Symbolic Execution: Challenges

Path explosion and state space blow-up

- Programs have lots of branches, loops, inputs:
  - the number of distinct execution paths grows **exponentially** in the size of the program.
    Each conditional (if/else) doubles potential paths; nested loops multiply things further.
- Symbolic execution tries to explore all paths, this quickly becomes intractable.

- **The issue:** Path explosion makes the analysis slow or impossible;
  - one cannot symbolically explore *all* paths for moderate or large programs.

```
function f(a) {
  var x = a;
  while (x > 0) { x--; }
}
```

Assume a_0 that is the initial symbolic value

How symbolic execution forks

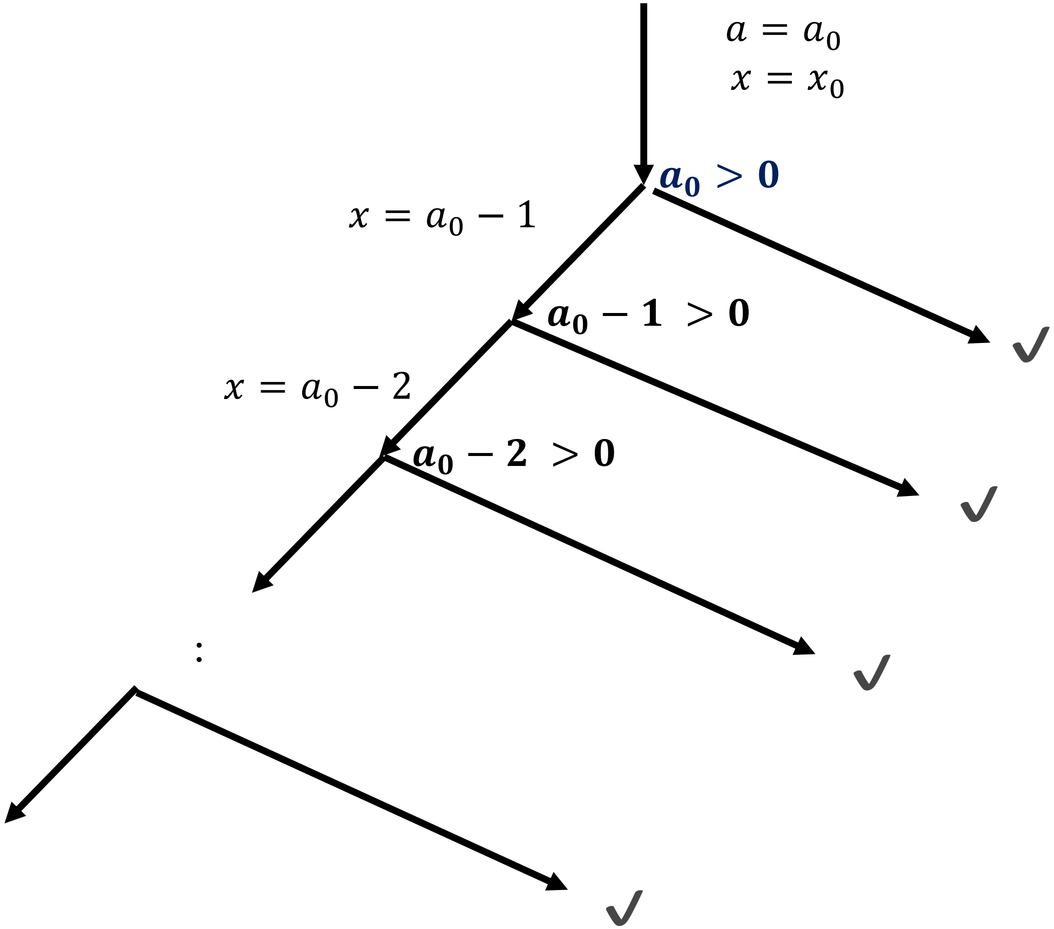
While loop: (x > 0) is the guard, if x is symbolic, the engine **forks**:

Entry loop: add constraint x > 0, then execute x := x - 1.
Exit loop: add constraint x ≤ 0, then leave the loop.

