Dynamic Symbolic Execution Engines
ECE653 - Final Project Report

Chong Hou Choi(chchoi@uwaterloo.ca)
Betty Gao (b36gao@uwaterloo.ca)

University of Waterloo, Waterloo, ON, Canada

Abstract. In this project, we implemented the EXE-style and DART-style dynamic symbolic execution engines and compared their advantages and disadvantages. Based on the relative strengths and weaknesses of the engines, we concluded by presenting suitable use cases for each DSE execution style.

Keywords: Symbolic Execution · Concolic Execution · Dynamic Symbolic Execution.

1 Introduction

1.1 Concrete Execution

Concrete execution refers to the traditional method of running computer programs by providing them with actual input values and observing their behavior. It involves the actual computation of program statements with real data. This approach is the basis of most testing and debugging practices commonly used in software development.

Here is the process of concrete exection to a program. The program reads real inputs, processes them according to the code, and produces outputs. The outputs can be observed and compared with the expected results. This method of execution is powerful for verifying the correct behavior of a program under different circumstances.

However, concrete execution has its own limitations. It is nearly impossible to exhaustively explore all execution paths of a program due to the vast number of possible inputs, especially for complex systems. As a result, it can only help us reveal the bugs. It does not guarantee the program is 100% bug-free.

1.2 Symbolic Execution

Symbolic execution is an analytical approach used in the field of computer science to reason about the behavior of programs by treating their inputs as symbolic variables rather than concrete values. This technique allows for a systematic exploration of a program's execution paths, enabling the discovery of bugs, security vulnerabilities, and logical errors that might not be apparent through traditional testing methodologies.

For symbolic executions, as it maintains the explored path with its symbolic expressions, a state in symbolic execution is denoted as a tuple:

$$< \textbf{path-condition, symbolic-environment} > \quad (1)$$

However, there are some drawbacks in symbolic execution engines:

1. It's a tough method to scale, especially for complex programs. The number of paths might increase exponenially for complex programs, making it more time-consuming and require more computing resources to solve the problem. It might cause path explosion[1] as the number of paths may easily become exponential in the number of branches.
2. Satisfiability is an NP-complete problem, especially for complex programs. It could take lots of time for the SMT solver to come up with a solution for a node.

Thus, dynamic symbolic execution engines can solve these problems with executing concrete execution and symbolic exeuction at the same time.

1.3 Dynamic Symbolic Execution

A disadvantage of pure symbolic execution is that it requires an SMT solver to determine the satisfiability of each path condition that arises during execution. Since satisfiability can be difficult to determine (and can even be NP-hard for some problems), a symbolic solver may encounter difficulties when executing programs with complex functions. To overcome this challenge, the technique of dynamic symbolic execution was developed.

For Dynamic Symbolic Executions, as it maintains the explored path with its symbolic expressions and its concrete expressions at the same time, a state in symbolic execution is denoted as a tuple:

$$< \textbf{path-condition, symbolic-environment, concrete-environment} > \quad (2)$$

During dynamic symbolic execution, each statement of the program is executed both concretely and symbolically. In path conditions that involve complex functions, the symbolic output of the function is assumed to be equal to the concrete output to simplify the satisfiability problem. This sacrifices some completeness of path coverage (i.e. solver might miss some feasible paths), but improves the practical applicability of the symbolic execution engine.

The two most popular techniques of dynamic symbolic execution are EXE and DART. The two techniques differ primarily in the search order in which the feasible paths are explored, as well as when the SMT solver is called in the execution process. In this report, we extended the wlang code package to include two new modules implementing each of the two techniques:

1. exe.py: Implements a EXE-style DSE engine
2. dart.py: Implements a DART-style DSE engine

While both DSE engines can be instantiated and used directly, we have also extended the run function of the symbolic engine module sym.py to include a concolic execution option. For any instance of the sym.SymExec class, users can now specify "EXE" or "DART" for concolic option to run the corresponding DSE engine instead of the pure symbolic engine.

The following sections of the report discuss and compare our implementations of the EXE- and DART- style DSE engines.

2 EXE-style Dynamic Symbolic Execution

2.1 EXE Engine Implementation

EXE is a kind of Dynamic Symbolic Execution engine based on Breadth-First Search(BFS). In this engine, both symbolic execution and concrete execution are executed in parallel. Whenever the SMT solver in symbolic execution couldn't solve particular paths, it will apply concretization to try if PC is satifiable.

