Leveraging Abstract Interpretation for Efficient Dynamic Symbolic Execution

Eman Alatawi, Harald Søndergaard, and Tim Miller
School of Computing and Information Systems
The University of Melbourne, Vic. 3010, Australia
Email: e.alatawi@student.unimelb.edu.au, {harald, tmiller}@unimelb.edu.au

***Abstract*—Dynamic Symbolic Execution (DSE) is a technique to automatically generate test inputs by executing the program simultaneously with concrete and symbolic values. A key challenge in DSE is scalability, as executing all feasible program paths is not possible, owing to the possibly exponential or infinite number of program paths. Loops, in particular those where the number of iterations depends on an input of the program, are a source of path explosion. They cause problems because DSE maintains symbolic values that capture only the data dependencies on symbolic inputs. This ignores control dependencies, including loop dependencies that depend indirectly on the inputs. We propose a method to increase the coverage achieved by DSE in the presence of input-data dependent loops and loop dependent branches. We combine DSE with abstract interpretation to find indirect control dependencies, including loop and branch indirect dependencies. Preliminary results show that this results in better coverage, within considerably less time compared to standard DSE.**

I. Introduction

Testing remains the most commonly used method to ensure software quality. However, direct construction of test inputs for a given program is a complicated task. Dynamic Symbolic execution (DSE) [9], or concolic testing [17], is a well-known dynamic analysis technique for more systematic test case generation. DSE systematically explores a program, keeping track of how the inputs forced execution to take the path it took, so that it can find alternative inputs that will make execution take other paths. In this task, a DSE tool is assisted by a suitable constraint solver. The aim is usually to achieve high *coverage* for some chosen definition of coverage. While high coverage in itself provides no guarantee of absence of bugs, it is still considered a desirable goal in testing.

The problem. Scalability is a significant challenge for DSE in practice. Even for a loop-free program, the number of execution paths may be exponential in the program size. Loops, and in particular input dependent loops (those where the number of iterations depends on an input of the program), are a source of program path explosion [16], [10], [3].

An obstacle to high coverage is that DSE maintains values of the symbolic variables during execution, but does not track how these are related. In effect, *data dependencies* are tracked, in particular dependencies on symbolic inputs, but *control dependencies* are not. This means that, if a loop condition is not directly dependent on inputs, large parts of the code may remain uncovered [16]. Consider this program:

```
x = 0;
y = input;
while (y > 0) {
    x = x + 1;
    y = y - 2;
}
if (x > 1117)
    action A;
else
    action B;
```

In its first round, a DSE tool might explore the branch that leads to `action B`. The information available just before the 'if' statement does not suggest which inputs, if any, would execute `action A`. This is because x, while dependent on y because y determines the number of iterations of the loop, is never made to depend on other variables via assignments; its dependence on y is *indirect*.

Approach. We wish to improve the coverage achieved by DSE in the presence of input data dependent loops and loop dependent branches. To this end we precede DSE with static analysis to capture the indirect control dependencies, including loop and branch indirect dependencies on the inputs of the program. We exploit *abstract interpretation* based on well-known relational numeric abstract domains [6].

At first sight it would seem that classical abstract interpretation has little to offer. For a given program point p, abstract interpretation provides an over-approximation φ of the set of runtime states that may occur at p. Every state actually met at p during a (concrete or symbolic) execution must satisfy φ by definition; hence it may appear that φ offers no real information. However, φ may expose non-trivial relations among sets of variables, and these relations can be used to strengthen generated path constraints. Consider again the program above. A DSE tool would be greatly helped if we were to express invariants directly, using symbolic names for input values, such as '`y0`' for `y`'s initial value:

```
x = 0;
y = input;
y0 = y;
while (y > 0) {
    x = x + 1;
    y = y - 2;
}
/* -2 < y, y <= 0, 2*x = y0 - y */
if (x > 1117)
    action A;
else
    action B;
```

The assertion makes it clear that, to reach action A, we need to satisfy $-2 < y \wedge y \leq 0 \wedge y_0 > 2234 - y$. A constraint solver can now find a solution such as $y_0 = 2235$. Automated generation of powerful invariants, such as the one used above, is not beyond the capacity of static analysis tools.