Start: x = a_0 .
1st check: forks on a_0 > 0 vs a_0 ≤ 0.
If we took the loop once, now x = a_0 - 1.
2nd check: forks on a_0 - 1 > 0 vs a_0 - 1 ≤ 0.
If we took it twice, x = a_0 - 2, and so on.

```
function f(a) {
  var x = a;
  while (x > 0) { x--; }
}
```

Assume a_0 that is the initial symbolic value



```
function f(a) {
  var x = a;
  while (x > 0) { x--; }
}
```

Assume a_0 that is the initial symbolic value

Exiting **after exactly k iterations** yields the path condition:
True for the first k checks: $a_0 > 0, a_0 - 1 > 0, \ldots a_0 - (k-1) > 0$
Then exit on the k-th: $a_0 - k \le 0$ aka $a_0 = k$

There's **one feasible path per non-negative integer k**.

Since k is unbounded, there are **countably infinitely many** distinct paths (each with a different path condition).

Path explosion

The same reasoning applies to recursive calls

```
void example(int a, int b) {
    if (a < 0) {
        if (b > 0) {
            // Path 1
        } else {
            // Path 2
        }
    } else {
        if (b > 0) {
            // Path 3
        } else {
            // Path 4
        }
    }
}
```

The symbolic execution explores **4 possible paths**, corresponding to all truth combinations of `(a < 0)` and `(b > 0)`

For two symbolic variables a and b, there are four distinct paths.
Adding a third symbolic variable c would create eight paths.

Because symbolic execution must analyze the true and false branch every time a conditional expression is encountered.

Path explosion

Data Structures

```
int foo(int *A, int n, int k) {
   int i = 0, sum = 0;
   while (i < n) {
      if (A[i] == k) {     // branch 1
         sum += 1;
      } else {             // branch 2
         sum -= 1;
      }
      if (sum < -5) {      // alarm
         return -1;
      }
      i++;
   }
   return sum;
}
```

Symbolic execution:
at each iteration one forks on
A[i]==k vs A[i]!=k

We have 2^n **paths**.

Path explosion

Challenge:

Handling Large Execution Trees

Handling Large Execution Trees

#1: Over-approx to prune big subtrees (sound but maybe imprecise)

```
int foo(int *A, int n, int k) {
  int i = 0, sum = 0;
  while (i < n) {
    if (A[i] == k) {    // branch 1
      sum += 1;
    } else {            // branch 2
      sum -= 1;
    }
    if (sum < -5) {     // alarm
      return -1;
    }
    i++;
  }
  return sum;
}
```

(Hoare-like reasoning) Loop invariant:

$$(0 \leq i \leq n) \land (sum \in [-i, i])$$

Handling Large Execution Trees

#1: Over-approx to prune big subtrees (sound but maybe imprecise)

```
int foo(int *A, int n, int k) {
   int i = 0, sum = 0;
   while (i < n) {
      if (A[i] == k) {     // branch 1
         sum += 1;
      } else {             // branch 2
         sum -= 1;
      }
      if (sum < -5) {      // alarm
         return -1;
      }
      i++;
   }
   return sum;
}
```

Loop invariant:

$$(0 \leq i \ \leq n) \ \wedge (sum \in [-i, i])$$

Immediate pruning when $n \leq 5$:

the alarm `sum < -5` is **unreachable** when `n ≤ 5`.

We can **skip exploring all 2^n branches** for every path with `n ≤ 5`.

Handling Large Execution Trees

#1: Over-approx to prune big subtrees (sound but maybe imprecise)

```
int foo(int *A, int n, int k) {
  int i = 0, sum = 0;
  while (i < n) {
    if (A[i] == k) {     // branch 1
      sum += 1;
    } else {             // branch 2
      sum -= 1;
    }
    if (sum < -5) {      // alarm
      return -1;
    }
    i++;
  }
  return sum;
}
```

Loop invariant:

$$(0 \leq i \ \leq n) \ \wedge (sum \in [-i, i])$$

Immediate pruning when `n` ≤ 5:

the alarm `sum < -5` is **unreachable** when `n ≤ 5`.