The following is the pseudo-code(or the steps) of how an EXE Dynamic Symbolic Execution engine test a program:

1. Receive a seed input state from the user. The input state should be concolic, containing a concrete and a symbolic environment. In addition, the input state should have concrete assignments for any variables that are not defined in the program.
2. Execute the program in the concrete environment, following the path reacheable by the concrete variables. In parallel, execute the program symbolically to ensure that the symbolic assignments contains the concrete ones.
3. Whenever the program reaches a branching node, run the symbolic solver to check whether the path condition for the path not taken is satisfiable. If so, pick a concrete model for the path not taken by the concrete execution. Both the original state and the new concrete model are then returned to be processed by the remainder of the program
4. Subsequent statements in the program are executed on each of the concolic states returned by the preceding statement. The above steps are repeated to discover all the feasible paths of the program.

The file of the implementation of our EXE Dynamic Symbolic Execution engines is in exe.py. The implementation is based on wlang, copyright by Prof. Arie. The structure of the code is similar to the one in sym.py, which is the implementation of symbolic execution engine. We've implemented a class named DynamicSymState to store the state of a node, and a class named DynamicSymExec to implement the EXE Engine.

2.2 exe.DynamicSymState

It is a class based on the implementation of SymState in sym.py, which is used to save all the variables needed for a state of Dynamic Symbolic Execution.

It includes the symbolic environment, the concrete environment and the path condition as well.

Besides, it still has a SMT Solver inside to help solve the satifiability problem for the path condition. Whenever the function $pick_concrete$ is executed, the SMT Solver will come up with a concrete state consistent with the symbolic state if the path condition is satisfiable, else return None.

2.3 exe.DynamicSymExec

This class defines the EXE-style Abstract Syntax Tree (AST) visitor, which runs concrete execution and symbolic execution in parallel. For a AST node, it returns a list of DynamicSymStates, which are the reachable states of the program. For each AST node, the engine will update the corresponding symbolic environment, concrete environment and the path condition of the concolic state.

For if and while statements, it is able to fork states to visit different branches of the program. While visiting different paths, our engines will check which path the concrete state takes, and check whether if it is possible to take the other paths using symbolic execution and the SMT solver.

In the while statements, as a loop limit of 10 is required, we add a new variable in args to count the number of loops and update this variable every time we visit the while statement function. When it reaches the loop limit, it terminates.

In the havoc statements, variables are assigned to 0 in the concrete environment, and to a Z3 Fresh Integer in the symbolic environment.

When the z3 solver returns unknown, we apply concretization to the variables, which means that we assign a new concrete value to the symbolic environment to see whether the path condition is satisfiable(run the z3 solver again). If it's unsatisfiable, we just ignore this path and not discover any further. However, as we are not able to find out any input that could make the z3 solver return unknown, we've not been able to cover the related branch.

2.4 Testing the EXE Engine

The exe.py module was unit tested to 93% branch coverage.

To test concretization, we chose a WHILE sample program that involved expressions involving a lot of multiplication since they are known to be difficult for SMT solvers to solve. The solver in the EXE engine is configured to time out and return an UNKNOWN status after attempting to solve for more than 0.2s. It can be observed that by applying concretization constraints, the Z3 solver is able to determine satisfiability of path conditions more quickly.

For concretization in EXE, we were not able to find a WHILE sample program which produced path conditions that are satisfiable but takes longer than 0.2s for Z3 to solve even though we were able to test this in DART mode. This is possibly because the solver constraints in EXE mode are much simpler than solver constraints in DART mode.

2.5 Advantages and Disadvantages

There are several advantages for EXE Dynamic Execution Engines:

1. As it executes concrete environment and symbolic environment in parallel, and use a searching strategy based on Breadth First Search, it is possible for modern processors to fork multiple threads and allocate resources to compute different branches simultaneously. Thus, it should be more efficient.
2. Since new concolic states are only generated when the program forks at branch nodes rather than at the start, the EXE engine also runs more efficiently than DART by virtue of not needing to repeatedly execute edges that are already visited. By contrast, a DART engine would need to execute the program from the beginning for each path even though some of the edges in the path are already covered.
3. Using concretization at the right time can help reduce complexity of the program significantly. Therefore, it makes it possible for us to test complex program using EXE Dynamic Execution Engines.

