, the value part (e.g. v_1 and v_2) in a value summary can be any symbolic expression or concrete program object including floating point numbers, functions. While this may seem like a small change, it has important effects:

1. MultiSE can often carry out the symbolic execution after a merge even when some of the values being merged are not supported by the constraint solver, while conventional state merging would have to drop some paths. This is true as long as the path constraint part of the value summary does not involve unsupported value types. The constraint solver is only used on the path constraint component of the state, e.g., to determine unfeasible paths, or to produce satisfying values for the inputs that enable one particular path. We show in Section 5 that about half of our benchmarks would require auxiliary variables of type other than integer or string if executed with conventional state merging, sometimes in the thousands, for up to 60% of the state merges; conventional state merging would have to discard feasible paths at all these merges, while MultiSE can proceed without dropping paths.

2. MultiSE can take more often advantage of a common optimization in symbolic execution: carry out concrete computation whenever possible. Since MultiSE does not introduce auxiliary variables, the values in a value summary are concrete as long as they are not data dependent on inputs. In our example, if v_1 is 0.1 and v_2 is 0.2, and the conditional statement is followed by "`x = x + 1`", MultiSE would perform the increment operation concretely, and yield the value summary $\{(p, 1.1), (\neg p, 1.2)\}$ instead of the symbolic state $(\texttt{x} = x_0 + 1) \wedge ((p \ \wedge \ x_0 = 0.1) \vee (\neg p \wedge x_0 = 0.2))$ where the value of `x` is computed symbolically. This optimization could potentially be applied to any data types such as functions, arrays, floating-point numbers, and objects etc; in such cases, we do not need to worry about unsupported symbolic expressions which would have arisen had there been auxiliary variables of these data types.

3. Conventional state merging must scan the entire two states to be merged at a join point in order to construct the merged state. In contrast, MultiSE uses a novel algorithm for updating the value summaries as it processes each instruction and achieves the effect of merging implicitly and incrementally, even in presence of unstructured or computed control flow.

We have implemented MultiSE for JavaScript programs in a publicly available open-source tool[1]. We use binary decision diagrams (BDDs) [9] to concisely represent and to efficiently manipulate path constraints and guards of value summaries. Our evaluation of MultiSE on several programs shows that MultiSE can run significantly faster than conventional DSE and achieve more precise state merging than conventional state-merging techniques.

2. OVERVIEW

We introduce the concepts of conventional symbolic execution and its state representation informally and then we describe the main elements of the MultiSE symbolic execution. We will use the program in Figure 1(a) as a running example. A statement of the form `var v = e;` declares and initializes a variable `v` with the value of the expression `e`. The execution of the statement `var x = readInput;` receives an integer input from the environment and assigns it to the variable `x`.

[1] `https://github.com/SRA-SiliconValley/jalangi` under the branch `symfront`.

2.1 Conventional DSE & State Merging

Dynamic Symbolic Execution (DSE) uses symbolic expressions for the program variables and memory locations. These expressions are in terms of fresh symbolic values that are introduced upon execution of `readInput` expressions. DSE executes one path at a time, and it maintains the current symbolic state that includes: the program counter, a mapping of program variables to symbolic expressions, and a symbolic path constraint ϕ, which is a quantifier-free propositional formula over symbolic expressions.

path	ϕ	pc	`x`	`z`	`r`
1-5	*true*	5	$2x_0$	z_0	r_0

For example, after executing statements 1–4 from our example the symbolic execution state is as shown on the left, where pc denotes the program counter, x_0, z_0, and r_0 are the symbolic values introduced for the result of the `readInput` expressions in lines 1–3, respectively. Each row in these tables corresponds to the symbolic execution state of a path.[2] Informally, for any concrete input values (concrete values for the symbolic values) that satisfy the path constraint, the concrete execution on those input values will follow the path given in the table.

Upon encountering a branch and if both sides of the branch are feasible, DSE replaces the current symbolic state with two copies of the state with updated values of pc and of the path constraints.

At every step, DSE will pick one state from the consolidated state, and will update the values of variables and the value of the program counter according to the statement at the program counter for that state.

path	ϕ	pc	`x`	`z`	`r`
1-5,8-10	$\phi_1 = 2x_0 \leq 100 \wedge r_0 > 1$	10	$2x_0$	$r_0 - 1$	r_0
1-5,8,10	$\phi_2 = 2x_0 \leq 100 \wedge r_0 \leq 1$	10	$2x_0$	z_0	r_0
1-5,6,8-10	$\phi_3 = 2x_0 > 100 \wedge z_0 \neq 1 \wedge r_0 > 1$	10	$2x_0$	$r_0 - 1$	r_0
1-5,6,8,10	$\phi_4 = 2x_0 > 100 \wedge z_0 \neq 1 \wedge r_0 \leq 1$	10	$2x_0$	z_0	r_0
1-10	$\phi_5 = 2x_0 > 100 \wedge z_0 = 1$	10	$2x_0$	0.3	1.3

Eventually, DSE will finish exploring all states, and will terminate with the consolidated state shown here, with each of the five states corresponding to one of the five feasible paths shown in Figure 1(b).

State Merging techniques identify join points in the control-flow graph and merge two states by introducing auxiliary variables to represent different values for the same variable in the merged states. For example, consider the merge after the conditionals in line 6 in Figure 1(a). State merging would execute symbolically the two sides of the conditional and at the join point would introduce a new symbolic value for the merged value of `r`, say r_1. (Such values are called *auxiliary variables*, to differentiate them from the input symbolic values.) The symbolic state would essentially be $(\texttt{r} = r_1) \wedge ((2x_0 > 100 \wedge z_0 = 1 \;\wedge\; r_1 = 1.3) \vee (2x_0 > 100 \wedge z_0 \neq 1 \wedge r_1 = r_0))$. This form of traditional state merging does have the effect of reducing the number of operations executed symbolically (e.g., the operations that follow the conditional are executed only once even though they appear in multiple paths). However, it has the unfortunate effect that it cannot be used when the values being merged are of types, e.g., floating point, not supported by the underlying constraint solver.

2.2 MultiSE Value-Summary Representation

The MultiSE representation of the symbolic execution state is based on the key observation that by considering a consolidated view of the execution state, we expose a significant opportunity for sharing of path constraints and symbolic expressions.