Our focus is on improving DSE by tackling two main problems, namely: 1) path explosion resulting from input data dependent loops, where the number of iterations depends—directly or indirectly—on unbounded input; and 2) low coverage of a loop dependent branch whose condition depends indirectly on the number of iterations a previous input dependent loop has executed.

In our approach, we aim to determine the number of symbolic iterations of input dependent loops to reach program statements. This is achieved using abstract interpretation to calculate the number of required iterations to cover a given loop, and to generate invariants that relate that number of iterations to the program inputs. In addition, we support DSE by adding some relational information to cover branches that depend on an input dependent loop, such as the branch condition given in the example above, `if (x > 1117)`. To cover this branch, the value of x has to be greater than 1117, and in fact, the value of x has been changed inside the loop whose number of iterations depends on the program input y. This loop has to be executed x times to enable DSE to reach that branch. Such an indirect relation cannot be captured by classical DSE. Thus, we use abstract interpretation in this case to 1) calculate in advance how many iterations are needed to enter that subsequent branch; and 2) to generate invariants that relate program inputs to other program variables included in the conditional statement of the branch. These invariants are expressed in terms of symbolic values of program inputs. As a result, DSE can more readily synthesize input that will steer execution toward a targeted branch. For the analysis we use Cousot and Halbwachs's polyhedral domain [6], as implemented in the Apron library [11].

We assume the reader is familiar with the basics of abstract interpretation [4], [5], although this knowledge is not required to understand the paper. A key component is the *abstract domain* which is a computer-representable class of invariants (or sets of runtime states). The polyhedral domain is among the more expressive domains. While it is also expensive and fails to scale to the analysis of large programs, it is easily up to the task of analysing small-to-medium-sized programs.

II. Background

Dynamic Symbolic Execution [9], [17] systematically executes a program simultaneously with concrete and symbolic values. During execution DSE collects symbolic constraints on the program inputs along the executed path. This yields a conjunction called a *path condition*. Negating some conjunct corresponds to an alternative path which should be explored. To check whether this path is feasible, the perturbed path condition is sent to a constraint solver to check its satisfiability. If it is satisfiable, concrete test inputs are generated that will force execution to follow this new path. If it is unsatisfiable, the corresponding path is deemed infeasible[1], and search is pruned. A path condition can be simplified during DSE by using concrete values instead of symbolic values in many cases that the symbolic reasoning is impossible or the underlying constraint solver cannot handle the generated path condition.

The number of feasible paths that can be executed by DSE is significantly large as it could be infinite or exponential in the program size [3]. This *path explosion* problem [10], [19] leads to poor coverage in limited time.

Program loops, recursion, and sequences of branches are three main causes for the problem of path explosion [19]. Loops in particular pose a critical challenge for DSE that may affect its coverage and fault detection ability, owing to the exponential or infinite growth of the number of the explored paths especially in the presence of loops with an unbounded number of iterations [21], [19]. As a result, dealing with code containing loops is a key limitation of DSE. The problem is that even a single loop can generate a huge number of different execution paths, corresponding to different numbers of loop iterations and taking various paths through the loop.

Available approaches for solving unbounded loop problems include [21]: (1) bounding loop iterations to reduce the whole search space to be finite by bounding loop iteration with the loss of completeness; (2) using search-guiding heuristics to prevent DSE from being stuck in loops by guiding DSE toward exploring specific paths; (3) summarizing loops into a set of formulas that can be solved by using constraint solvers [16], [10], [12]; and (4) using abstraction to model loop iterations states, handling infinite loops with symbolic execution.

III. Approach and motivating examples

To illustrate our approach and the motivation for combining abstract interpretation and DSE, we introduce two examples to explain the details of our proposed method.