One disadvantage of the EXE engine is that because concrete values are assigned mid execution where paths branch rather than at the start of the program, the engine can only return a list of terminating states and does not naturally generate a list of corresponding concrete inputs that result in each terminating state. This creates challenges in using EXE in conjunction with concrete testing techniques such as fuzzing. Besides, EXE engines are not complete, which means that it may not be able to cover all the feasible paths of the program.

3 DART-style Dynamic Symbolic Execution

3.1 DART Engine Implementation

Directed Automated Random Testing (DART) originated as a way to supplement concrete random testing techniques such as fuzzing, by using symbolic execution to direct test case generation.[2] Unlike the EXE engine, which predominately extends the pure symbolic engine with the additional utility of concretization, DART behaves more like an extension of the concrete execution engine with the additional utility of path condition solving.

The pseudocode for DART is as follows:

1. Receive a seed input from the user or a random generator.
2. Execute each statement of the program concretely in the same fashion as a normal concrete interpretor. In addition to the concrete state, the DART engine also tracks the symbolic path conditions that are visited as the program executes.
3. Once the execution of the program is completed, solve for a new input state that would explore a path that has not yet been covered (i.e. an input state that satisfies the negation of the disjunction of all the path conditions explored on prior test runs).

4. Continuously repeat the execution and input generation cycle until there are no more unvisited feasible paths (i.e. when the negation of the disjunction of past path conditions becomes unsatisfiable.)

Our implementation of the DART engine (located in dart.py) is structured as an extension of the concrete interpretor int.py. Likewise, the state object on which the DART engine operates is a subclass of the int.State class. This is intended to provide compatibility between the DART engine and the normal concrete interpretor, allowing the DART engine to be used by any program that expects a concrete Interpreter object. Likewise, the test inputs generated by the DART engine can be seemlessly processed by the concrete interpretor in int.py without further conversion.

The dart.py module in our wlang package consists of the following classes:

3.2 dart.InstrumentedState(int.State)

This class defines the concolic state object on which our DART engine operates. It inherits all the methods and members of the concrete int.State class, but also includes a symbolic environment and a path condition list. In addition, it extends the parent class with methods to update the path condition during concrete execution, as well as methods and members to flag whether the state is an error state.

The resulting class is a concolic state that is quite similar to the DynamicSymState object used by the EXE engine. The main difference is that the InstrumentedState class does not include a SMT solver and does not have the ability to check the satisfiability of its path conditions. This is because in DART style execution, the symbolic solving occures "offline" once the concrete execution terminates.

3.3 dart.InstrumentedInterpretor(int.Interpreter)

This class defines the DART-style Abstract Syntax Tree (AST) visitor, which concretely executes the AST that represents the program while symbolically tracking path condition. For expression-type AST nodes, InstrumentedInterpretor returns a tuple containing a concrete judgement as well as a symbolic formula representing the expression. For statement-type AST nodes, InstrumentedInterpretor updates the concrete environment, symbolic environment, and path condition of the concolic state, and then returns the new state.

For most AST nodes, the concrete execution in dart.InstrumentedInterpretor exactly mimics its parent class int.Interpreter. The exceptions are the three modelling statements: assert, assume, and havoc.

For assert and assume statements, a pure concrete interpretor would immediately raise an exception and terminate the program if the assert or assume condition is not satisfied. In DART execution however, we need the engine to continue executing in order to update the cumulative path condition and generate new test cases. Therefore, dart.InstrumentedInterpretor would print an error

when assert or assume is violated, but flag the concolic state as an error state instead of raising an exception. The DART visitor for statement list considers concrete execution to be terminated whenever an error state appears.

In addition to explicit asserts and assume statements, implied assertion failures such as dividing by 0 would also cause a pure concrete interpretor to terminate. In DART execution, to prevent the failure from terminating the entire DART engine, the interpretor catches the exceptions and sets the concrete judgement for the problematic expression to be Null. In addition, the state that generated the failure is marked as an error state to terminate the concolic run. Finally, the path condition for the state is updated via conjunction with equality constraints to denote this particular concrete assignment as "visited". This will later prompt the DART input generator to find new input that continues to explore the current symbolic path while avoiding this particular error state.