Consider the final consolidated state of DSE, as shown above. We can obtain a more compact representation if we represent it by variables, i.e., by columns. For each variable, and for each distinct symbolic expression of the variable, we construct the disjunction of the corresponding path constraints. For example, for pc the only symbolic expression is 10 with the disjunction of path constraints $\phi_1 \vee \phi_2 \vee \phi_3 \vee \phi_4 \vee \phi_5$ which is equivalent to *true*. Consequently, we represent the consolidated value of pc as the pair $(\mathit{true}, 10)$. We call such a pair, a *guarded symbolic expression.* For variables that take different symbolic expressions on different paths we represent their value as a set of pairs, with one pair for every distinct symbolic expression. We call such a set of guarded symbolic expressions a *value summary.* The MultiSE state is a mapping that maps each variable to a value summary. The path constraints of different guarded expressions for a given variable are disjoint and their disjunction is *true*. The MultiSE representation of the final state for our example program is:

$$\begin{aligned} \{ \quad & pc \mapsto \{(\mathit{true}, 10)\}, \quad \texttt{x} \mapsto \{(\mathit{true}, 2x_0)\}, \\ & \texttt{z} \mapsto \{(\phi_1 \vee \phi_3, r_0 - 1), (\phi_2 \vee \phi_4, z_0), (\phi_5, 0.3)\}, \\ & \texttt{r} \mapsto \{(\neg\phi_5, r_0), (\phi_5, 1.3)\}, \} \end{aligned} \tag{Final State}$$

A MultiSE final state describes compactly the final values of all variables in all feasible concrete executions. Given any assignment of integer input values to the symbolic values corresponding to the program inputs, exactly one of the path constraints will hold for each variable. The corresponding symbolic expression, evaluated at the given program inputs, gives the value of the variable at the end of the execution of the program on the given program inputs.

There are several advantages to the MultiSE value-summary representation. The obvious one is that often times a state can be represented in a concise form due to the following three observations:

- if s is a value summary and (ϕ, v) and (ϕ', v') are any two distinct elements of s such that $v = v'$, then we can replace the two elements with $\{(\phi \vee \phi', v)\}$ to obtain the equivalent value summary $s \setminus \{(\phi, v), (\phi', v')\} \cup \{(\phi \vee \phi', v)\}$.

- if (false, v) is an element of a value summary, then it can be removed from the value summary to get an equivalent value summary.

- each guard in a value summary can be represented and manipulated efficiently using a binary decision diagram (or a BDD).

As we will show in our experiments there is a significant amount of sharing for the symbolic expressions of variables among the many execution paths.

The less obvious but more important advantage of value summary is that this representation achieves a natural form of state merging, which in turn can reduce dramatically the number of statements executed symbolically, as we discuss in the next section and we show experimentally in Section 4.

[2] The `path` component of the state is shown here for clarity, but is not explicitly maintained during symbolic execution.

2.3 MultiSE: Symbolic Execution with Value Summaries

To illustrate the operation of MultiSE, consider the state when the symbolic execution has explored all three paths up to the conditional in line 8. For a conventional DSE the state would be as shown below.

path	ϕ	pc	x	z	r
1-5,8	$2x_0 \leq 100$	8	$2x_0$	z_0	r_0
1-5,6,8	$2x_0 > 100 \wedge z_0 \neq 1$	8	$2x_0$	z_0	r_0
1-5,6-8	$\phi_5 = 2x_0 > 100 \wedge z_0 = 1$	8	$2x_0$	z_0	1.3

This state representation with three separate rows corresponds to the three separate instances of execution paths ending at the statement at line 8 shown in DSE execution tree from Figure 1(b).

The corresponding MultiSE value-summary representation of this state is:

$$\{\quad pc \mapsto \{(true, 8)\},\quad \mathtt{x} \mapsto \{(true, 2x_0)\},\\ \mathtt{z} \mapsto \{(true, z_0)\},\quad \mathtt{r} \mapsto \{(\neg\phi_5, r_0), (\phi_5, 1.3)\}\quad \}$$

(Intermediate State 8)

This value summary represents a merge of the three separate conventional DSE states, corresponding to three separate executions paths. Continuing with this state allows MultiSE to evaluate the conditional in line 8 twice (i.e. once for each value of `r` in the value summary), instead of three times for conventional DSE, as shown in the MultiSE execution DAG (directed acyclic graph) shown in Figure 1(c).

MultiSE symbolic execution first considers the value summary for the program counter. It picks one of the values, in this case 8, guarded by the path constraint *true*, and executes the statement "`if (r > 1) ...`". This requires the computation of the value of the expression `r > 1`.

The symbolic execution of the expression `r > 1` goes over each guarded expression in the value summary for variable `r`, applies the operation $>$ on the expression part of each guarded expression, and computes the value summary $\{(\neg\phi_5, r_0 > 1), (\phi_5, true)\}$. Note that the second guarded expression for `r > 1` contains the symbolic expression "*true*", which is obtained from $1.3 > 1$. MultiSE eagerly simplifies the parts of symbolic expressions that do not depend on symbolic values. This optimization would often not be possible in traditional state merging in presence of auxiliary variables.

Next MultiSE processes the actual conditional statement. We compute the condition for the computed value of `r > 1` to be *true*, as a disjunction over the guarded expressions in the value summary for `r > 1`. We prepend also a conjunction for the current path constraint (*true*) as follows:

$$\phi_6 = true \wedge ((\neg\phi_5 \wedge r_0 > 1) \vee (\phi_5 \wedge true))$$

Therefore, after the execution of the conditional statement at line 8, in the new state pc maps to the value summary $\{(\phi_6, 9), (\neg\phi_6, 10)\}$, where $\neg\phi_6$ is logically equivalent to $\neg\phi_5 \wedge r_0 \leq 1$, the condition for the computed value of `r > 1` to be *false*. The value summary representing compactly *both* the "then" and the "else" branches can be written as:

$$\{\quad pc \mapsto \{(\phi_6, 9), (\neg\phi_6, 10)\},\quad \mathtt{x} \mapsto \{(true, 2x_0)\},\\ \mathtt{z} \mapsto \{(true, z_0)\},\quad \mathtt{r} \mapsto \{(\neg\phi_5, r_0), (\phi_5, 1.3)\}\quad \}$$

(Intermediate State 9+10)

This value summary represents five paths, three of which end at line 9 with combined path constraint ϕ_6 after taking the "then" branch at line 8, and the remaining two paths end at line 10 with the combined path constraint $\neg\phi_6$.

One of the novel aspects of MultiSE is that it performs *incremental state merging* at every assignment statement to obtain a new consolidated representation of states using value summaries. To illustrate this aspect, we continue with the above MultiSE state. Say that for the program counter, MultiSE picks the guarded value $(\phi_6, 9)$ and executes line 9 next, with the path constraint ϕ_6. First, we symbolically evaluate the right-hand side of the assignment (`r - 1`), and we obtain the guarded value: $\{(\neg\phi_5, r_0 - 1), (\phi_5, 0.3)\}$

Since line 9 is guarded by the path constraint ϕ_6, symbolic execution of the assignment `z = r - 1` should only update the value of `z` for those paths for which ϕ_6 is *true*. The value of `z` must remain unchanged in the symbolic state for the other paths. This is achieved by computing the new value of `z` using a *guarded value-summary union*, where we preserve the previous value of `z` with the additional guard $\neg\phi_6$ (the negation of the current path constraint) to which we add the value summary for the right-hand side with the additional guard ϕ_6. By applying a conjunction of $\neg\phi_6$ to the guards of the current value summary stored in `z`, we keep unchanged the portion of the value summary for the other two paths (whose combined path constraint is $\neg\phi_6$).

