Software Testing, Quality Assurance and Maintenance Winter 2025

Lecture 9 — February 3, 2025

Patrick Lam *version 1*

We've talked about symbolic execution, which seems too good to be true, from the examples that we saw. What? Automatically finding inputs that explore all interesting behaviour? How?

The issue with symbolic executiion is that it's hard to scale. Consider these three problems.

1. Code that's hard to analyze: for instance, think of a cryptographic hash function. Smart people have tried to understand how they do what they do. Symbolic execution won't be able to analyze it.
2. Path explosion: the number of paths grows exponentially with the number of nested if statements; handling function calls can also lead to exponential growth; and loops can simply be unbounded.
3. Environment: in the examples we saw, the input was just integers. But in general there might be data structures (and pointers); files and databases; the network, and sockets; and threads (you need to explore all thread schedules).

Example. Here's a bit of a contrived example.

```
1  int obscure(int x, int y) {
2    if (x == complex(y))
3      error();
4    return 0;
5  }
```

Recall that symbolic execution is trying to find inputs that reach all program points, including the call to `error()`. So, it considers the `if` condition `x == complex(y)`. In the previous lecture, we had conditions that were just simple arithmetic and boolean expressions, and hence easy to make either true or false (by changing the values of the variables in the expressions), using an SMT solver. Now, it has to find values for `x` and `y` such that (in this case) `x == complex(y)`. But `complex()` is opaque to the SMT solver, and can be arbitrarily complicated.

- Who is being called? We see `complex()` here, but that could be a virtual function (`this.complex()`) and we don't necessarily know the runtime type of `this`. Or, it could be a function pointer.
- Function `complex()` could be a cryptographic function, like a password hash. Now you have to find a password that matches a given hash. Good luck!
- `complex()` could contain non-linear integer or floating point arithmetic, hard for SMT solvers to reason about.
- `complex()` could contain system calls, file I/O, network I/O, etc.

Dynamic Symbolic Execution (DSE)

We have some tools to mitigate the obstacles to symbolic execution, under the general name of "dynamic symbolic execution". Symbolic execution operates completely on symbolic inputs (X, Y), not concrete inputs (numbers 17, 5). But sometimes that's hard or impossible. We thus aim to use concrete execution (as formalized in the operational semantics) when symbolic execution is hard. The concrete execution will hopefully guide symbolic execution to useful parts of the code. This approach is therefore also known as a *concolic testing* ("**conc**rete" + "symb**olic**").

Let's see how it works. Here's the program again.

```
1   int obscure(int x, int y) {
2     if (x == complex(y))
3       error();
4     return 0;
5   }
```

One of the approaches to dynamic symbolic execution is called DART, for Directed Automated Random Testing.

Run 1. So, we'll start with some random inputs. Let `x=33` and `y=42`, which are concrete values you generated randomly. You concretely run `complex(42)` and find that it returns concrete value 567. Since $33 \neq 567$, these inputs lead to the else branch.

OK, now we want to explore the then-branch. Symbolic execution wants to negate the `if`-condition. Oops, that's too hard. Let's instead try running again. What `x` should we use? Why not `x=567`?

Run 2. This time we are trying with `x=567` and `y=42`, as indicated by the previous run. It turns out that `complex(42)` still returns 567 (it didn't have to, but it does). So the execution goes to the then branch, covering all branches, and finding the error.

Flavours of Symbolic Execution

In the previous lecture, we talked about *static symbolic execution* (or classical symbolic execution).

- simulates execution on program source code (with symbolic values)
- starts from entry point, computes strongest post-conditions (symbolically) for the method (by computing them for every program point).

Now, we're talking about *dynamic symbolic execution.*

- run/interpret program with concrete state (concrete values)
- in parallel, also compute symbolic state alongside concrete state ("concolic")
- occasionally, SMT solver generates new concrete inputs, increasing coverage.

Note that the notion of path coverage is key here. We're aiming to increase it.

There are two main flavours of DSE:

- EXE-style [CGP+06]: tools include KLEE (Imperial College London), SPF (NASA), Cloud9, S2E (EPFL)
- DART-style [GKS05]: tools include SAGE, PEX (Microsoft), CUTE (UIUC), CREST (UC Berkeley)

More on the differences between these later.

1 EXE

Let's start by seeing how the EXE-style dynamic symbolic execution works.

Overview. First, as with the example run we saw before, EXE runs the program concretely, with concrete inputs. You might use fuzzing or just hard-code some initial inputs.

But, at the same time, EXE runs the symbolic execution in parallel. The symbolic execution follows the concrete execution, keeping track of which values are inputs (symbolic) or not inputs (concrete). We've talked about computing path conditions before, and we'll do that here, in terms of the symbolic inputs.

Things get interesting at branch points, where our goal is to explore both cases. At every branch point:

- concrete execution will take a branch, say branch1.
- symbolic execution then forces execution into branch2 as well; it updates the input to do that, updating the path condition, and hence creating a new concrete state that reaches branch2.

We now have concrete inputs that reach both branch1 and branch2.

Details. For dynamic symbolic execution, a program state contains a path condition pc, as well as values for all variables. Variables always have a concrete value (in our formalization, an integer), and may have a symbolic value as well (an expression which may contain input variables e.g. X). To start, the state contains mappings to symbolic state like $\texttt{x} = X$ for all inputs X.

At each execution step:

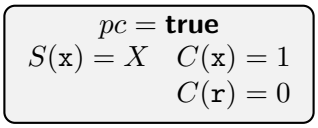
- update the concrete state by executing a program instruction concretely;
- update the symbolic state by executing the same instruction symbolically;
- if the instruction is a branch:
  - fork the execution state into two
  - true branch: conjoin the branch condition to the previous path condition
  - false branch: conjoin the negated branch condition to the previous path condition (that's how you get in the else branch)
  - for the branch that the concrete execution didn't take, ask the SMT solver to compute new initial concrete values from the symbolic state. These new inputs must be consistent

with the relevant path condition. Replace concrete values with the newly-computed ones, obtaining them by substituting the new inputs into the symbolic state. (Possibly this is impossible, in which case you give up on that path.)