A. Input data dependent loops: Indirect dependency

Input dependent loops where the number of iterations depends on unbounded input lead to large numbers of possible execution paths. DSE tools typically impose a bound on the number of iterations of input-dependent loops, so as to terminate in a reasonable amount of time. However, arbitrarily bounding the number of loop iterations might leave important execution paths unexplored.

We aim to solve the following problem: Given a piece of code containing loops whose number of iterations to reach particular statements is *indirectly* dependent on program unbound inputs, the goal is to know in advance the minimum number of iterations that are needed to cover that loop and each branch presented within the loop, and capture the relation between program inputs and that number.

Consider the example in Listing 1 which is inspired by the examples given in [19], [10]. The number of iterations of the while loop indirectly depends on the input value x_0 stored in

[1]However, the lack of satisfiability can be due to other limitations, such as the generation of constraints that are too difficult for the constraint solver to solve.

```
void inputDependentLoop(int x) {
    int i = 3, p = x;
    while (true) {
        if (p <= 0)
            break;
        i = i+4;
        p = p-1;
        if (i >= 80)
            abort (); // target
    }
}
```

Listing 1. Input dependent loop

```
            :
    while (true) {
    /* x = counter + p, counter >= 0,
       counter <= 19, counter = (i-3)/4 */
            :
        if (p <= 0)
        /* x = counter + p - 1, counter >= 1,
           counter <= 20 */
            :
        if (i >= 80)
        /* x = counter + p, counter = (i-3)/4,
           counter = 20 */
            abort ();

    } /* x = counter + p, counter >= 0,
         counter <= 19, counter = (i-3)/4 */
```

Listing 2. Invariants generated for Listing 1

variable x at the beginning of the program execution. Suppose we start testing this program with $x_0 = 3$. The loop condition triggers the generation of the following path constraints:

$$(0 < x_0) \wedge (0 < x_0 - 1) \wedge (0 < x_0 - 2) \wedge (x_0 - 3 \leq 0).$$

Obviously x_0 appears in the path condition, but p, which appears in the loop exit branch, does not, because DSE tracks only direct data dependencies on the program inputs. Then it negates each constraint in the path condition one by one to generate new tests to exercise new program paths. This can be repeated, in principle forever if x_0 can be any integer.

Limited knowledge of the relations between program input and other variables in a program prevents DSE from achieving high coverage. This is manifest when DSE reaches a control point (branch or loop) that depends on how many times some previous input dependent loop was executed. For example, our target in this example is to cover the `abort()` statement, which is guarded by $i \geq 80$. In fact, the value of i depends on how many times the loop is iterated during the program execution. The DSE tool Pex [18], which we use as a representative for DSE tools in this paper, could not cover the branch for $(i \geq 80)$ when we set its exploration bounds (maximum runs, and maximum conditions) to 100. Pex generated the test inputs 1 and 3 which could not cover the target.

Abstract interpretation can help in maintaining the indirect data dependencies of program variables on inputs. The key insight here is that it is possible through a combination of abstract interpretation and DSE to obtain higher coverage.

Invariant generation: First, we add a new local variable called $counter$ that represents the number of loop iterations. It is incremented each time the loop body is entered. Its purpose is to explicitly express the relation between program variables and the number of times the loop body is executed. Second, we run forward abstract interpretation on the program to generate loop invariants using the *polyhedral abstract domain*. This produces invariants at each program point[2]. The Polyhedra domain is commonly used in static analysis to prove safety properties in programs like the absence of buffer overflow, division by zero, etc. It is a fully relational numerical domain, able to express arbitrary linear inequalities (and equalities) among program variables. It is more expressive than weakly relational domains such as Octagons [15], Pentagons [13] or Zones [14], and it comes with a correspondingly higher cost.

