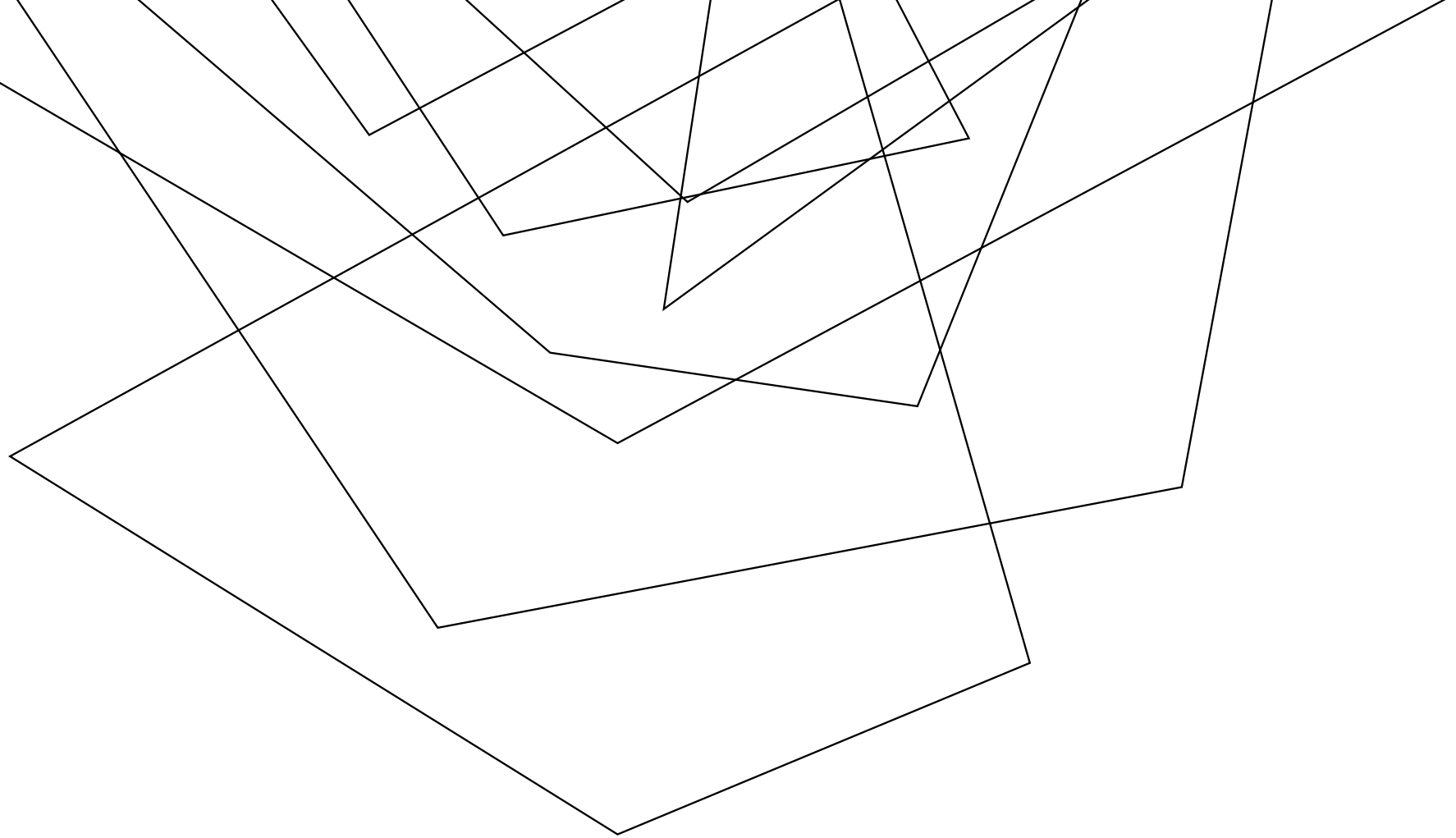
EXERCISE

DYNAMIC ANALYSIS REVIEW

Write your name and answer the following on a piece of paper

*In fuzzing, it is easy to generate additional test cases for an analysis target. What are some of the strategies for **prioritizing** which test case to run next?*