Let's look at an example. We did this example with classical symbolic execution, but let's do EXE-style on it.

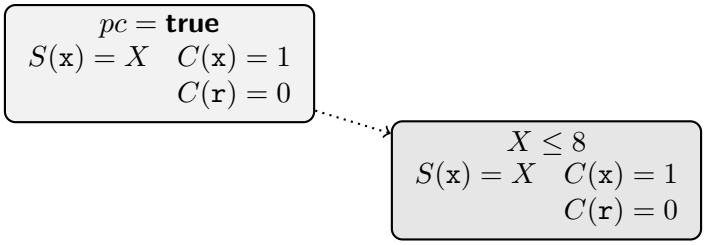
```
1   int proc(int x) {
2     int r = 0;
3
4     if (x > 8) { // (1)
5       r = x - 7
6     }
7
8     if (x < 5) { // (2)
9       r = x - 2;
10    }
11  }
```

Here's our initial state for dynamic symbolic execution.

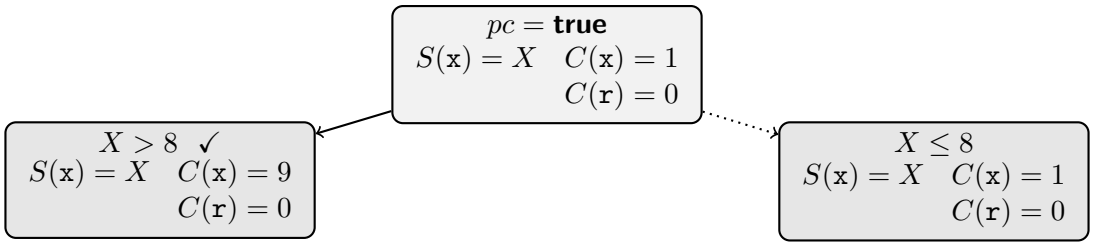


The start of the method is always reachable, so $pc = \textbf{true}$. There is one input X, so we have symbolic state $S(\texttt{x}) = X$. We also have an arbitrarily-drawn concrete value for `x`, which is 1, and `r` will be 0 soon enough.

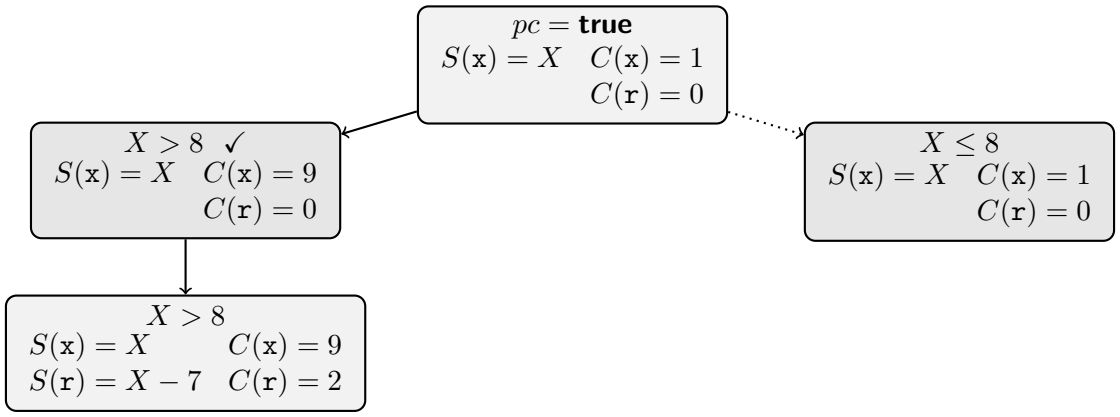
Let's start executing the program. We have a branch at (1) on condition $\texttt{x} > 8$. The concrete state $\texttt{x} = 1$ drives the concrete execution into the else branch, indicated by the dotted line.



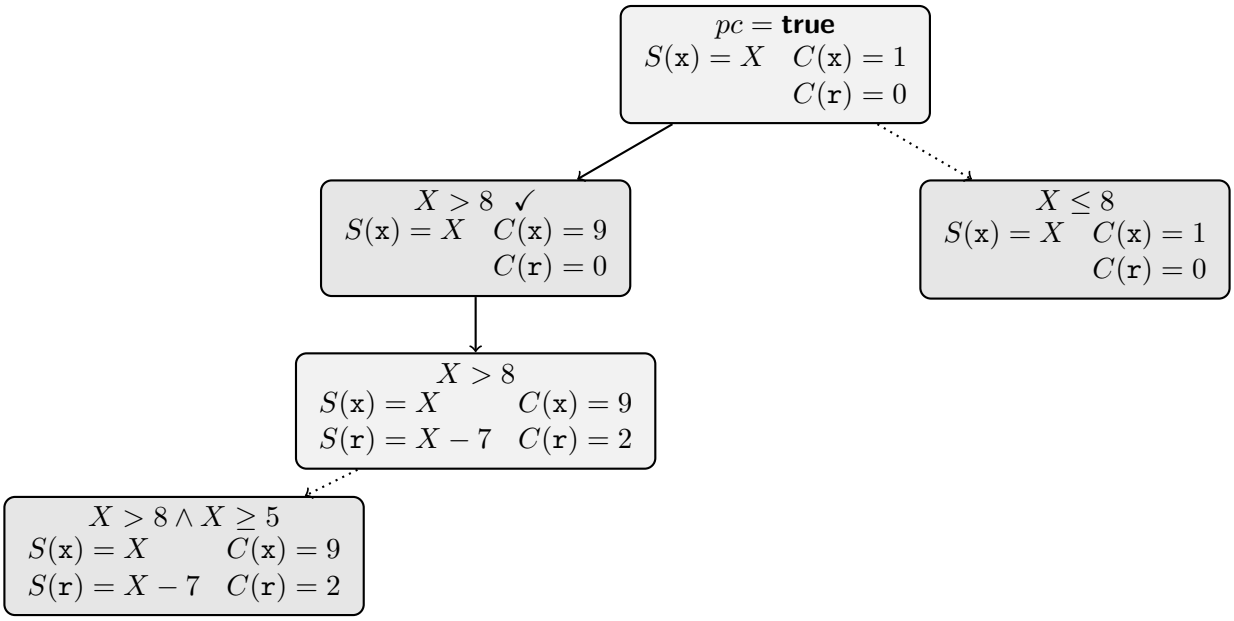
Now we also visit the then-branch. The path condition is simply the branch condition $X > 8$. We ask the SMT solver to generate a value for X that satisfies $X > 8$; it can (✓), and comes back with 9 for X and hence `x`. We don't update `r` since it doesn't depend on X.