The value summary for `z` is: $\{(\neg\phi_6, z_0), (\phi_6 \wedge \neg\phi_5, r_0 - 1), (\phi_6 \wedge \phi_5, 0.3)\}$ which is logically equivalent with the value summary we have used in (Final State) for the final value of `z`. Finally, the value summary stored in pc is also updated to $\{(\neg\phi_6, 10), (\phi_6, 10)\}$, which simplifies to $\{(true, 10)\}$. Therefore, after the execution of the statement `z = r - 1` at line 9, the state becomes the same as the (Final State).

The above description shows how MultiSE performs incremental state merging at every assignment statement. Note that in conventional state-merging techniques, at join points, state merging needs to iterate over the part of the symbolic state that has been modified by the paths converging at the join point and merge that part of the state. Identifying the join points, keeping track of the modified part of the state, and merging the modified state could pose various implementation challenges. In contrast, MultiSE does not require to perform state merging at join points because the MultiSE state is at *all times consolidated* over all the paths being explored.

Every time a new guarded symbolic expression is added to the value summary for pc, MultiSE invokes a quick BDD satisfiability check followed by an SMT solver satisfiability check for the path constraint. This is important in order to avoid exploring unfeasible paths. For the value summaries of other variables, only a BDD satisfiability check is used, to reduce the overall cost of SMT solving, which is a significant fraction of the overall cost.

2.4 Advantages of MultiSE

Value Summary representation is a powerful way of sharing symbolic expressions and path constraints among different paths. This avoids redundant computation and alleviates the path-explosion problem compared to DSE.

Incremental State Merging avoids scanning the two states to be merged. Conventional state merging techniques iterate over the part of the symbolic state that has been modified by the paths converging at the join point and merge that part of the state. Identifying the join points, keeping track of the modified part of the state, and merging the modified state could pose various implementation challenges which are not present in MultiSE.

Furthermore, all paths share consolidated state in MultiSE, even for programs where *the join points are not known statically*, such as programs with exceptions, or computed control-flow, or for binary programs with unstructured control-flow where the join points are non-trivial to compute.

Auxiliary Variables are avoided in MultiSE. This has three advantages:

1. Execution can proceed even if certain theories are not supported by the constraint solver.
2. There is no need for expensive constraint solver calls where conventional state merging introduces auxiliary symbolic values for functions and subsequently functions denoted by those auxiliary symbolic values are called.
3. Execution can carry out most operations concretely.

path	ϕ	*pc*	x	z	r
1-5,8	$2x_0 \leq 100$	8	$2x_0$	z_0	r_0
1-5,6,8	$2x_0 > 100 \land z_0 \neq 1$	8	$2x_0$	z_0	r_0
1-5,6-8	$\phi_5 = 2x_0 > 100 \land z_0 = 1$	8	$2x_0$	z_0	1.3

Existing techniques introduce auxiliary symbolic values to represent the value of a variable computed along two or more paths merging at a point. For example, consider the intermediate DSE state (above) of the example program at line 8 where three paths merge.

1. Proceed with Unsupported Theory: The symbolic expressions for the variable `r` along the three paths are not all the same. Conventional DSE stores only one symbolic expression for each variable. Therefore, conventional state merging would introduce an auxiliary variable r_1 to denote the value of `r`, and would add to the path constraint the relationship between r_1 and the different symbolic expressions for `r` along the merged paths, as:

path	ϕ	*pc*	x	z	r
...,8	$((2x_0 \leq 100 \land r_1 = r_0)$ $\lor(2x_0 > 100 \land z_0 \neq 1 \land r_1 = r_0)$ $\lor(2x_0 > 100 \land z_0 = 1 \land r_1 = 1.3))$	8	$2x_0$	z_0	r_1

The problem is that the new path constraint containing the auxiliary variable has a predicate $r_1 = 1.3$. However, if the constraint solver does not support floating point constraints, then symbolic execution cannot merge the paths to generate a path constraint that is beyond the scope of the constraint solver.[3]

In MultiSE, we never introduce auxiliary symbolic values (See Intermediate State 8 in previous pages). Therefore, path constraints in MultiSE are always formulas over the input symbolic values, which we restrict to integer and string types. Concrete values of data types that are not supported by the constraint solver remain in the state as concrete values guarded by symbolic predicates. This also implies MultiSE can perform more operations concretely than existing techniques, as demonstrated below.

2. Avoid Constraint Solver Calls: The fact that MultiSE does not introduce auxiliary symbolic values while merging paths also helps MultiSE to efficiently handle function values, which are often first-class objects in dynamic languages such as JavaScript, Python, and Ruby. We illustrate this using the following program:

```
1  var x = readInput;
2  var f, r = 0;
3  if (x > 0)
4    f = function f1() { return 1;}
5  else
6    f = function f2() { return -1;}
7  r = f();
```

In the program, `x` gets an input from the environment. Depending on whether `x>0`, `f` is assigned the function `f1` or `f2`. Then the function stored in `f` is called and the value returned by the call is stored in `r`.

path	ϕ	*pc*	x	f	r
1-3,4,7	$x_0 > 0$	7	x_0	f1	0
1-3,6,7	$x_0 \leq 0$	7	x_0	f2	0

Consider a conventional DSE state (shown on the left) with two paths both of which end at line 7.

If we merge the two paths using existing path merging techniques, then the state becomes:

path	ϕ	*pc*	x	f	r
...,7	$((x_0 > 0 \land f_0 = \texttt{f1})$ $\lor(x_0 \leq 0 \land f_0 = \texttt{f2}))$	7	x_0	f_0	0

Merging introduces an auxiliary variable f_0 and the path constraint now refers to the function objects `f1` and `f2`. If we treat `f1` and `f2` as symbolic references to the two functions, then when symbolic execution executes the statement `r = f()` at line 7, it needs to resolve what are the possible function values that may be invoked. This is typically done by invoking a constraint solver to find all satisfying assignments to f_0 given the path constraint [5]. Invoking a constraint solver to obtain all satisfying assignments is expensive.

MultiSE requires no such constraint solving as it explicitly stores both `f1` and `f2` as separate guarded expressions in the value summary denoted by `f`. Specifically, the state of MultiSE will be:

$$\{\ pc\ \mapsto 7,\ \texttt{r} \mapsto 0,\ \texttt{x} \mapsto x_0,\ \texttt{f} \mapsto \{(x_0 > 0, \texttt{f1}), (x_0 \leq 0, \texttt{f2})\}\}$$

In this state, and others that follow, we simplify the notation and drop the constraint *true* from a guarded expression. Symbolic execution of the statement `r = f();` will then create two paths corresponding to the invocation of the two functions stored in the value summary denoted by `f`. MultiSE's mechanism of explicitly storing all uninterpreted objects as values in value summaries allows us to avoid repeated constraint solver calls.

```
1  var r = [2]; // array
2  var s = [3]; // array
3  var x = readInput;
4  if (x > 0)
5    t = r;
6  else
7    t = s;
8  t[0] = 4;
```

3. Concrete Execution: Keeping the values along different paths separate is very helpful when dealing with memory addresses and pointers, since it allows MultiSE to maintain in a natural way the set of memory addresses that a variable may point to, which in turn will make it possible to lookup and update memory addresses directly in many cases. Consider the example program to the right of this paragraph.