We can **skip exploring all 2^n branches** for every path with `n ≤ 5`.

Memory-safety assumption (precondition):

If we require `0 ≤ n ≤ len(A)`, the access `A[i]` is in-bounds.

No need to track Out Of Bound checks; those subtrees are **cut**.

Handling Large Execution Trees

#1: Over-approx to prune big subtrees (sound but maybe imprecise)

```
int foo(int *A, int n, int k) {
  int i = 0, sum = 0;
  while (i < n) {
    if (A[i] == k) {     // branch 1
      sum += 1;
    } else {             // branch 2
      sum -= 1;
    }
    if (sum < -5) {      // alarm
      return -1;
    }
    i++;
  }
  return sum;
}
```

Loop invariant:

$$(0 \leq i \leq n) \land (sum \in [-i, i])$$

Immediate pruning when `n ≤ 5`:
the alarm `sum < -5` is **unreachable** when `n ≤ 5`.
We can **skip exploring all 2^n branches** for every path with `n ≤ 5`.

Memory-safety assumption (precondition):
If we require `0 ≤ n ≤ len(A)`, the access `A[i]` is in-bounds.
No need to track Out Of Bound checks; those subtrees are **cut**.

Effect: For the whole slice of states where n ≤ 5, the execution tree collapses to **one summarized node** (no alarm).
For n ≥ 6, we continue (since the over-approx can't rule the alarm out)

Handling Large Execution Trees

#2: Under-approx to get a bug witness fast (no false positives)

```
int foo(int *A, int n, int k) {
  int i = 0, sum = 0;
  while (i < n) {
    if (A[i] == k) {     // branch 1
      sum += 1;
    } else {             // branch 2
      sum -= 1;
    }
    if (sum < -5) {      // alarm less k
      return -1;         // than exptected
    }
    i++;
  }
  return sum;
}
```

We *assert a concrete under-approx case* for the first 6 iterations:

`i = 0, n = 6, and A[0..5] != k`

Handling Large Execution Trees

#2: Under-approx to get a bug witness fast (no false positives)

```
int foo(int *A, int n, int k) {
  int i = 0, sum = 0;
  while (i < n) {
    if (A[i] == k) {     // branch 1
      sum += 1;
    } else {             // branch 2
      sum -= 1;
    }
    if (sum < -5) {      // alarm less k
      return -1;         // than exptected
    }
    i++;
  }
  return sum;
}
```

We *assert a concrete under-approx case* for the first 6 iterations:

```
i = 0, n = 6, and A[0..5] != k
```

The path is straight-line (no forking):

After 1st iter: sum = −1

…

After 6th iter: sum = −6 < −5 ⇒ return -1.

This provides a witness input of a (real) bug

n = 6, A[0..5] = {k+1, k+1, k+1, k+1, k+1, k+1} (or any ≠ k)

Handling Large Execution Trees

#2: Under-approx to get a bug witness fast (no false positives)

```
int foo(int *A, int n, int k) {
  int i = 0, sum = 0;
  while (i < n) {
    if (A[i] == k) {     // branch 1
      sum += 1;
    } else {             // branch 2
      sum -= 1;
    }
    if (sum < -5) {      // alarm less k
      return -1;         // than exptected
    }
    i++;
  }
  return sum;
}
```

We *assert a concrete under-approx case* for the first 6 iterations:

```
i = 0, n ≥ 6, and A[0..5] != k
```

The path is straight-line (no forking):
After 1st iter: sum = −1
…
After 6th iter: sum = −6 < −5 ⇒ return -1.

This provides a witness input of a (real) bug
n = 6, A[0..5] = {k+1, k+1, k+1, k+1, k+1, k+1} (or any ≠ k)

Effect: For n ≥ 6, instead of exploring an exponential tree, we **pick 1 guded path** to the alarm and stop (or keep a few patterns if we want diversity)

Handling Large Execution Trees

#3: Putting them together (execution strategy)

```
int foo(int *A, int n, int k) {
   int i = 0, sum = 0;
   while (i < n) {
      if (A[i] == k) {     // branch 1
         sum += 1;
      } else {             // branch 2
         sum -= 1;
      }
      if (sum < -5) {      // alarm less k
         return -1;        // than exptected
      }
      i++;
   }
   return sum;
}
```

Step 1
Pre-pass (Over-approx):
Compute invariants and **global pruning rules**:

If $n \leq 5$ then alarm unreachable. Result: **prune entire subtree**.