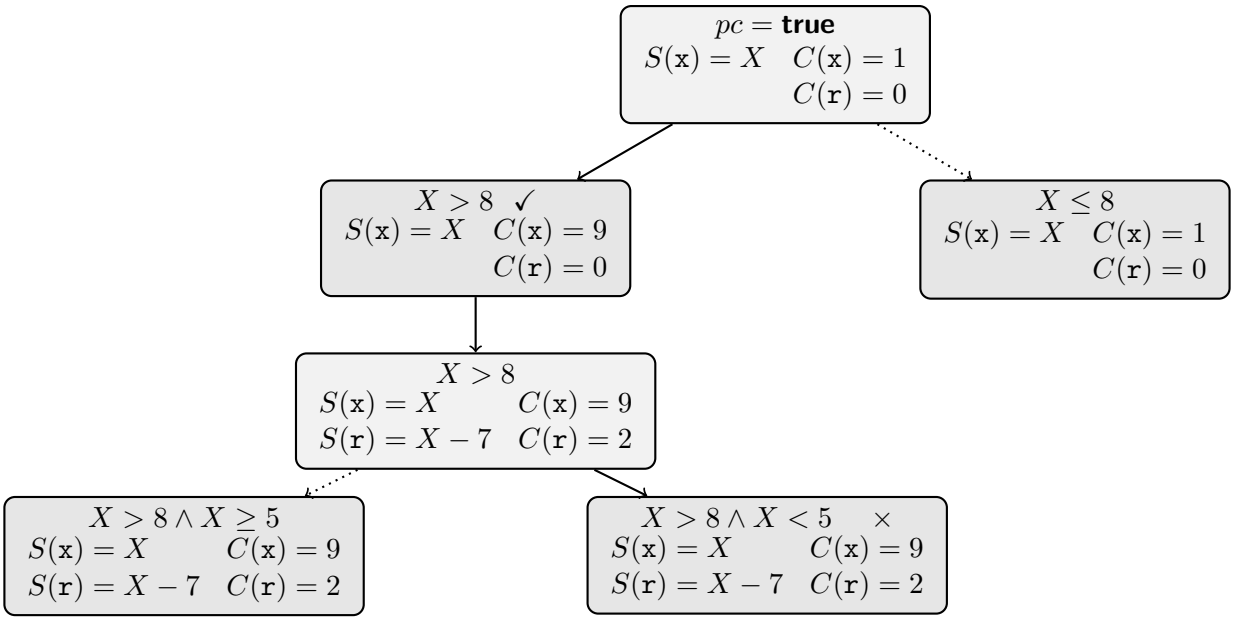
We continue execution on the then-branch, which reassigns `r`.



Let's continue with the next if-statement (2). In this case we have concrete `x` being 9, which runs to the else-branch, and finishes with no further changes in state.

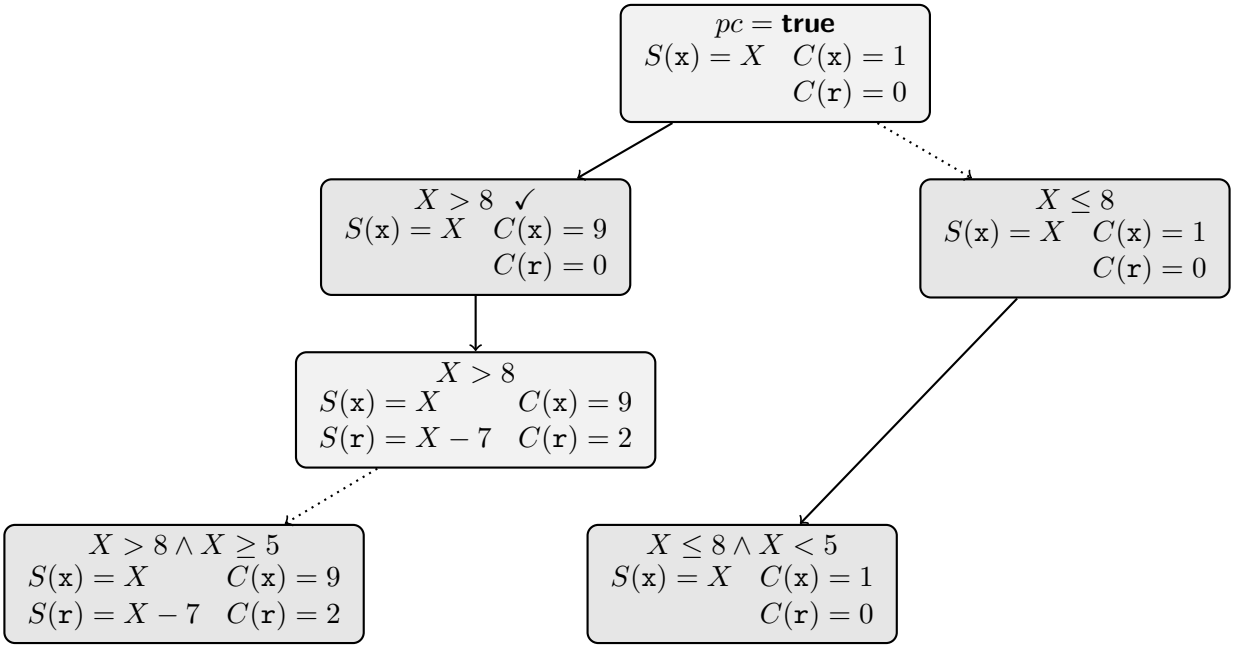


The other possibility is the then-branch of (2). Let's try that.