At each array allocation, MultiSE returns a new concrete memory address, such as a_0 and a_1 in this example[4], and keeps value summaries for the value stored at each address symbol, just as for variables. The MultiSE state before the store statement in line 8 will be:

$$\{\quad pc\ \mapsto 8,\ \texttt{x} \mapsto x_0,\ \texttt{r} \mapsto a_0,\ \texttt{s} \mapsto a_1,\ a_0 \mapsto 2,\ a_1 \mapsto 3,\\ \texttt{t} \mapsto \{(x_0 > 0, a_0), (x_0 \leq 0, a_1)\}\}$$

[3] The same problems arise if we write the path constraint using *ITE* (if-then-else): $r_1 = ITE(2x_0 \leq 100, r_0, ITE(z_0 \neq 1, r_0, 1.3))$.

[4] a_0 and a_1 are not auxiliary symbolic values. a_0 and a_1 denote the concrete addresses of the arrays allocated at line 1 and 2, respectively.

$$\begin{array}{lll}
\mathit{Pgm} & ::= & (\ell\colon\ \mathit{stmt}\ ;)^* \\
\mathit{stmt} & ::= & x = c \\
 & & x = \ \mathit{readInput} \\
 & & z = x \bowtie y \\
 & & \text{if } x \text{ goto } y \\
 & & y = *x \\
 & & *x = y \\
 & & \text{error} \\
 & & \text{halt}
\end{array}$$

where

V	is a set of variables
C	is the set of constants and statement labels
A	is a set of memory addresses
x, y, z	are elements of V
pc	an element of V denoting the program counter
c	is an element of $C \cup A \cup L$
ℓ	is an element of L
$\bowtie$	is a binary operator

Figure 2: Syntax of a simple imperative language

When processing the store statement on line 8, MultiSE will resolve the address being written (`t[0]`) to either $*a_0$ or $*a_1$. Thus the value summaries for a_0 and a_1 are modified to contain a combination of their previous values and their values as updated by the store statement. After processing the store statement on line 8, the state becomes:

$$\{\ pc\ \mapsto 9, \mathtt{x} \mapsto x_0, \mathtt{r} \mapsto a_0, \mathtt{s} \mapsto a_1, \mathtt{t} \mapsto \{(x_0 > 0, a_0), (x_0 \le 0, a_1)\}, \\ a_0 \mapsto \{(x_0 > 0, 4), (x_0 \le 0, 2)\}, a_1 \mapsto \{(x_0 \le 0, 4), (x_0 > 0, 3)\}\}$$

When $x_0 > 0$, the variable t contains address a_0, which is updated to 4 under this path constraint. In the alternative, t contains address a_1 (updated to 4).

This allows MultiSE value summaries to maintain precise aliasing information and to perform strong updates and strong reads, updating and accessing directly the memory locations that may be involved in the memory operation, without having to resort to encoding constraints for the theory of arrays. This is in contrast with state merging techniques that use auxiliary variables. For example, if the value of `t` is represented using the auxiliary variable t_1, along with the constraint $(x_0 > 0 \wedge t_1 = a_0) \vee (x_0 \le 0 \wedge t_1 = a_1)$, then symbolic execution would have to either invoke a solver to enumerate the possible addresses that t_1 refers to, or must defer the reasoning about the memory operations to a solver using the theory of arrays. We show in Section 5 that about half of our benchmarks would require auxiliary variables of type other than integer or string if executed with conventional state merging, sometimes in the thousands, for up to 60% of the joins; MultiSE avoids all these problematic auxiliary variables.

3. ALGORITHM

In this section, we formally describe MultiSE using a simple programming language.

3.1 Syntax

The syntax of the language is shown in Figure 2. A program in the language is a sequence of labelled statements. We use $x = \ \mathit{readInput}$ to denote that x gets an input from the environment. $*x$ denotes the memory cell whose address is stored in x. The language is similar to a simple untyped assembly language. Objects, references, and functions can be modeled using memory and memory address arithmetic: the heap grows from lower address to higher addresses and the call stack grows from higher address to lower addresses. Structured and unstructured control-flow, as well as exceptions, jump tables, can be modeled using if x goto y with computed jumps. Variables can be thought as the registers of the machine. The special variable pc contains the program counter and ℓ_0 is the label of the first statement of the program.

3.2 MultiSE Symbolic Execution Semantics

We use the following notations to describe the semantics of MultiSE execution:

- S is the set of symbolic input values,
- E is the set of all symbolic expressions built using the binary operators $\bowtie$ over elements of S, constants C, addresses A, and labels L,
- F is the set of all propositional logical predicates over elements of E; we use ϕ, ϕ', ϕ_i to denote a predicate in F,
- If ℓ is a statement label, then $\mathit{Pgm}\ (\ell)$ returns the statement in the program whose label is ℓ.

The state of MultiSE is denoted by a mapping for variables and addresss to *value summaries*. A value summary is a set of guarded symbolic expressions, each consisting of a symbolic predicate along with a symbolic expression:

$$\Sigma \in (A \cup V) \to 2^{F \times E}$$

The predicate in a pair of a value summary is called a *path constraint*. Note that the program counter is represented as any other variable, which allows MultiSE to deal naturally with computed control flow constructs.

A key advantage of using a value summary is that often times a state can be represented in a concise form due to the following three observations:

- if s is a value summary and (ϕ, v) and (ϕ', v') are any two distinct elements of s such that $v = v'$, then we can replace the two elements with $\{(\phi \vee \phi', v)\}$ to obtain the equivalent value summary $s \setminus \{(\phi, v), (\phi', v')\} \cup \{(\phi \vee \phi', v)\}$.
- if (false, v) is an element of a value summary, then it can be removed from the value summary to get an equivalent value summary.
- each guard in a value summary can be represented and manipulated efficiently using a binary decision diagram (or a BDD), which we discuss in detail later in the paper.

We take advantage of the above simplification rules by way of a special value-summary union operation. We write $s1 \uplus s2$ for the value summary obtained from the union of $s1$ and $s2$ followed by removing guarded expressions with guards that are unsatisfiable, i.e. guards that are equivalent to *false*, and replacing guarded expressions with the same symbolic expression with a single guarded expression using the union of the guards. Note that with an alternative implementation of $\uplus$ that does not do coalescing of repeated symbolic expressions we obtain an algorithm that operates essentially like conventional DSE.