Listing 2 shows the invariants generated for Listing 1, by polyhedral abstract interpretation, as implemented in the *Interproc analyzer*. We are interested in discovering loop invariants and any invariants generated at each branch inside or after the loop. Note that abstract interpretation is able to generate a strong loop invariant, relating the program input x, the loop counter, and the local variable p, whose initial value depends on x, and appears in loop exit branch. In addition, polyhedral abstract interpretation generated invariants at each branch location specifying the number of loop iterations required to reach that point of the program expressed using the loop counter. Augmenting the path condition with these new invariants guides DSE successfully to find a test input that steers execution toward that targeted location. In other words, knowing the loop invariant $x = counter + p$, and the calculated invariants at the targeted location $counter = 20$, the DSE tool can produce the input $x = 20$ using the constraint solver. The invariant $x = counter + p - 1 \wedge counter \geq 1 \wedge counter \leq 20$ guides DSE to cover the input dependent loop with a minimal number of iterations. Thus, to terminate the loop, and for the break statement to be reached, the counter has to be at least 1, and in this case DSE can generate the test input $x = 0$. Invariants on the program counter can be exploited to identify the loop bounds, and use that bound to ensure both coverage, and execution termination with a minimum number of iterations. Considering the values assigned to the variable counter, we can set the exploration bound to 20. When we do that, Pex achieves full coverage in less test generation time.

Program annotation: We use program annotations to insert the necessary invariants at loop entry, after the loop, loop internal branches, and branches after the loop. These annotations can be added automatically or manually. We add these invariants as *assumptions* in the code.

Test input generation: We use the annotated program to generate test inputs using DSE. Pex needs bounds to ensure terminating the loop execution, by setting the bound. We arbitrarily set the bounds to 2, 4, 8, and 10. Within 10 iterations, Pex alone was able to generate only three tests,

[2]In fact, for integer variable $counter$, Interproc has $counter \geq 19.25$, $counter \leq 20$ after the second 'if', which we translate to $counter = 20$.

```
void inputDependentLoop(int n, int m) {
    int i = 0, k = 0;
    int j;
    int limit = 1000;

    while (i < n) {
        j = 0;
        while (j < m){
            k = k + 5;
            j = j + 1;
        }
        i = i + 1;
    }

    if (k >= limit)
        abort();      // target
}
```

Listing 3. Nested loops

```
   :
while (i < n) {
    /* k>=5*j, n>i, i>=0, j>=0 */
    while j < m {
         /* k>=5*j, n>i, m>j, j>=0, i>=0 */
        :

    } /* k>=5*j, n>i, j>=m, i>=0, j>=0 */

     :
} /* k>=5*j, i>=n, i>=0, j>=0 */

if (k >= limit)
    /* k>= 5*j, k>=1000, j>=m, j>=0, i>=n, i>=0 */
    abort();
  :
```

Listing 4. Generated invariants for Listing 3

$x = 0, x = 1, x = 7$ that could not cover the target. A new test $x = 24$, is generated by Pex when the invariants are added, which leads to coverage of the target.

B. Input dependent loops: Direct dependency

We aim to solve the following problem: Given a piece of code containing a loop whose number of iterations is *directly* dependent on program inputs, and a targeted loop dependent branch anywhere in the code after the loop whose condition depends on the number of times that the loops have iterated, generate a test of that target. The goal is to know in advance how many iterations are needed to enter that branch, and to capture the relation between program inputs and that variables presented in the branch condition.

Consider the program example shown in Listing 3, where two nested loops has input-data dependent termination condition; that is, the number of the iterations of the nested two loops directly depends on the input values m_0 and n_0 stored in variables m and n at the beginning of the program execution. The goal here is to generate inputs to cover the branch `if (k >= limit)` to steer the program execution toward the abort statement. DSE-based reasoning will almost certainly not be able to generate such inputs, as this requires relating the value of variable k to the values of the program inputs m_0 and n_0. Pex, for example, was not able to reach that branch with different bounds including 10, 100, and 1000.