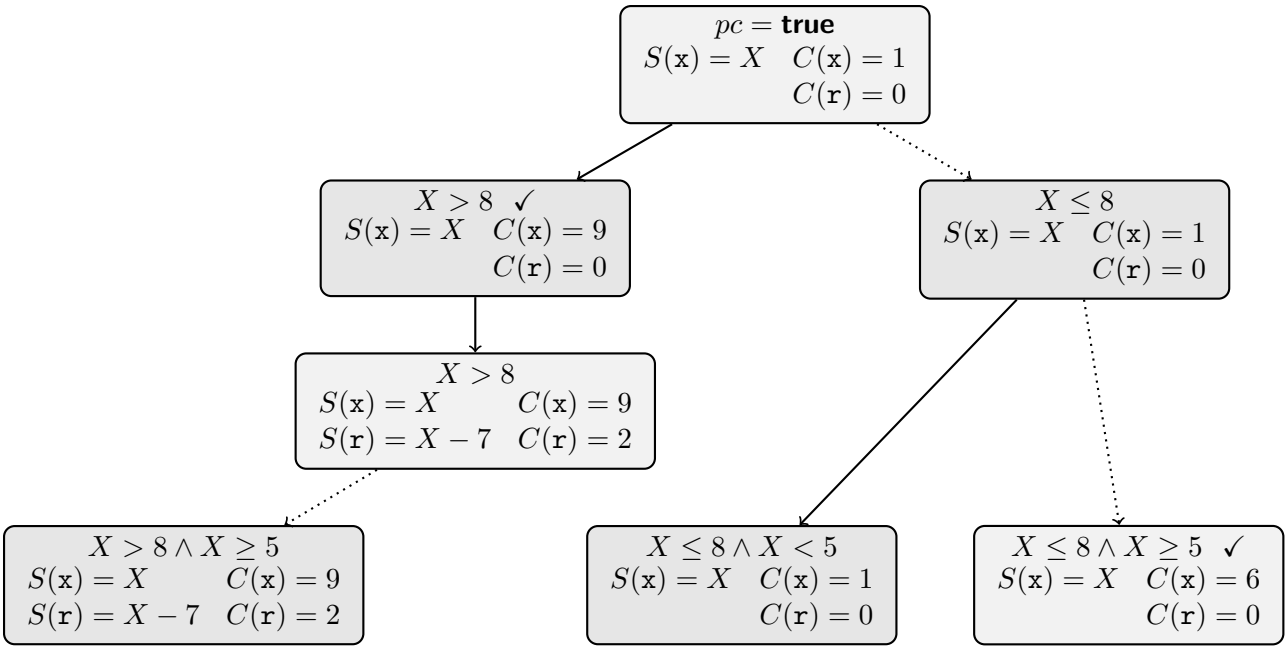


We have an unsatisfiable (×) path condition $X > 8 \wedge X < 5$ so we give up.

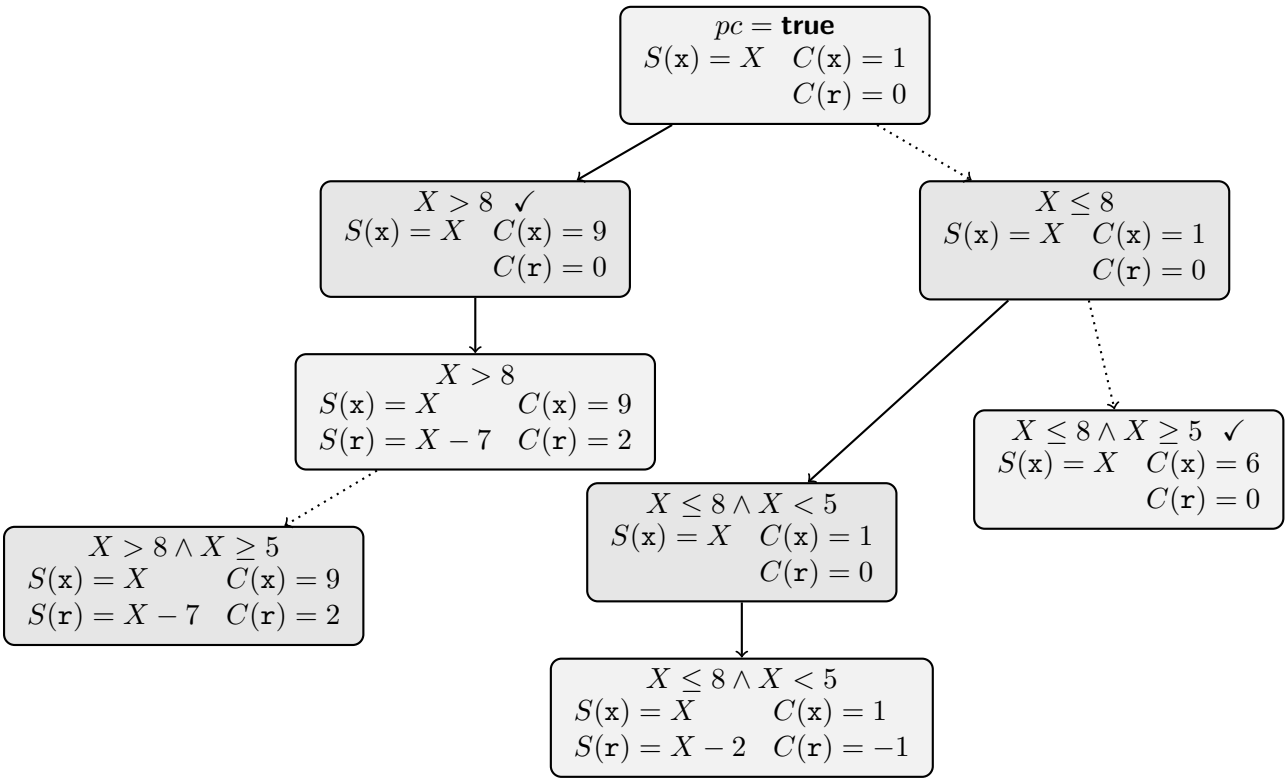
We've now completely explored the then-branch of (1). We still haven't finished exploring the else-branch of (1), so we continue executing it until we reach (2). In this case, the concrete state says to execute the then-branch of (2).