Guarded Update

$$\frac{}{\{(\phi_i^a, v_i^a)\}_i \uplus_\phi \{(\phi_j^b, v_j^b)\}_j = \{(\neg\phi \wedge \phi_i^a, v_i^a)\}_i \uplus \{(\phi \wedge \phi_j^b, v_j^b)\}_j}$$

NextPC

$$\frac{}{\mathit{NextPC}(\Sigma, \phi, \ell) = (\Sigma(\ \mathit{pc}\ ) \setminus \{(\phi, \ell)\}) \uplus \{(\phi, \ell + 1)\}}$$

Constant

$$\frac{(\phi, \ell) \in \Sigma(\ \mathit{pc}\ ) \qquad \mathit{Pgm}(\ell) = (x = c)}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_\phi \{(\mathit{true}, c)\}][\ \mathit{pc}\ \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Symbolic Input

$$\frac{(\phi, \ell) \in \Sigma(\ \mathit{pc}\ ) \qquad \mathit{Pgm}(\ell) = (x =\ \ \mathit{readInput}\ ) \qquad s \text{ is a fresh symbolic value from } S}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_\phi \{(\mathit{true}, s)\}][\ \mathit{pc}\ \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Binary Operation

$$\frac{(\phi, \ell) \in \Sigma(\ \mathit{pc}\ ) \qquad \mathit{Pgm}(\ell) = (z = x \bowtie y) \qquad \Sigma(x) = \{(\phi_i^x, v_i^x)\}_i \qquad \Sigma(y) = \{(\phi_j^y, v_j^y)\}_j \qquad \phi_{ij}^{x \bowtie y} = \phi_i^x \wedge \phi_j^y \qquad v_{ij}^{x \bowtie y} = v_i^x \bowtie v_j^y}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \uplus_\phi \{(\phi_{ij}^{x \bowtie y}, v_{ij}^{x \bowtie y})\}_{ij}][\ \mathit{pc}\ \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Conditional

$$\frac{(\phi, \ell) \in \Sigma(\ \mathit{pc}\ ) \;\; \mathit{Pgm}(\ell) = (\text{ if } x \text{ goto } y) \;\; \Sigma(x) = \{(\phi_i^x, v_i^x)\}_i \;\; \Sigma(y) = \{(\phi_j^y, \ell_j^y)\}_j \;\; s = \{(\phi_i^x \wedge v_i^x \wedge \phi_j^y, \ell_j^y)\}_{ij} \uplus \{((\phi_i^x \wedge \neg v_i^x), \ell + 1)\}_i}{\Sigma \longrightarrow \Sigma[\ \mathit{pc}\ \mapsto (\Sigma(\ \mathit{pc}\ ) \setminus \{(\phi, \ell)\}) \uplus_\phi s}$$

Load

$$\frac{(\phi, \ell) \in \Sigma(\ \mathit{pc}\ ) \qquad \mathit{Pgm}(\ell) = (y = *x) \qquad \Sigma(x) = \{(\phi_i^x, v_i^x)\}_i \qquad \Sigma(v_i^x) = \{(\phi_{ij}, v_{ij})\}_j}{\Sigma \longrightarrow \Sigma[y \mapsto \Sigma(y) \uplus_\phi \{(\phi_i^x \wedge \phi_{ij}, v_{ij})\}_{ij}][\ \mathit{pc}\ \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Store

$$\frac{(\phi, \ell) \in \Sigma(\ \mathit{pc}\ ) \qquad \mathit{Pgm}(\ell) = (*x = y) \qquad \Sigma(x) = \{(\phi_i^x, v_i^x)\}_i \qquad \Sigma(y) = \{(\phi_j^y, v_j^y)\}_j}{\Sigma \longrightarrow \Sigma[v_i^x \mapsto \Sigma(v_i^x) \uplus_{\phi \wedge \phi_i^x} \{(\phi_j^y, v_j^y)\}_j)]_i[\ \mathit{pc}\ \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Figure 3: Alternative symbolic execution semantics using value summaries

Figure 3 gives the operational semantics of MultiSE symbolic execution as a transition relation between MultiSE states:

$$\Sigma \longrightarrow \Sigma'$$

The execution starts from an initial state that maps each variable, except pc, to the value summary $\{(\mathit{true}, \bot)\}$, where ($\bot$ denotes the undefined value), and maps pc to $\{(\mathit{true}, \ell_0)\}$, where ℓ_0 denotes the first statement label.

The crucial operation used in the definition of the MultiSE algorithm is $s_1 \uplus_\phi s_2$, which given two value summaries s_1 and s_2 computes a value summary that should behave as s_1 on paths where $\neg\phi$ holds, and as s_2 on paths where ϕ holds. This function is defined in the rule Guarded Update.

The NextPC defines the function *NextPC* that is used to update the value summary for the program counter when advancing to the next statement. The Constant and Symbolic Input are simple rules that update the value of the assigned variable and the program counter. As for all assignments, we use the function $\uplus_\phi$ to ensure that we represent the fact that the assignment takes effect only on paths that satisfy the current path constraint ϕ.

The rule Binary Operation triggers for a statement of the form $z = x \bowtie y$. The value summary for the right-hand side is computed by combining each symbolic expression for the variable x with each symbolic expression for the variable y.

The Conditional rule is a bit more involved. For a computed jump of the form if x goto y we compute a value summary s for the possible destination labels, including the cases when the jump is taken and those when it is not. For the cases when the jump is taken we consider every combination of the value summaries for x and y, adding to the path constraint the condition that the symbolic expression for x holds. For the cases when the jump is not taken, we consider every guarded expression in the value summary for x, along with the condition that x is *false*. Finally, as shown in the conclusion of the rule, we do a guarded union of this set with the existing value summary for pc.

The Load rule shows the lookup operation. For the statement $y = *x$, we first consider the value summary for `x` to obtain the possible addresses that `x` refers to. Then, we get the value summaries for these addresses as the value of $*x$.

The Store rule for statement $*x = y$ also considers first the value summary for `x` to obtain the possible addresses being written. Each of these addresses is updated with the value summary for `y`. Note the guard $\phi \wedge \phi_i^x$ in the guarded update for the address v_i^x, to model accurately the condition under which v_i^x should be updated.

3.3 Approximation in MultiSE

There are several situations when MultiSE may need to approximate a concrete execution, in the sense that not all concrete execution paths will be represented in the symbolic state.

First, if the program contains a loop or a recursive function, and the loop termination condition or the recursion base case are input dependent, then MultiSE symbolic execution could run forever. In such cases we may want to stop the symbolic execution after a certain number of iterations. This is a typical problem with any kind of symbolic execution. This can be handled in MultiSE by simply dropping guarded symbolic expressions from the value summary of the program counter, e.g., when a label has been visited more than a certain number of times.