For havoc statements, in theory a concrete execution engine should assign some nondeterministic value to each of the variables included. In practice, since present-day computers are deterministic machines, implementations of concrete interpretors must pick a single value to assign to each variable. In int.Interpreter, each of the havoc variables are assigned a value of 0 for simplicity. In the DART visitor, each of the havoc variables are only assigned 0 if they are undefined in the preceding state. Havoc variables that already existed in the preceding state are simply assigned the same value as the preceding state by the DART visitor. This is necessary to allow the DART engine to cover different feasible paths when path conditions depend on variables that are havoc'ed part way through the program.

For most AST nodes, the DART generation of the symbolic formula as well as path condition is similar to that of the EXE visitor (exe.DynamicSymExec) as well as the pure symbolic visitor (sym.SymExec). For the DART while visitor, the path condition must be tracked somewhat differently since we cannot directly enforce a symbolic loop iteration limit in the AST visitor itself. To prevent the DART input generator from being inundated with infinite loops when it comes time to solve for the next inputs, the DART visitor tracks the number of loop iterations when concretely executing the while loop. For the first 10 or fewer iterations, DART updates the path condition with the loop condition so that the next test case would be generated to explore a different branch. For any iterations beyond the first 10, DART stops incorporating the loop condition into the path conditions of the concolic state. This effectively indicates to the input generator that all branches resulting from more than 10 iterations have already been tested and do not need to be considered during input generation. Thus, the input generator would only produce test cases that explore the first 10 iterations of each loop and then move on to generating test cases for coverage of other parts of the program.

3.4 dart.DARTInputGenerator

This class inplements the full Directed Automated Random Testing process by 1) formulating the initial "seed" input state, 2) running the DART interpretor

to process the AST representing the program, and 3) running the SMT solver to generate the next input state.

Initializing the DART engine Prior to the first run, the DART engine examines the AST to determine which variables in the program would need to be assigned a value in the test cases. This is done using an AST visitor similar to the undefined variable checker as implemented in undef.UndefVisitor. Ultimately, the DART engine ensures that each input state contains all variables that are either undefined in the program or only defined by havoc statements.

For the initial seed input, the DART engine accepts a concrete state object instantiating any child of the int.State class. If the input state is a InstrumentedState object, it would be used directly. Otherwise, it would be replaced with a InstrumentedState object whose concrete environment is an exact duplicate of the original input.

The input state provided to the DART engine may have full, empty, or partial assignments for each of the variables that must have an input assignment. For any such variable that does not already have an assignment in the seed input, DART assigns a value to it randomly.

In addition, the input state provided to the DART engine may have concrete or symbolic assignments for unused variables or variables that would be reassigned within the program. These variables are kept in the initial input state environment for the first run, but will not be included by the DART engine during subsequent input generations.

After completing the initial input assignments, the DART engine stores an image of the initial symbolic expression that represents each of the variables requiring assignments. After each run terminates, the DART engine would generate the next concrete input by solving for an assignment for each of the symbolic variables in the image. The symbolic environment of the next input state would also be assigned by duplicating this initial image. This is necessary to ensure that the same symbolic variable names are used across all runs, which in turns allow the SMT solver to find unexplored paths by taking negation of the disjunction of path conditions from each run.

Running the concolic interpretor Once the DART engine and initial seed input have been initialized, the DART engine calls the run function of the concolic interpretor (dart.InstrumentedInterpretor) to execute the program as described in the previous section. Upon termination of the concolic execution, the DART engine logs the return state from the interpretor along with the corresponding input state that was fed to the interpretor. The path condition from the return state is added to the cumulative path condition by disjunction to log the increase in path coverage. Now that the cumulative path condition has been updated, the DART engine is ready to generate the next input state by symbolically solving for the negation of the cumulative path.

One challenge we encountered when running the DART concolic interpretor was the fact that python concrete execution handles integer division differently

than the Z3 solver. Python integer divisions may return a float value, whereas in the Z3 environment integer division would round the fraction down to the nearest integer. This creates edge cases where the concolic interpretor ends up exploring a different path than the path which the Z3 solver expected the test cast to take. While this has no impact on the soundness or completeness of the DART engine (because the cumulative path condition logs the path that was actually explored), we did need to implement workarounds to prevent the Z3 solver from repeatedly generating the same input because the input satisfies the "unvisited" path in the Z3 environment. To prevent the DART engine from generating the same input state twice, equality constraints are also added to the cumulative path to indicate each input state assignment as "already seen". This directs the DART engine away from edge cases and forces it to pick a different input that covers the unexplored paths.