Let's put that on hold for now and explore the else-branch of (2). Here we falsify its branch condition and successfully (✓) find $X = 6$ as a solution to that path constraint, thus setting $\mathtt{x} = 6$.



Finally, we finish unfinished business with the then-branch of (2) and run through the body of that conditional.



We have satisfying assignments $X = 9$, $X = 1$, and $X = 6$, yielding test cases `proc(9)`, `proc(1)`, `proc(6)`.

EXE: Implementation considerations. An implementation of EXE needs to maintain multiple execution states while running the program, with both symbolic and concrete components. As we saw, we need to switch back and forth between different program points and states. The answer is a special-purpose virtual machine.

KLEE (`klee.github.io`) uses the LLVM bitcode interpreter to maintain and update state. This means that programs to be run under KLEE need to be compiled into LLVM bitcode. You can't compile everything into bitcode though, especially assembly, system calls, and some third-party libraries.

S^2E (`s2e.systems`) instead uses the QEMU virtual machine to fork and restore the entire machine state, including the operating system. It can therefore execute everything. It executes any code from the executable of interest symbolically. There's still the problem with system code (the code that couldn't be compiled into bitcode), and it runs that code concretely. Under-the-hood, it uses LLVM.

2 DART: Directed Automated Random Testing

Besides EXE, the other approach is DART. Here, we test the (entire) program concretely, recording the execution paths taken in each run, and then use symbolic execution to compute new inputs.

For the concrete runs, we can use random inputs (from fuzzing), user-provided inputs (from tests), or both. We instrument the program to record execution paths (similar to measuring path coverage), typically at binary level.

Given a run, we can then re-execute the program symbolically and compute path conditions along the way. As with EXE, at each branch condition, we adjust inputs to force that a different branch run, yielding new inputs that cover new execution paths.

We can repeat the concrete run/adjust input loop until we run out of time, or (ha, ha) we find all bugs.

There is pseudocode in the slides, but I think Wikipedia's description[1] is actually more clear, and I'll excerpt it here. It's not exactly what we do, since we have both symbolic and concrete values for variables sometimes.

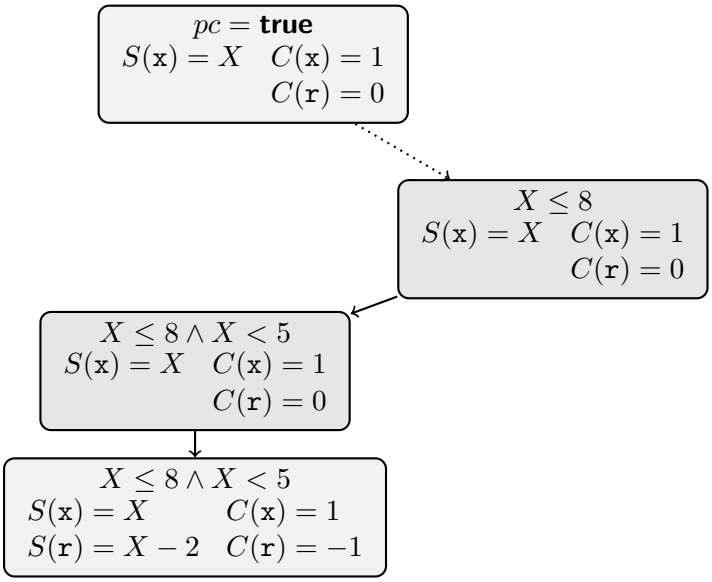
1. Classify a particular set of variables as *input variables*. These variables will be treated as symbolic variables during symbolic execution. All other variables will be treated as concrete values.
2. Instrument the program so that each operation which may affect a symbolic variable value or a path condition is logged to a trace file, as well as any error that occurs.
3. Choose an arbitrary input to begin with.
4. Execute the program.
5. Symbolically re-execute the program on the trace, generating a set of symbolic constraints (including path conditions).
6. Negate the last path condition not already negated in order to visit a new execution path. If there is no such path condition, the algorithm terminates.

[1] `https://en.wikipedia.org/wiki/Concolic_testing`

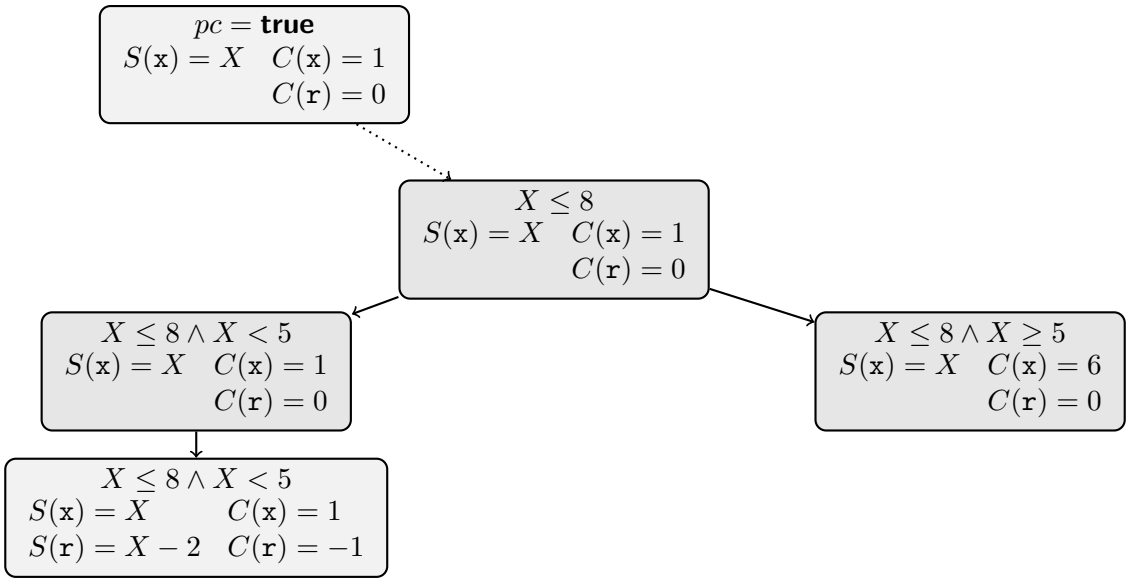
7. Invoke an automated satisfiability solver [SMT solver] on the new set of path conditions to generate a new input. If there is no input satisfying the constraints, return to step 6 to try the next execution path.
8. Return to step 4.