Second, it is possible for MultiSE to generate a symbolic expression that is outside the scope of the theories supported by the associated SMT solver, e.g., a product of symbolic expressions (if we assume that the associated SMT solver cannot handle non-linear arithmetic). Consider, for example, the following MultiSE state:

$$\begin{aligned} \{ \quad & \mathit{pc}\ \mapsto \{\ldots, (\phi, \ell), \ldots\}, \\ & \mathtt{x} \mapsto \{(\phi_x, 2), (\neg\phi_x, x_0)\}, \mathtt{y} \mapsto \{(\phi_y, 3), (\neg\phi_y, y_0)\}\} \end{aligned}$$

This state suggests that the variables `x` and `y` have been initialized with constants on some paths and with `readInput` on other paths. The label ℓ, pointing to statement `z = x * y`, is reached under path constraint ϕ. When evaluating the binary expression `x * y` under path constraint ϕ, MultiSE combines the symbolic expressions from the value summaries of x and y, and one of the resulting guarded expressions will be $(\neg\phi_x \wedge \neg\phi_y, x_0 * y_0)$. If we assume that non-linear arithmetic is not supported by our SMT solver, MultiSE approximates it as follows. First, we find a satisfying assignment for $\phi \wedge \neg\phi_x \wedge \neg\phi_y$, from which we extract a possible concrete value for x, e.g., $x_0 = 5$. At this point we approximate by dropping from further consideration the concrete paths where $\phi \wedge \neg\phi_x \wedge \neg\phi_y \wedge x_0 \neq 5$. We do this by refining the path constraint for pc to $\phi \wedge (\phi_x \vee \phi_y \vee x_0 = 5)$, to obtain the following MultiSE symbolic state:

$$\begin{aligned}
\{ \quad & pc \mapsto \{\ldots, (\phi \wedge (\phi_x \vee \phi_y \vee x_0 = 5), \ell + 1), \ldots\}, \\
& \mathtt{z} \mapsto \{\ (\phi_x \wedge \phi_y, 3), (\phi_x \wedge \neg\phi_y, 2y_0), \\
& \qquad (\neg\phi_x \wedge \phi_y, 3x_0), (\neg\phi_x \wedge \neg\phi_y \wedge x_0 = 5, 5y_0)\} \ldots\}
\end{aligned}$$

This sort of simplification allows MultiSE to make progress and get around the limitations of the underlying SMT solver. When such an approximation happens, we set a flag `incomplete` to true indicating that MultiSE cannot guarantee full coverage of the code. This approach has the same end result as the simplification approach proposed in DART [20]. In DART, there was no need to use an SMT solver to find a concretization because DART could read the concrete value of v from the concrete execution.

3.4 Soundness of MultiSE

There are two correctness results that hold for MultiSE, which we summarize informally:

- Soundness and completeness w.r.t. DSE: The final symbolic state of MultiSE encodes exactly the same set of behaviors as the final symbolic state of DSE.
- Soundess w.r.t. concrete executions: Any program behavior encoded in the final symbolic state of MultiSE corresponds to a concrete program behavior, and

The first correctness result follows from the fact that the effect of DSE can be obtained by changing the implementation of $\uplus$ in MultiSE: in DSE we do not merge guarded expressions in $\uplus$, whereas in MultiSE we do. Value summaries obtained from the two implementations of $\uplus$ are logically equivalent. The second correctness result follows from the first correctness result and the fact that DSE is sound with respect to concrete executions. We note also that the only reason completeness does not hold w.r.t. concrete executions is due to the approximations discussed in Section 3.3. Those approximations drop concrete paths from the symbolic representation. However, the paths that are kept are still faithfully represented in the symbolic state.

4. IMPLEMENTATION

We have implemented a prototype framework for MultiSE execution for JavaScript using the Jalangi framework [43] and we made it publicly available at `https://github.com/SRA-SiliconValley/jalangi` in the branch `symfront` under Apache 2.0 open-source licence. We chose JavaScript for prototyping because we are actively developing a symbolic execution engine for JavaScript in our Jalangi dynamic analysis framework [43]. The simple interface of the Jalangi's symbolic execution engine enabled us to quickly prototype MultiSE for JavaScript. Furthermore, being a dynamically-typed language, JavaScript made it convenient for the instrumentated program to carry symbolic expressions in place of the concrete values, without having to worry about static typing errors. We use CVC3 [6] for constraint solving, to handle the theory of integer linear arithmetic and strings (with append, length, equality check, parseInt, and regular expression matching).

4.1 Using BDDs To Represent Guards

We use binary decision diagrams (BDDs) to compactly represent Boolean formulas over Boolean variables. If we need to check if a guard or a path constraint ϕ is satisfiable, we first check if its BDD representation is not *false* and then we replace each Boolean variable in the formula by its corresponding symbolic predicate and check the satisfiability of the resulting formula using an SMT solver. The ordering on the Boolean variables in a BDD is the same as the order in which they are created. We noticed that the use of BDDs helped us to efficiently maintain and manipulate the guards. For example, in our experiments (see Section 5), on an average we spend less than 10% of total execution time in BDD manipulation, whereas over 85% of total execution time is spent in SMT solving.

Every time a new guarded symbolic expression is added to the value summary for pc, we invoke a quick BDD satisfiability check followed by an SMT solver satisfiability check for the path constraint. This is important in order to avoid exploring unfeasible paths. For the value summaries of other variables, only a BDD satisfiability check is used, to reduce the overall cost of SMT solving. During both MultiSE and DSE execution we generate an input for each satisfiable SMT solver call made by the respective techniques at a conditional statement. We generate inputs only at conditional statements because one of the key goals of symbolic execution is to generate a set of inputs that give maximal branch coverage. Generating inputs that forces program execution along both branches of a conditional statement ensures that we maximize branch coverage for the particular conditional statement.

5. EVALUATION

We ran experiments with the prototype implementation of MultiSE to: (1) measure the effectiveness of sharing in the value summary representation in MultiSE, and the total cost of symbolic evaluation and how much of it is due to BDD or to SMT solver calls; (2) estimate the performance gains in MultiSE compared to conventional DSE; (3) estimate the increase in the number of precise state merges compared to conventional state merging, due to the MultiSE's ability to merge even states involving non-integer variables.

For the evaluation we ran MultiSE on test harnesses created for publicly available JavaScript libraries. We create a symbolic test harness for a library by calling sequentially the methods of the library (possibly with repetitions) with inputs marked as `readInput`. Even if the tested library is small, the execution trees can be quite large if the test harnesses contain multiple invocations. The need to construct a test harness is the limiting factor for performing more experiments.

The experiments were performed on a laptop with 2.3 GHz Intel Core i7 and 16 GB RAM, and we averaged the timing measurements over several runs.

The first set of columns in Table 3.4 (with header **"MultiSE"**) show the relative cost of the various aspects of MultiSE. The "Total time" column reports the total running time of MultiSE in seconds, and the columns "BDD time" and "Solver time" report the percentage of time spent

Table 1: Results: DSE vs MultiSE vs Conventional State Merging (CSM)