If $n > len(A)$ then memory unsafe. Result: filter by precondition

These rules are cached at the loop head and function entry.

Handling Large Execution Trees

#3: Putting them together (execution strategy)

```
int foo(int *A, int n, int k) {
   int i = 0, sum = 0;
   while (i < n) {
      if (A[i] == k) {     // branch 1
         sum += 1;
      } else {             // branch 2
         sum -= 1;
      }
      if (sum < -5) {      // alarm less k
         return -1;        // than exptected
      }
      i++;
   }
   return sum;
}
```

Step 2
Symbolic execution with pruning:
When the executor sees a state with n ≤ 5, it **does not fork** inside the loop. (alarm absent.)

When it sees n ≥ 6, it **does not fork 2^n paths**.
Strategy: asks the **under-approx oracle** for a **bug pattern**; it injects the conjunct A[0..5] != k and executes a **single path** to return -1.

Handling Large Execution Trees

#3: Putting them together (execution strategy)

```
int foo(int *A, int n, int k) {
   int i = 0, sum = 0;
   while (i < n) {
      if (A[i] == k) {     // branch 1
         sum += 1;
      } else {             // branch 2
         sum -= 1;
      }
      if (sum < -5) {      // alarm less k
         return -1;        // than exptected
      }
      i++;
   }
   return sum;
}
```

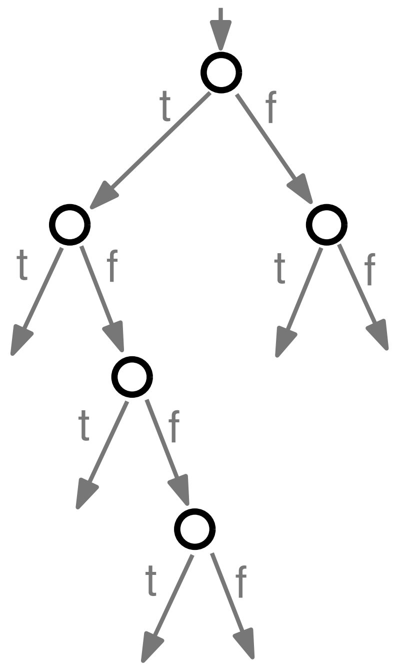
Step 3

For any remaining alarm candidates (e.g., if the under-approx. oracle didn't find one), try to **prove absence** with a suitable abstraction.

Handling large execution trees

Heuristically select which branch to explore next

- Select at random
- Select based on coverage
- Prioritize based on distance to ”interesting” program locations
- Interleaving symbolic execution with random testing



Challenges of Symbolic Execution

- **Environment modeling:** Dealing with native code or library calls

Symbolic model for library `y = sqrt(x);`

If `sqrt` is a native library call (implemented in assembly or math library), the symbolic executor doesn't know its internal behavior.

Challenge:
It canmoot derive the relation between `x` and `y` symbolically.
It may either concretize `x` (pick one value) or drop the path (loss of coverage).

Impact:
Path explosion is reduced (by dropping paths), but **soundness is lost**.

Typical fix:
Provide *models* for common math functions: e.g., $y \geq 0 \land y^2 = x$.

System calls

```
n = read(fd, buf, len);
if (n < 0) error();
```

Symbolic execution doesn’t know what the OS will return.

Challenges:
What is in `buf`? Is `n` symbolic or concrete?
Each possible return value creates a new path.

Fixes:
Abstract models: `n ∈ [0, len]` and `buf` = symbolic array of length `n`.

Pointer aliasing and memory layout in native libraries

```
memcpy(dst, src, n);
```

Native functions like `memcpy`, `strcpy`, or `malloc` are highly optimized and platform-specific.