Let's run through the same program as from EXE again. I don't need to show the concrete execution step-by-step; DART runs it all through, anyway. We are again starting with input `x = 1`, i.e. test case `proc(1)`.

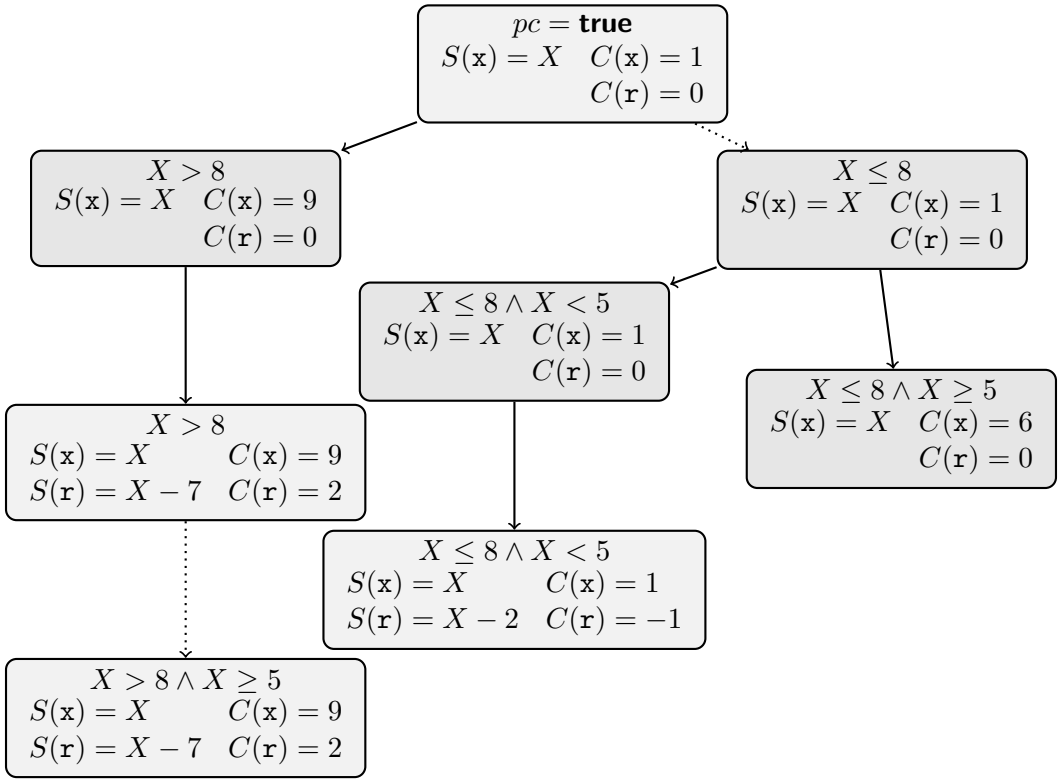
```
1    int proc(int x) {
2      int r = 0;
3
4      if (x > 8) { // (1)
5        r = x - 7
6      }
7
8      if (x < 5) { // (2)
9        r = x - 2;
10     }
11   }
```



Looking at the execution, the last path condition occurred at (2) where we had `x < 5`. We therefore form the new path condition $X \leq 8 \wedge \neg(X < 5)$; the SMT solver tells us that it is satisfiable (✓) and gives us solution $x = 6$ i.e. test case `proc(6)`. We run this test case and paste the new state onto the picture. Really, there are different nodes in the tree for the new run, from the root, but I don't know how to represent that, so I'm just leaving the old nodes in place. There are really multiple overlapping nodes e.g. for the root we also have a state with $C(\mathtt{x}) = 6$.



Because we don't run the then-case of (2), this is the end of the execution for `proc(6)`, and we've fully explored this part of the space. We back up to (1) and negate that path condition, giving us new path condition $\neg(X \leq 8)$. The SMT solver once again says that this is satisfiable (✓) and gives us solution $X = 9$ or test case `proc(9)`. We run the program again with this input.



Running through the program, we collect another path condition $X \geq 5$ and conjoin it with what we had before to get $X > 8 \wedge X \geq 5$. The last possible path condition we might explore is from

negating $X \geq 5$ to yield $X > 8 \wedge X < 5$, but the SMT solver tells us that this is unsatisfiable ($\times$), so we are done, and we once again have the inputs $9, 1, 6$, a.k.a. test cases `proc(9)`, `proc(1)`, and `proc(6)`.

A Second DART Example

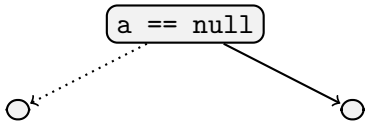
Let's now look at this code, which looks a bit more realistic.

```
1   void CoverMe(int [] a) {
2     if (a == null) return;
3       if (a.Length > 0)
4         if (a[0] == 1234567890)
5           throw new Exception("bug");
6   }
```

Note again the reliance on exceptions to indicate bugs. We are trying to manufacture the non-empty array with element 0 containing 1234567890. We start with input `null` for `a` and observe path constraint `a == null`.

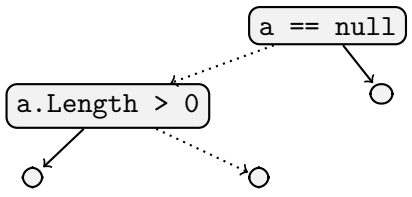
Constraints to Solve	Input	Observed constraints
	`null`	`a == null`

We then negate the observed path constraint to get `a != null`, which we pass to the SMT solver. It creates empty array `[]`. (It could create any array, but it's usually going to choose the simplest one first, unless you force it not to.) At this point we have the constraint tree:



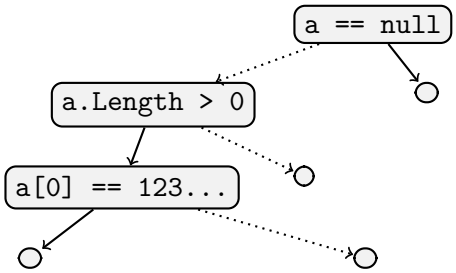
Constraints to Solve	Input	Observed constraints
	`null`	`a == null`
`a != null`	`[]`	

We execute and monitor the function with input `a = []` and observe a new constraint `a.Length > 0`, which evaluates to false on our input. We once again negate the path condition to yield constraint `a != null && a.Length > 0`. We ask the SMT solver to solve that constraint and get `a = [0]`.