Test	LOC	MultiSE					DSE/ MultiSE ratio				Improvement over CSM		
		Total time (s)	BDD time (%)	Solver time (%)	Avg. value summ. size	Avg. value summ. sharing factor	Time ratio (× speedup)	# of operations ratio	Solver time ratio	Avg. solver call time ratio	CSM precise merges (%)	MultiSE precise merges (%)	MultiSE merges
Find Max	32	5.0	1.2	97.9	1.9	23.0	10.0	28.7	9.9	0.8	100	100	102
Kadane Subarray	38	6.5	1.0	98.4	2.4	3.2	2.7	6.9	2.6	0.9	100	100	78
Array Index	56	11.7	5.3	93.4	9.1	9.1	3.7	3.3	3.9	0.9	100	100	478
Calc Parser	66	35.5	8.9	90.2	20.4	9.8	1.6	2.8	1.6	1.0	100	100	898
Stack	81	0.6	6.2	89.0	2.4	7.7	26.2	9.1	29.2	1.2	58	100	106
Queue	85	0.3	0	93.1	1.0	5.4	6.7	4.2	7.2	1.1	100	100	106
Heap Sort	87	4.0	1.5	96.7	1.7	5.6	2.5	8.5	2.5	1.0	100	100	274
Quick Sort	93	15.1	4.6	94.6	3.6	7.1	2.6	3.7	2.7	3.0	100	100	332
PL/0 Parser	135	246.4	18.7	80.4	29.3	45.8	1.3	2.7	1.2	0.9	100	100	5936
Linked List	148	2.5	3.6	95.1	2.8	5.3	11.1	5.1	11.6	0.9	43	100	218
Priority Queue	190	0.9	3.2	92.3	1.2	31.5	87.7	47.5	94.5	1.3	90	100	100
Binary Search Tree	386	6.5	2.4	96.6	2.4	9.4	7.3	5.6	7.4	0.9	22	100	240
Symbolic Arithmetic	475	1.5	9.1	82.3	1.8	39.3	49.3	34.0	51.4	40.4	72	100	592
BDD	623	6.2	68.2	19.6	2.5	6.4	7.5	5.4	29.6	24.3	41	100	26724
Red Black	1061	37.1	11.3	88.0	3.5	43.6	6.5	8.8	7.1	0.7	42	100	1878

in BDD manipulation and in SMT solving running time, respectively. We observe that even though MultiSE involves numerous boolean predicate constructors, the overall time spent in the BDD library is negligible. The SMT solver time takes most of the time. Compared to the SMT time, the time actually spent in interpreting statements and constructing symbolic expressions is also very small.

The column "Avg. value summ. size" reports the cardinality of the value-summary set, averaged over all variables during the execution of MultiSE. We observe that in many cases the value summaries are small (between 1 and 30). The smaller the size of a value summary, the more efficient it is to perform an operation on the value summary. This is especially true for statements that involve multiple variables, such as binary operations and conditionals, where we need to process all combinations of the value summaries involved. The effectiveness of the value summary technique is shown in the "Avg. value summ. sharing factor". This column contains the ratio between the number of paths to a point in the program and the size of the value summary, averaged over all variables and all program points. Recall that the size of a value summary is given by the number of distinct symbolic values for a variable at a point in the program. Our experiments show that the distinct values are shared on average between 3 to 45 paths. This validates our premise that there is a significant opportunity for a representation based on sharing as a value summary, instead of performing the computations for each path independently, as in conventional DSE.

The second set of columns in Table 3.4 (with header **"DSE/MultiSE ratio"**) show how much value-summary based symbolic execution improves over conventional DSE. We point out that in our implementation of MultiSE we can obtain the conventional DSE behavior by turning off compacting of value summaries. Recall that if s is a value summary and (ϕ, v) and (ϕ', v') are any two distinct elements of s such that $v = v'$, then we can merge the two elements as $\{(\phi \vee \phi', v)\}$ to obtain the equivalent value summary $s \setminus \{(\phi, v), (\phi', v')\} \cup \{(\phi \vee \phi', v)\}$. If we do not merge guards, we get conventional symbolic execution (DSE). At the end of conventional symbolic execution pc maps to a value summary where for each feasible path we have a statement label guarded by the path constraint for the path. We use this methodology to compare MultiSE and DSE.

In the "Time ratio" column we show how much more time it takes to run DSE compared to MultiSE. We observe a significant speedup, between 1.3× and up to 87×. This speedup is due to two related factors. First, DSE performs a lot more operations than MultiSE because it processes statements following a join multiple times, as shown in the "# Operations ratio". This column shows how many more operations DSE has to perform compared to MultiSE. Note that for each statement processed by MultiSE we count as many operations as the size of the value summary at that statement. Second, DSE spends significantly more time in SMT solver calls, as shown in "Solver time ratio" (DSE/-MultiSE). Finally, the column "Avg. solver call time ratio" shows the ratio between the average duration of a call to the SMT solver in DSE vs. MultiSE. We present this number to show that even in the face of more complicated constraints in MultiSE the cost of an individual SMT call is not higher. The real problem is the higher number of SMT calls that DSE must make.

The right-hand side of Table 3.4 (with header **"Improvement over CSM"**) estimates the increased effectiveness of MultiSE compared to conventional state merging (CSM). As discussed before, CSM techniques based on auxiliary variables cannot merge two states where variables of types other than integer or string have to be merged, because this would require introducing an auxiliary variable with constraints that cannot be handled by most SMT solvers. Therefore, conventional state-merging techniques would avoid those merges by dropping paths. To estimate the increased precision in MultiSE we modified MultiSE to count the total number of merges, and also the merges with value summaries that include entries for variables of non-integer types. In our JavaScript experiments these include variables containing floating-point values, objects, or function closures. The column "# of merges" shows the total number of merges performed, and the column "CSM precise merge" shows the percentage of those merges that conventional state merging can perform precisely. We note that MultiSE never introduces auxiliary variables and can proceed along all paths even when dealing with variables of types not supported by the constraint solver, which is why we show the value 100% in the column "MultiSE precise merge".

6. RELATED WORK

Recently several techniques for state merging [19, 1, 46, 23, 32, 5, 49] have been proposed to tackle the path-explosion problem. Dynamic state merging [32] merges state opportunistically so that the resulting path constraints do not stress the underlying constraint solver. MergePoint [5] alternates between path-based exploration of DSE and state-merging based exploration of static symbolic execution. State merging is only performed for code that does not contain system calls, indirect jumps, or other statements that are difficult to reason about precisely. Both of these techniques introduce auxiliary symbolic values and cannot merge states when there are unstructured control-flow and operations that introduce outside-theory constraints over auxiliary symbolic values. Rossette [49] also does state merging and manages to avoid some of the auxiliary variables. Rosette's state merging happens at join points and is type-based where two data-structure values, such as two lists having same length, are merged recursively to further compact the merged state. Recursive merging of data-structure values only works for immutable data-structures and cannot be applied to get state compaction in mutable data-structures used in imperative languages. Sinha [46] uses ITE (*if-then-else*) expressions for merging symbolic expressions at join points. Generated expressions usually have nested ITE expressions and are simplified using rewriting rules. Value summaries are similar to ITE expressions; however, in MultiSE there is no need for simplification as we perform explicit symbolic execution with the values in an ITE expression. This makes sure that we never encounter any nested ITE expressions. SMART [19] performs compositional test generation by computing summaries of all program functions. The summary of a function is computed by exploring all paths of the function using DSE and then by merging the symbolic states of those paths via symbolic auxiliary variables. For real programs, SMART can generate function summaries that are outside the theories that can be handled by an SMT solver. In such situations, it simplifies the summaries at the cost of completeness. Demand-driven compositional symbolic execution [1] was subsequently proposed to incrementally construct partial summaries to avoid analyzing unnecessary paths in functions. SMASH [23] incorporates both symbolic execution summaries (must summaries) and static analysis summaries (may summaries). SMASH performs reachability analysis to reason about possible buggy program states and to prune out group of uninteresting execution paths. All the above three techniques inherit the same limitation: they introduce auxiliary symbolic values at function interfaces. Therefore, they can reason about a function precisely only if the function's behavior can be captured by the given decidable theories.