Challenges:
If `src` or `dst` are symbolic, modeling byte-by-byte copying symbolically is costly.
Alias relationships (if `src` and `dst` overlap) can make the SMT constraints explode.

Fix:
Use logical summary instead of actually iterating byte-by-byte (nly the final effect)

$$\forall i \in [0, n): dest[i] = src[i]$$

Uninterpreted external functions

```
token = SHA256(data);
```

Challenge:
Cryptographic functions are intentionally opaque; symbolic reasoning is impossible.

Fix:
Treat them as **uninterpreted functions**: only reason about equality (e.g., `SHA256(x) == SHA256(y)` ⇒ `x == y`).

Cross Language Calls

```
extern "C" { fn fast_hash(input: *const u8, len: usize) -> u32; }
```

Challenge:
Different calling conventions, heap models, and memory ownership rules.
The symbolic engine must switch between language runtimes.

Fix:
Use **hybrid symbolic interpreters** or translate native components into *logical summaries* (contracts on input–output relations).

Challenges of Symbolic Execution

- Solver limitations: Dealing with complex path conditions

Path conditions grow exponentially

```
int foo(int x, int y) {
    if (x * y > 10) {
        if (x - y == 3) {
            assert(x < 100);
        }
    }
}
```

Symbolic state:

At the assertion, the path condition is:

$$(x * y > 10) \wedge (x - y = 3) \wedge \neg(x < 100)$$

The solver must check:

$$(x * y > 10) \wedge (x - y = 3) \wedge (x \geq 100)$$

Intermezzo: SAT Sat Solver Again

A formula is **linear** if **each variable appears at most to the first power** and variables are **not multiplied or divided by each other**.

Allowed operations:
Addition and subtraction of variables.
Multiplication or division by **known constants**.
Comparisons using $=, \neq, <, \leq, >, \geq$.
Example:

$$3x - 2y \leq 7$$
$$x + 4y = 10$$
$$x \geq 0$$

Intermezzo: Sat Solver again

If any term multiplies or divides **two variables**, or uses non-linear functions (e.g., powers, `sin`, `exp`, etc.), it becomes **non-linear**.

Example:

$$x * y > 10$$
$$x^2 + y \leq 5$$
$$sin(x) = 0$$

Intermezzo: Sat Solver Again

- **Linear arithmetic** is well-understood, efficient solving algorithms (based on linear programming, Gaussian elimination, or simplex methods).
- Solvers can handle **thousands of linear constraints** quickly.
- **Non-linear arithmetic** requires far more expensive reasoning
- That's why symbolic execution engines and SMT solvers like **Z3** have specialized "theories":
  - **LIA** = Linear Integer Arithmetic
  - **LRA** = Linear Real Arithmetic
  - **NIA / NRA** = Non-linear Integer/Real Arithmetic (much slower)

Back to our example

Path conditions grow exponentially

```
int foo(int x, int y) {
    if (x * y > 10) {
        if (x - y == 3) {
            assert(x < 100);
        }
    }
}
```

Symbolic state:

At the assertion, the path condition is:

$$(x * y > 10) \wedge (x - y = 3) \wedge \neg(x < 100)$$

The solver must check:

$$(x * y > 10) \wedge (x - y = 3) \wedge (x \geq 100)$$

Path conditions grow exponentially

```
int foo(int x, int y) {
    if (x * y > 10) {
        if (x - y == 3) {
            assert(x < 100);
        }
    }
}
```

Symbolic state:

At the assertion, the path condition is:

$$(x * y > 10) \land (x - y = 3) \land \neg(x < 100)$$

The solver must check:

$$(x * y > 10) \land (x - y = 3) \land (x \geq 100)$$

This constraint includes **non-linear arithmetic** (`x * y`),
which most SMT solvers handle *poorly*

The result:
Solver may time out.

Path Conditions with data structures

```
if (arr[a] == arr[b]) {
    if (map[key] == val) { ... }
}
```

Symbolic execution must exploit theories of arrays and maps: and these are embedded in SMT formulas.

Challenge:
Each array access or update adds *quantifiers* and nested *select/store* terms.
Solving these leads to **heavy quantifier instantiation** and exponential blow-up.