Constraints to Solve	Input	Observed constraints
	`null`	`a == null`
`a != null`	`[]`	`a != null` `&& !(a.Length>0)`
`a != null &&` `a.Length > 0`	`[0]`	

Same thing, executing and monitoring. This time we get the constraint `a[0] != 1234...` and add it to our constraints to solve. That yields input `a=[1234567890]`.

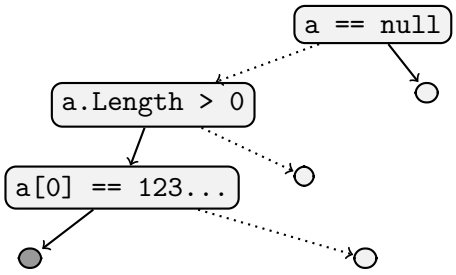


Constraints to Solve	Input	Observed constraints
	`null`	`a == null`
`a != null`	`[]`	`a != null` `&& !(a.Length>0)`

`a != null &&`
`a.Length > 0`
	`[0]`	`a != null`
		`&& !(a.Length>0)`
		`&& a[0] != 123...`

`a != null &&`
`a.Length > 0`
`a[0]==123...`
	`[123...]`	

We run on the final input and observe that we have achieved path coverage.



Constraints to Solve	Input	Observed constraints
	`null`	`a == null`
`a != null`	`[]`	`a != null` `&& !(a.Length>0)`

`a != null &&`
`a.Length > 0`
	`[0]`	`a != null`
		`&& !(a.Length>0)`
		`&& a[0] != 123...`

`a != null &&`
`a.Length > 0`
`a[0]==123...`
	`[123...]`	`!(a.Length>0)`
		`&& !(a.Length>0)`
		`&& a[0] == 123...`

SAGE: Zero to Crash in 10 Generations

We've talked quite a bit about dynamic symbolic execution. It works in industry too. [GLM12] talks about how it's been used at Microsoft to detect bugs in Windows, for instance to create crashing tests for a Media1 parser. (That is an easy article to read and I recommend it.) It's a lot like what we just saw. The slides do a better job at showing this, but I'll include excerpts here. The generation 0 seed file is all 0s:

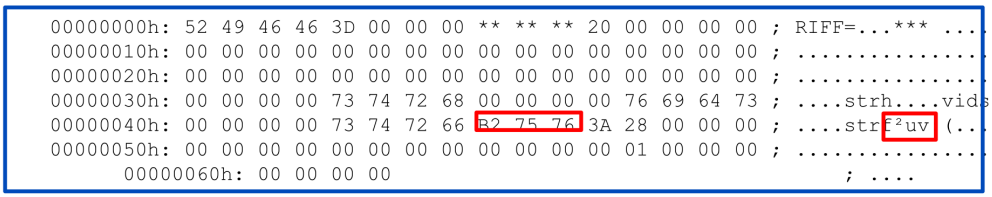
```
00000000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
      00000060h: 00 00 00 00                                     ; ....
```

Generation 0 – seed file

We do the same thing as we just did to get generation 1:

Generation 1

and repeat it 10 times, which yields a file that will crash the parser.



Generation 10

The main idea with SAGE is *generational input generation*. The insight is that the thing that is expensive is the symbolic execution. SAGE uses one symbolic execution to generate multiple tests at once. In the `CoverMe` example just above, we negated one constraint at a time, generating one new test per symbolic execution. The paper says:

> Given a path constraint, *all* the constraints in that path are systematically negated one by one, placed in a conjunction with the prefix of the path constraintleading to it, and attempted to be solved by a constraint solver.

Here's a program which we'll use to demonstate generational input generation.

```
1   void top(char input[4])
2   {
3     int cnt = 0;
4     if (input[0] == 'b') cnt++;
5     if (input[1] == 'a') cnt++;
6     if (input[2] == 'd') cnt++;
7     if (input[3] == '!') cnt++;
8     if (cnt > 3) crash();
9   }
```

We start with input `good`. This yields path constraint

$$i[0] \neq \texttt{'b'} \wedge i[1] \neq \texttt{'a'} \wedge i[2] \neq \texttt{'d'} \wedge i[3] \neq \texttt{'!'}$$

which then leads to four new inputs in generation 1: `bood`, `gaod`, `godd`, and `goo!`; each input is obtained from the original input and negating the path constraint, as we've seen before, and without rerunning the symbolic execution.

We repeat the same process on all of the inputs in generation 1 to get generation 2. For this example, we can enumerate all paths (note that this is more than branch coverage). In general there are too many paths and one quits at some point.

As for the SAGE tool itself, it was the first whitebox fuzzer used for security testing. Between 2007 and 2012 (when the paper was published), it ran for 400+ machine years, computing over 3 billion constraints, on hundreds of apps, finding hundreds of security bugs that no other tool found.

During the development of Windows 7, SAGE found about 1/3 of all bugs discovered by file fuzzing. Even after release, it continued to find bugs, and the fixes were quietly shipped to over 1 billion PCs.

DART Implementation Considerations. Now that we've talked about generational search, we can talk about some implementation details for DART-style dynamic symbolic execution.

Recall that the program under test is executed. That's a normal program execution, augmented with instrumentation or recording of executed instructions. Based on the recordings, a DART-style tool will compute a path condition. The ensuing symbolic execution may be computed in a separate phase (as done with SAGE) or on-the-fly (as with CUTE/CREST).

For generational search, the path condition computation happens offline for each branch point. Since there are a lot of branch points, it's possible to create many new inputs from a single concrete execution and symbolic execution, though the SMT solver has to be re-executed. This is easy to parallelize and distribute in the cloud.

3 EXE versus DART

The table you've all been waiting for.