Another line of work [50, 27, 35, 28, 7, 34, 13] tries to mitigate the path-explosion problem by pruning out redundant or unnecessary executions. Most of these techniques are orthogonal to compositional reasoning and state merging. A subset of these techniques avoid redundant executions by checking whether the current symbolic program state has been visited before. JPF [50] first uses state matching to avoid redundant state exploration. Kuznetsov et al. [32] implemented the similar idea to analyze C programs. Boonstoppel et al. [7] uses read and write sets to relax state matching condition. McMillan [35] proposed the idea to store interpolants as a generalization of visited states and to check inclusion instead of exact state matching. Tracer [28, 27], a symbolic execution engine targeting C programs, proposes to use interpolants to mitigate the path explosion problem by subsuming paths that can be proved to be safe. Another subset of these techniques try to decompose the program execution space into a number of independent sub spaces. For example, Majumdar and Xu [34] and Chakrabarti and Godefroid [13] applied program slicing ideas to cluster the program execution space. Recently, Santelices et al. [40], Qi et al. [37], Godefroid et al. [21], and Yang et al. [53] suggested incremental symbolic execution techniques to reduce the cost of regression testing of gradually evolving programs.

Function summaries [45, 38] have been used extensively in static program analysis. Graph-reachability based analysis [38], constraint-based analysis [52, 26], pointer analysis [51, 14], alias analysis [18], shape analysis [31], separation logic [12, 24], and abstract interpretation [39, 8] have incorporated function summaries for scalability. More recently, Gulwani et al. [25] and Yorsh et al. [54] investigated a general framework for summary-based static analysis. In general, function summaries in static analysis use interface symbolic values at function boundaries. Summaries are instantiated by replacing interface variables with real variables at call sites. Therefore, static analysis faces the same problem: a function behavior can be captured precisely only if it can be described in underlying decidable theories. However, this is not a serious limitation for static analysis because summaries can always be over approximated. Dynamic symbolic execution cannot over approximate a summary since over-approximation leads to loss of soundness.

Saturn [52] is one of the most closely related static analysis techniques. For intra-procedural analysis, Saturn and MultiSE have a number of commonalities: both perform symbolic execution, use guarded values to track values in a path sensitive manner, and maintain a path constraint. However, Saturn introduces fresh symbolic variables at function interfaces and updates guards of values at join points. MultiSE, in contrast, never introduces auxiliary symbolic values and performs join at every assignment to maintain a consolidated state throughout the execution. Guarded value flow analysis [15] is another closely related work. It performs value flow analysis using both path constraints and guards on values. Value can flow from a source to a sink if the conjunction of the guard on a value at the sink and the path constraint at the sink is satisfiable. However, the technique computes path constraint and guards on demand, while MultiSE performs symbolic execution to obtain guarded values at every execution point.

In general, one of the biggest advantages of static analysis techniques using summaries is that they can over-approximate a summary if it falls outside the scope of decidable theories; dynamic symbolic execution cannot over-approximate because it needs to know the exact path constraint for test generation.

7. ACKNOWLEDGEMENTS

The first and last author did part of this work while visiting Samsung Research America, San Jose. The work presented in this paper is supported in part by NSF grants CCF-1017810, CCF-0747390, CCF-1018729, CCF-1423645, CCF-1409872, and CCF-1018730, and gifts from Samsung and Mozilla.

8. REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, pages 367–381, 2008.

[2] S. Anand and M. J. Harrold. Heap cloning: Enabling dynamic symbolic execution of java programs. In *ASE*, pages 33–42, 2011.

[3] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *TACAS'07*, 2007.

[4] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA'08*, 2008.

[5] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.

[6] C. Barrett and C. Tinelli. CVC3. In 19th *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 298–302, 2007.

[7] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS'08*, 2008.

[8] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, pages 578–589, 2011.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[10] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE'08*, Sept. 2008.

[11] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, Dec 2008.

[12] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.

[13] A. Chakrabarti and P. Godefroid. Software partitioning for effective automated unit testing. In *EMSOFT*, pages 262–271, 2006.

[14] B.-C. Cheng and W. mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, pages 57–69, 2000.

[15] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI*, pages 480–491, 2007.

[16] V. Chipounov, V. Kuznetsov, and G. Candea. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2, 2012.

[17] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, pages 488–491, 1976.

[18] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577, 2011.

[19] P. Godefroid. Compositional dynamic test generation. In *POPL'07*, Jan. 2007.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI'05*, June 2005.

[21] P. Godefroid, S. K. Lahiri, and C. Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, pages 112–128, 2011.

[22] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS'08*, Feb. 2008.

[23] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL'10*, 2010.

[24] B. S. Gulavani, S. Chakraborty, G. Ramalingam, and A. V. Nori. Bottom-up shape analysis using lisf. *ACM Trans. Program. Lang. Syst.*, 33(5):17, 2011.

[25] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP*, pages 253–267, 2007.

[26] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *PLDI*, pages 168–181, 2003.

[27] J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 48–58, New York, NY, USA, 2013. ACM.

[28] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. Tracer: A symbolic execution tool for verification. In *CAV*, pages 758–766, 2012.

[29] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *In NFM'09*, Apr. 2009.

[30] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[31] J. Kreiker, T. W. Reps, N. Rinetzky, M. Sagiv, R. Wilhelm, and E. Yahav. Interprocedural shape analysis for effectively cutpoint-free programs. In *Programming Logics*, pages 414–445, 2013.

[32] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *PLDI*, pages 193–204, 2012.

[33] G. Li, I. Ghosh, and S. P. Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. In *CAV*, pages 609–615, 2011.

[34] R. Majumdar and R.-G. Xu. Reducing test inputs using information partitions. In *CAV'09*, LNCS, pages 555–569, 2009.

[35] K. L. McMillan. Lazy annotation for program testing and verification. In *CAV*, pages 104–118, 2010.

[36] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA'08*, July 2008.

[37] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/FSE*, pages 33–42, 2009.

[38] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[39] X. Rival and B.-Y. E. Chang. Calling context abstraction with shapes. In *POPL*, pages 173–186,

2011.

[40] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *ASE*, pages 218–227, 2008.

[41] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528. IEEE, 2010.

[42] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *CAV'06*, 2006.

[43] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *ESEC/FSE'13*, August 2013. To appear.

[44] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE'05*, Sep 2005.

[45] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[46] N. Sinha. Symbolic program analysis using term rewriting and generalization. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, page 19. IEEE Press, 2008.

[47] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS'08*, Dec. 2008.

[48] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *TAP'08*, Apr 2008.

[49] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541, New York, NY, USA, 2014. ACM.

[50] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *ISSTA'06*, July 2006.

[51] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

[52] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, pages 351–363, 2005.

[53] G. Yang, S. Khurshid, and C. S. Pasareanu. Memoise: a tool for memoized symbolic execution. In *ICSE*, pages 1343–1346, 2013.

[54] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *POPL*, pages 221–234, 2008.