The value of k is modified inside the inner loop. So the question here is: *How many times do we need to iterate these two nested loops to reach the abort statement, and what is the relation between the required number of iterations, and the program inputs that could help DSE hit that statement?*

To answer this we want to find the linear relation between loop iteration counts, program local variables, and inputs, that is, relate program inputs to both k and loop counters.

To identify the linear relation between k and the program inputs n and m, we introduce a new counter to the program to record the required number of iterations needed to reach any program control point after a loop using abstract interpretation theory. We are interested in computing the minimum number of iterations required to cover the loop, and to cover any subsequent loop dependent control points.

Abstract interpretation computes statically the number of required counts by relating it to other program variables. We use forward abstract interpretation to compute sound invariants using the polyhedral abstract domain. Listing 4 shows the invariants generated by the *Interproc analyzer*. Note that we cannot hope to find the non-linear invariant $k = mn$. The invariant $k \geq 5j$ is valid for both nested loops, and knowing the invariants at the target location, that is,

$$k \geq 5j \wedge k \geq 1000 \wedge j \geq m \wedge j \geq 0 \wedge i \geq n \wedge i \geq 0,$$

gives DSE a chance to find appropriate test inputs such as $(n = 0, m = 200)$. Indeed, once we annotate the program with such invariants, Pex can cover the targeted location and achieve 100% coverage with only 3 tests, compared to 75% coverage with 5 tests for the original subject without the invariants.

IV. Preliminary evaluation

We performed a preliminary evaluation of our approach using polyhedral analysis [6] as implemented in the *Apron library* and the state-of-the-art invariant generation tool *Interproc* [11]. We used the Interproc analyser to infer invariants for each program point automatically. These invariants are intended to relate the program inputs to the established loop counter and other related variables in the program under test. We used 6 small examples taken directly from, or inspired by, the literature [8], [19], [10], [7], each subject contains an input dependent loop, and at least one loop dependent branch and/or an assertion. Subject 2 has two nested loops, subject 6 has two sequential loops.

First we ran *Interproc* on the subjects, and then the code was annotated with the generated invariants at each marked target in the code. Second, we ran Pex on both the original and the annotated subject. We set the maximum exploration bound (MaximumRuns) in Pex to 100 to limit the exploration, so that the tool does not get stuck in loops that might have infinitely many execution paths. We measured the size of the generated test suite, the line coverage of the generated test inputs, the time taken by Pex to generate the test inputs, and the number of runs attempted by Pex during the exploration.

TABLE I
PRELIMINARY EVALUATION RESULTS

	Interproc		Pex			Pex+AbstractInterpretation			
Subj	**InvG Time**	**#Tests**	**Coverage**	**TestG Time**	**#Runs**	**#Tests**	**Coverage**	**TestG Time**	**#Runs**
Subj1 (List. 1)	0.01	4	100%	6.25	14	4	100%	0.50	23
Subj2 (List. 3)	0.01	5	75%	2.40	100	3	100%	2.30	100
Subj3 [8]	0.01	1	75%	3.73	100	2	100%	1.80	100
Subj4 [10]	0.01	2	56%	0.75	14	3	100%	0.76	15
Subj5 [7]	0.01	5	82%	1.22	25	3	100%	1.00	28
Subj6 [1]	0.01	0	0%	0.64	100	2	100%	0.43	4
Avg	0.01	2.8	65%	2.50	58	2.8	100%	1.10	45

Table I presents the results of our experimental evaluation. The column labeled "InvG Time" indicates how long (in seconds) Interproc took on each subject, to generate the invariants. The columns grouped under the section Pex, namely: "# tests","Coverage", "TestG Time", and "#Runs" indicate the size of the generated test suites, coverage achieved by a Pex-generated test suite, test generation time, and number of runs that are attempted by Pex during DSE exploration, respectively. Similarly, the third section of the table list the results for Pex supported by abstract interpretation.