Strategy: apply **array abstraction**
(summarize properties instead of enumerating cells).

Path Conditions with Chains

```
for (i = 0; i < n; i++) {
    if (hash[i] == 42) break;
}
```

Unrolling the loop, the path condition will look like:

$(hash[0] \mathrel{!=} 42) \land (hash[1] \mathrel{!=} 42) \land \ldots \land (hash[k] = 42)$

Path Conditions with Chains

```
for (i = 0; i < n; i++) {
    if (hash[i] == 42) break;
}
```

Unrolling the loop, the path condition will look like:

(hash[0] !=42)∧(hash[1] !=42)∧...∧(hash[k]=42)

Challenge:
k iterations implies k disjunctive constraints; real programs have thousands of loops!!!.

Strategy:
Use **loop invariants** to avoid enumerating all iterations.

A smart approach

- Mix symbolic with concrete execution

Concolic testing

Mix concrete and symbolic execution = "concolic"

CONCOLIC = CONCrete + symbOLIC

- Perform concrete and symbolic execution side-by-side
- Gather path constraints while program executes
- After one execution, negate one decision, and re-execute with new input that triggers another path

The core idea

- Symbolic execution explores all paths *symbolically*, but that quickly leads to **path explosion** and **solver bottlenecks**.
- **Concolic execution** mitigates by:
  - Executing the program **concretely** on specific inputs.
  - Simultaneously **tracking symbolic constraints** along that *single* concrete path.
  - Using those constraints to generate *new* inputs that explore new paths.
- Concolic execution = iterative approach:
  Concrete run; record symbolic constraints;
  solve to get new inputs; next run; .....

Concolic step by step

```
int foo(int x, int y) {
    if (x > 5) {
        if (y == x + 2)
            bug();
    }
}
```

Concolic step by step

```
int foo(int x, int y) {
    if (x > 5) {
        if (y == x + 2)
            bug();
    }
}
```

Step 1
Start with a concrete test
`x = 0, y = 0.`

Concrete run follows the **false** branch of
`x > 5.`
No bug triggered.

Symbolic execution records:
`Path condition: (x ≤ 5)`

Concolic step by step

```
int foo(int x, int y) {
    if (x > 5) {
        if (y == x + 2)
            bug();
    }
}
```

Step 2
Negate one branch condition
To explore a new path, **flips one condition** in the path constraint:
`(x > 5)`

The solver gives a new input,
`x = 6, y = 0.`

Concolic step by step

```
int foo(int x, int y) {
    if (x > 5) {
        if (y == x + 2)
            bug();
    }
}
```

Step 3
Run again with new input
Concrete execution
takes the **true** branch of x > 5,
checks y == x + 2 (0 == 8) which evaluates false.

Path condition:
$(x > 5) \land (y \neq x + 2)$

Negate $y \neq x + 2$
new constraint $(y == x + 2)$.

Solver produces x = 6, y = 8.

Concolic step by step

```
int foo(int x, int y) {
    if (x > 5) {
        if (y == x + 2)
            bug();
    }
}
```

Step 4

Run again

Concrete execution triggers **bug();**

Found a real bug with **no false positives**.

Concolic step by step

```
int foo(int x, int y) {
    if (x > 5) {
        if (y == x + 2)
            bug();
    }
}
```

The strategy: the concolic engine explored all paths **sequentially, guided by concrete runs**, instead of exploring all 4 combinations symbolically.

Discussion

Symbolic execution	How concolic testing helps
Exponential path explosion	One path per iteration (systematic exploration)
Constraint solving overload	Smaller, incremental path constraints per run
Missing real inputs	Concrete execution gives actual input values
Unmodeled library/native code	Concrete execution uses the real runtime behavior

Concolic execution: the algorithm

Repeat until all paths are covered

- Execute program with concrete input i and collect symbolic constraints at branch points: C
- Negate one constraint to force taking an alternative branch b': Constraints C'
- Call constraint solver to find solution for C': New concrete input i'
- Execute with i' to take branch b'
- Check at runtime that b' is indeed taken
  Otherwise: "divergent execution"