EXE	**DART**
Fine-grained control of execution	Complete execution from first step
Shallow exploration	Deep exploration
(many queries early on)	(one query per run)
Online symbolic execution	Offline SE possible
(SE and interpretation in lockstep)	(execute along recorded trace)

4 Loops: Dynamic Symbolic Execution Doesn't Solve All Problems

Loops are a problem for symbolic execution. Symbolic execution can only reason about finitely many execution steps. Programs have loops and may execute for arbitrarily many steps. (Recall that when we write proofs about loops, we need invariants).

One way to handle this is to dynamically unroll loops. That is, copy-paste the loop body N times in a row, for some finite N. One would treat the backwards conditional branch implementing the loop as a branch condition for symbolic execution, and the choices are to do one more loop iteration, or to exit the loop.

If a loop can only run some small (finite) number of iterations, then unrolling can completely explore the behaviour of that loop. (This is also a key idea behind bounded model checking, which we'll discuss later).

Other loops can run unbounded numbers of iterations (e.g. just about any operation on a collection data structure). Symbolic execution will either get stuck or skip such loops. More powerful techniques are needed.

Actually, only loops whose iteration count depends on the input are problematic. If we know the maximum iteration count, then it's possible to execute the loop concretely.

Dynamic Symbolic Execution and Input-Dependent Loops . Here's some options to handle loops.

In EXE (online symbolic execution), then every iteration of the loop forks the execution, and the search algorithm (at runtime) can get bored and break out, or it can continue unrolling the loop and running one more iteration.

In DART (offline symbolic execution), we have an input and hence a concrete trace. The trace encodes some number of unrollings. Then, the search algorithm can cause a different number of loop executions by flipping one of the loop branches.

In either case, a naïve search algorithm might get stuck.

5 Concretization

Continuing on the "dynamic symbolic execution doesn't solve all problems" theme, let's think about this. DSE executes the program both concretely and symbolically.

- concrete execution is "easy" (just ask a VM)
- symbolic execution can be hard (e.g. multiplication)

What can we do when symbolic reasoning is too hard? One answer is *concretization*.

In concretization, we selectively replace symbolic inputs by concrete values.

- this simplifies symbolic reasoning; concrete expressions are easy, we said, and we can make a symbolic expression easier to reason about by substituting in concrete values. At the limit, if we replace all symbolic variables, then the operational semantics gives a concrete value for any expression.
- but, how do we know what to concretize? It's not always clear why an expression is hard to symbolically reason about.
- concretization can result in *unsoundness*; to mitigate this, we must record the concretization decision in the path condition;
- concretization will definitely result in *incompleteness*: we had all possible values of X and now we just have 5. What about $X = 6$? Or $X = -5$?

Example. Let's do a concretization. Here's a state and a program.

$true$			
	$C(X) = 5$	1	`if (m*m > size) {`
$S(\texttt{m}) = X + 2$	$C(\texttt{m}) = 7$	2	`// ...`
$S(\texttt{size}) = Y$	$C(\texttt{size}) = 256$		

The program contains condition `m*m > size`, which expands to symbolic $(X + 2)(X + 2) > Y$.

It turns out that symbolic multiplication is difficult for SMT solvers. We therefore choose to concretize variable X, using its current concrete value $C(X) = 5$. The condition therefore becomes $49 = (5 + 2)(5 + 2) > Y$.

If we want to enter the then-branch of the if statement, we do as usual: update the path condition with the (concretized) branch condition $49 > Y$, solve for symbolic input Y, and get e.g. $Y = 48$.

$49 > Y$			
	$C(X) = 5$	1	`if (m*m > size) {`
$S(\texttt{m}) = X + 2$	$C(\texttt{m}) = 7$	2	`// ...`
$S(\texttt{size}) = Y$	$C(\texttt{size}) = 48$		

Now say that inside the then-branch we furthermore have `if (m < 5)`. Symbolically, this works out to $X + 2 < 5$ due to the symbolic value for `m`. To explore this then-branch, we'd ask the SMT solver for a solution, and we could well get $X = 2$. But that's wrong. We had concretized X to 5, and now we're saying that X has to be 2.

How to avoid this: record $X = 5$ in the path condition when you concretize. The SMT solver might then say "no solutions" (you're missing behaviours), but you won't get wrong results.

Concretization Algorithm. Here's an algorithm for concretizing variables in a symbolic expression, because it's too hard to symbolically compute with. As input:

- a state, including path condition pc and concrete and symbolic states C and S;
- a symbolic expression E to concretize

The algorithm:

- choose variables $x_1, \ldots, x_k$ in E to concretize
- let E' be the result of replacing x_i by $C(x_i$ in E
- let the concretization constraint CC be $x_1 = C(x_1) \wedge \cdots \wedge x_k = C(x_k)$ and add it to pc;
- return the result of evaluating E' on symbolic state S.

Soundness and Completeness. Conceptually, each path that you explore in the dynamic symbolic execution is exact. The concrete results are concrete, and the symbolic state is computing what is called the strongest postcondition. There's no approximation here, on a per-path basis.

But, symbolic execution (static or dynamic) doesn't guarantee that you explore all of the paths (remember what we said about loops); it underapproximates. In finite time, you only visit a subset of the paths. There is a sort of "eventual" completeness in that if you let it run for long enough, you'll observe any behaviour that is possible. It's sound, in that you don't see anything that is infeasible.

Concretization remains sound (if you modify the path condition as stated above). But you are exploring less. In particular, you lose the eventual completeness property mentioned above.

Summary of concretization. This technique, concretization, is a key technique for making dynamic symbolic execution work in practice. It enables external calls, which you can't evaluate symbolically. You can evaluate the calls if you concretize the call arguments and just call the callee.

We talked about concretization constraints. It is possible to not add them to the path condition, as was done in the original DART. Then the dynamic symbolic execution might diverge when it has contradictory values for symbolic variables. This isn't sound. But if you are willing to sacrifice soundness, you can deal with divergences with random restarts.

References

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 322–335, New York, NY, USA, 2006. Association for Computing Machinery.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

[GLM12] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. *Queue*, 10(1):20–27, January 2012.