Generating the next test case The pick_concrete() method of DARTInputGenerator implements the process of solving for the next test case. To generate the next input state, the DART engine uses the Z3 SMT solver to determine satisfiability of the negation of the cumulative path condition. This solver constraint represents the combined path conditions of the unexplored paths.

If this constraint is satisfiable, then the Z3 solver will find a model that satisfies the constraint. The symbolic expression for each input variable awaiting assignment is then evaluated under this model to produce the concrete environment of the next input state. The DART engine will continue the interpretor run - input generation cycle so long as the solver constraint remains satisfiable.

If the solver constraint is unsatisfiable, then a Null state is returned. This would cause the DART engine to terminate and return the list of return states generated by all the test cases that the engine has generated. In addition, users can choose to extract just the return states, just the input states, or a zipped list of input-return state pairs.

If the solver constraint is unknown, the DART engine will incrementally concretize the input variables one by one and re-running the Z3 solver. This is done by replacing the symbolic assignment for the variable with its concrete assignment from the preceding input state. In addition, a concretization constraint is added to the Z3 solver in a new scope. If the solver returns UNSAT after concretizing any variable, the concretization is reversed prior to trying the next variable. Finally, if the solver still returns unknown after attempting to concretize all input variables. the DART engine treats the remaining unexplored path as infeasible. On the other hand, if the solver returns SAT after concretizing a variable, the DART engine generates the next input state from the satisfying model. All concretization constraints are then removed from the solver prior to proceeding with the next interpretor run.

3.5 Testing the DART Engine

The dart.py module was unit tested to 98% branch coverage. The covered statements and branches include all reacheable branches within the dart.py module.

To test concretization, we chose a WHILE sample program that involved expressions involving a lot of multiplication since they are known to be difficult for SMT solvers to solve. The solver in the DART engine is configured to time out and return an UNKNOWN status after attempting to solve for more than 0.2s. It can be observed that by applying concretization constraints, the Z3 solver is able to determine satisfiability of path conditions more quickly.

3.6 Advantages and Disadvantages

There are several advantages of DART-style dynamic symbolic execution:

1. The main advantage of DART-style dynamic symbolic execution is the engine's ability to produce input states that can be used as test cases for normal concrete testing methods, as opposed to only the result of testing. This makes it easy to verify test results by rerunning the generated input states on a pure concrete interpretor. It also provides human testers with an example of what kind of inputs would produce an error, instead of only the knowledge that an error can occure. Finally, the ability to export test cases from the DART engine allows for seamless integration with concrete testing methods such as manual testing and fuzzing, and standard coverage tracking modules can be used to calculate overall coverage of all the testing methods.
2. A secondary benefit of DART-style execution over EXE is the separation between the interpretor execution step and the symbolic solver step. Although not implemented on our DART module, it should be possible to run the interpretor and SMT solver asyncronously and on different machines. This would optimize computing resource and execution time for large programs with more complex path conditions to solve.

The main disadvantage of DART-style execution is the difficulty in taking advantage of concretization. Because the SMT solver in DART only runs after the concolic interpretor terminates, it only has access to the input state and return state from the interpretor. This means the solver can only attempt to simplify the satisfiability problem by concretizing individual inputs, rather than concretizing the difficult expression. By contrast, the SMT solver in EXE engine has access to all the internal states, making it much easier to concretize a complex function as it gets executed.

4 Conclusion

In summary, the EXE engine offers an efficient way to test programs when integration with concrete testing methods is not required. It is especially useful for long programs that have few branches, where the time requirement for the concrete execution is large compared to the additional overhead from running the symbolic solver. It is also useful for programs involving functions that are difficult for the symbolic solver to determine path condition satisfiability, since

the solver constraints within the EXE engine are much smaller than DART. In addition, the engine can take full advantage of concretization to simplify the problem.

Meanwhile, the DART engine offers a way to easily integrate dynamic symbolic execution with concrete testing methods and easily verify testing outcomes. It is also expecially useful for heavily branched programs where the overhead from running symbolic solver mid-execution is large, and it would be beneficial to run the solver offline to avoid holding up the concrete execution. However, since the DART engine accomplishes all this by sacrificing some execution efficiency, it may be less suitable for programs that are long and computationally expensive to run from start to finish. In addition, it may be less suitable for programs with overly complex path conditions since the DART engine is required to solve a cumulative path condition constraint all at once rather than one path condition at a time.

References

1. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques (2018)
2. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. SIGPLAN notices **40**(6), 213–223 (2005)