These preliminary results show that Pex with abstract interpretation was able to generate test inputs that achieved better coverage (average of 35% increase in the coverage). Moreover, with the help of generated invariants, Pex was able to handle input dependent loops efficiently: the testing time was clearly decreased, compared with the time taken by Pex alone. For example, when bounds were set to 100, Pex was able to reach to 100% coverage only for subject1 (Listing 1), and it took comparatively longer time (6.25 seconds, compared to 0.5 seconds for Pex with abstract interpretation). For the remaining subjects, Pex alone was not able to achieve full coverage (it was unable to cover loop dependent branches, as it consumed the available number of runs, iterating the input dependent loop). Conversely, Pex with abstract interpretation achieved full coverage through the guidance of the added invariants. These helped Pex to iterate the input dependent loop enough to be able to terminate the loop execution, and at the same time cover any loop dependent branches.

Adding program invariants guides Pex toward more interesting inputs that are able to satisfy the inferred relation between program inputs and other variables in the program under test. For Listing 3, again, static analysis allowed Pex to generate fewer tests, compared to the case without invariants, but with better coverage. In addition, test generation time was slightly better in this case (2.30 seconds compared to 2.40 seconds). In our experiments, adding loop invariants did not increase the exploration time, but it did lead to better tests that achieve better coverage even if Pex in both cases reached to the limit of the assigned exploration bound. However, if we reduced the limit of runs to 50, Pex with abstract interpretation achieved 100% coverage in only 1.30 seconds.

Relating program inputs to other variables in the program is significant for DSE tools to achieve good coverage, especially in the presence of assertions that involve only non-input variables. Subject 6 provides an example of this—the program has an input k and contains an assertion that includes only local variables. Here Pex alone was not able to generate any test within the exploration bound. On the other hand, with only 4 runs, 0.43 seconds, and 2 generated test inputs, Pex + abstract interpretation was able to achieve 100% coverage.

These results suggest that, compared to DSE alone, our technique produces slightly smaller test suites with better coverage, while reducing test generation time, for code containing input dependent loops and loop dependent branches.

V. RELATED WORK

There are few approaches that are focused on how to deal with input-dependent loops in DSE. Saxena et al. [16] propose an approach called Loop-Extended Symbolic Execution to capture the relation between program inputs and loops in DSE. They introduce a new symbolic variable called *trip count* for each program loop. This count represents the number of times a loop body was executed, at any given time. Saxena et al. obtain the relationship between these variables and other variables in the program by running a separate static analysis. They relate the trip counts to the program input format by introducing auxiliary variables to capture how loop-dependent variables are related to the lengths and counts of elements in the program input based on an input grammar. In contrast, our approach infers loop invariants and captures the relations between program inputs and all program variables purely statically before DSE. Symbolic execution is then guided by the inferred relations. Our method distinguishes between direct and indirect dependency of loops on the input data, and introduces a loop counter as a new local variable to explicitly capture the relation between the program input and loop iterations.

The introduction of loop counter variables is also found in the context of abstract interpretation, in the search for numerical abstract domains that trade away some expressiveness to avoid the heavy cost of polyhedral analysis. The Gauge domain [20] offers a limited relational analysis, in which relations among program variables cannot be expressed, but

relations *are* maintained between each program variable and specially introduced loop counter variables.

Godefroid and Luchaup [10] use loop summaries to deal with certain types of unbounded loops that include induction variables, whose values are modified by a constant value or constant times for each loop iteration. Their goal is to determine the number of iterations of input-dependent loops by automatically guessing an input constraint using simple loop-guard pattern matching rules. This solution is performed dynamically without requiring any static analysis or input-format specifications (unlike [16]). Loops are summarized by loop pre-conditions and post-conditions that are derived from inferred partial loop invariants relating the program inputs to the induction variables. Our approach is simpler and does not require detection of induction variables or loop structure. We simply infer program invariants at each program point, relating the value of input variables, and all other local variables in the program within any loop (that is, not only induction variables). In addition, we determine the required number of iterations required to reach any branch within or after the loop statically.