Example

```
function f(a) {
    if (Math.random() < 0.5) {
        if (a > 1) {
            console.log("YES");
        }
    }
}
```

```
function f(a) {
    if (Math.random() < 0.5) {
        if (a > 1) {
            console.log("YES");
        }
    }
}
```

Type	Values	Notes
Concrete	a = 0	The real input
Symbolic	a_0	Symbolic values
Path Cond.	True	

```
function f(a) {
    if (Math.random() < 0.5) {
        if (a > 1) {
            console.log("YES");
        }
    }
}
```

Step 1 – `if (Math.random() < 0.5)`

Concrete

Suppose the runtime call returns `Math.random() = 0.3`.

`0.3 < 0.5 -----` **true** branch taken.

Symbolic

Since `Math.random()` is *external* we record its result, but not a symbolic variable.

New path condition:

$$PC = (random < 0.5)$$

```
function f(a) {
    if (Math.random() < 0.5) {
        if (a > 1) {
            console.log("YES");
        }
    }
}
```

Step 2 – if (a > 1)

Concrete

a = 0 then the gaurda 0 > 1 is **false**, the inner conditional is not executed.

Nothing printed.

Symbolic

Add condition for the branch actually taken:

$$PC = (random < 0.5) \land (a_0 \leq 1)$$

```
function f(a) {
    if (Math.random() < 0.5) {
        if (a > 1) {
            console.log("YES");
        }
    }
}
```

RUN #1 SUMMARY

Run	Concrete input(s)	Branch outcome	Path Condition	Output
1	a = 0, random = 0.3	Outer = true, Inner = false	(random < 0.5) ∧ ($a_0 \leq 1$)	(none)

```
function f(a) {
    if (Math.random() < 0.5) {
        if (a > 1) {
            console.log("YES");
        }
    }
}
```

RUN #2

Step 4 – Generate new paths

Option A – Flip inner condition

Negate $(a_0 \leq 1)$ this becomes $(a_0 > 1)$

Solver solution: `a = 2`.

New run (#2): `a = 2`, keep `random = 0.3`

Path condition $(random < 0.5) \land (a_0 > 1)$

Concrete run prints `"YES"`.

```
function f(a) {
    if (Math.random() < 0.5) {
        if (a > 1) {
            console.log("YES");
        }
    }
}
```

RUN #3

Step 4 – Generate new paths

Option B – Flip outer condition
Negate (random < 0.5) that is (random ≥ 0.5) forcing the value (e.g., 0.8).

New run (#3): random = 0.8, a = 2

Path condition (random ≥ 0.5)

No "YES" printed.
This is called **divergent execution**

SUMMARY

Path	Condition (symbolic)	Example concrete values	Output
1	$random < 0.5 \land a \leq 1$	$random = 0.3, a = 0$	(no output)
2	$random < 0.5 \land a > 1$	$random = 0.3, a = 2$	"YES"
3	$random \geq 0.5$	$random = 0.8$, (any a)	(no output)

Discussion (#2)

- **Concrete engine**: runs the program on actual data.
  Symbolic engines: tracks the execution ro build formulas for the path conditions.

- **Concolic executor** feeds new inputs from the solver to the concrete runner.
- **Symbolic constraints** are used to systematically cover *unexplored* branches.

- Actual toolkits
- **DART** (Directed Automated Random Testing, Godefroid et al., PLDI 2005),
- **CUTE** (Sen et al., FSE 2005),
- **SAGE** (Microsoft fuzzing platform),
- **KLEE** (for LLVM),

Doscussion (#3)

- Still needs heuristics to decide *which branch* to flip
- Loops with symbolic bounds can still cause huge state spaces.
- Constraint solving can still be expensive (e.g. with non-linear terms).
- Handling concurrency and I/O is hard because the concrete environment affects symbolic tracking.

Final remarks

Solver-supported, whitebox testing

- Reason symbolically about (parts of) inputs
- Create new inputs that cover not yet explored paths
- More systematic but also more expensive than random and fuzz testing
- Open challenges
  - Effective exploration of huge search space
  - Other applications of constraint-based program analysis, e.g., debugging and automated program repair