Čadek *et al.* [2] propose an algorithm for computing upper bounds for execution counts of individual instructions of an analyzed program during any program run. The algorithm is based on symbolic execution and the concept of path counters. The upper bounds parameterized by input values of the analyzed program path counters are more general: executions of a single program path in a loop can be counted by several path counters relative to some other program paths even in other loops.

VI. Conclusion and future work

We have proposed a way of improving the coverage achieved by DSE in the presence of input data dependent loops and loop dependent branches. The idea is to precede DSE with well-known analyses from abstract interpretation based on relational numeric abstract domains to capture the indirect control dependencies on the inputs of the program. Preliminary results suggest that, compared to dynamic symbolic execution alone, the outcome is better coverage, achieved in considerably less time.

So far we have communicated static analysis results to a DSE tool by manual transfer. Hence, while the results are very promising, the subjects on which we have tested the idea have been small, single functions, albeit non-trivial from a reasoning perspective. We plan to integrate the two components properly, so that we can work on scaling up the method. For very large programs, the domain of convex polyhedra is likely to prove too expensive. However, a large number of relational abstract domains, of varying expressiveness, have been suggested and implemented, and often made available as open source code. One should be mindful, however, that while DSE tools usually deal with programming languages that assume finite-width integers, most relational abstract domains are based on proper integers, and this discrepancy has the potential to cause unsoundness of analysis.

Acknowledgments

The first author gratefully acknowledges support from Taibah University, Saudi Arabia, through a PhD scholarship. The work was also supported by the Australian Research Council through Linkage Grant LP140100437.

References

[1] N. Bjørner and A. Gurfinkel. Property directed polyhedral abstraction. In D. D'Souza *et al*., editor, *Verification, Model Checking, and Abstract Interpretation (VMCAI'15)*, volume 8931 of *Lecture Notes in Computer Science*, pages 263–281. Springer, 2015.

[2] P. Čadek, J. Strejček, and M. Trtík. Tighter loop bound analysis. In *14th Int. Symp. Automated Technology for Verification and Analysis (ATVA'16)*, pages 512–527. Springer, 2016.

[3] M. Christakis, P. Müller, and V. Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proc. 38th Int. Conf. Software Engineering (ICSE'16)*, pages 144–155. ACM, 2016.

[4] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, 1996.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Proc. Fifth ACM Symp. Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.

[7] CVE—common vulnerabilities and exposures, May 2017. http://cve.mitre.org/.

[8] P. Dinges and G. Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proc. 29th ACM/IEEE Int. Conf. Automated Software Engineering (ASE'14)*, pages 31–36. ACM, 2014.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005.

[10] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proc. 2011 Int. Symp. Software Testing and Analysis (ISSTA'11)*, pages 23–33. ACM, 2011.

[11] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[12] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger. Loop summarization using state and transition invariants. *Formal Methods in System Design*, 42(3):221–261, 2013.

[13] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *Proc. 2008 ACM Symp. Applied Computing*, pages 184–188. ACM, 2008.

[14] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, Ecole Polytechnique, 2004.

[15] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[16] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proc. 18th Int. Symp. Software Testing and Analysis (ISSTA'09)*, pages 225–236. ACM, 2009.

[17] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. the 10th European Software Engineering Conf.*, pages 263–272. ACM, 2005.

[18] N. Tillmann and J. De Halleux. Pex—white box test generation for .NET. In B. Beckert and R. Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[19] M. Trtík. *Symbolic Execution and Program Loops*. PhD thesis, Faculty of Informatics, Masaryk University, 2013.

[20] A. J. Venet. The Gauge domain: Scalable analysis of linear inequality invariants. In P. Madushan and S. A. Seshia, editors, *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2012.

[21] X. Xiao, S. Li, T. Xie, and N. Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proc. 28th IEEE/ACM Int. Conf. Automated Software Engineering (ASE'13)*, pages 246–256. IEEE Comp. Soc., 2013.

View publication stats