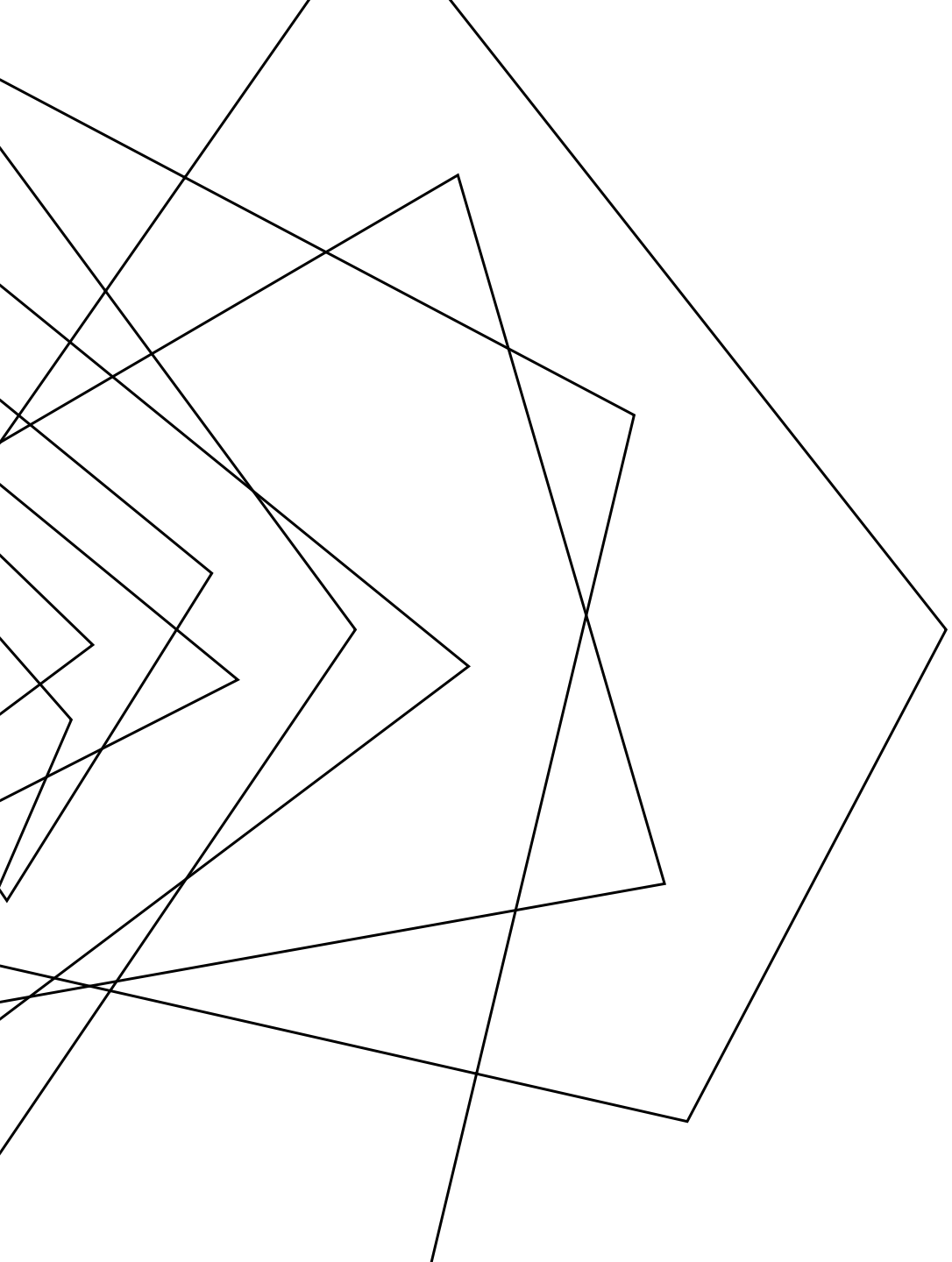
ADMINISTRIVIA AND ANNOUNCEMENTS



SYMBOLIC EXECUTION

EECS 677: Software Security Evaluation

Drew Davidson



WHERE WE’RE AT

DYNAMIC ANALYSIS

Generating test cases

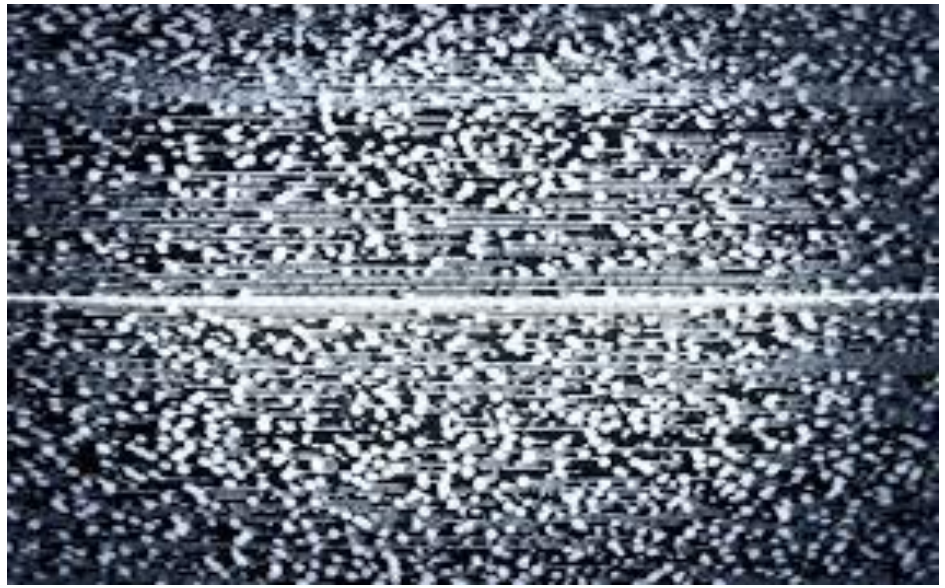
PREVIOUSLY: FUZZING

OUTLINE / OVERVIEW

GENERATING RANDOM TEST CASES

Surprisingly effective in practice

Main challenge is exploring “new” behavior



The random “fuzz” of white noise

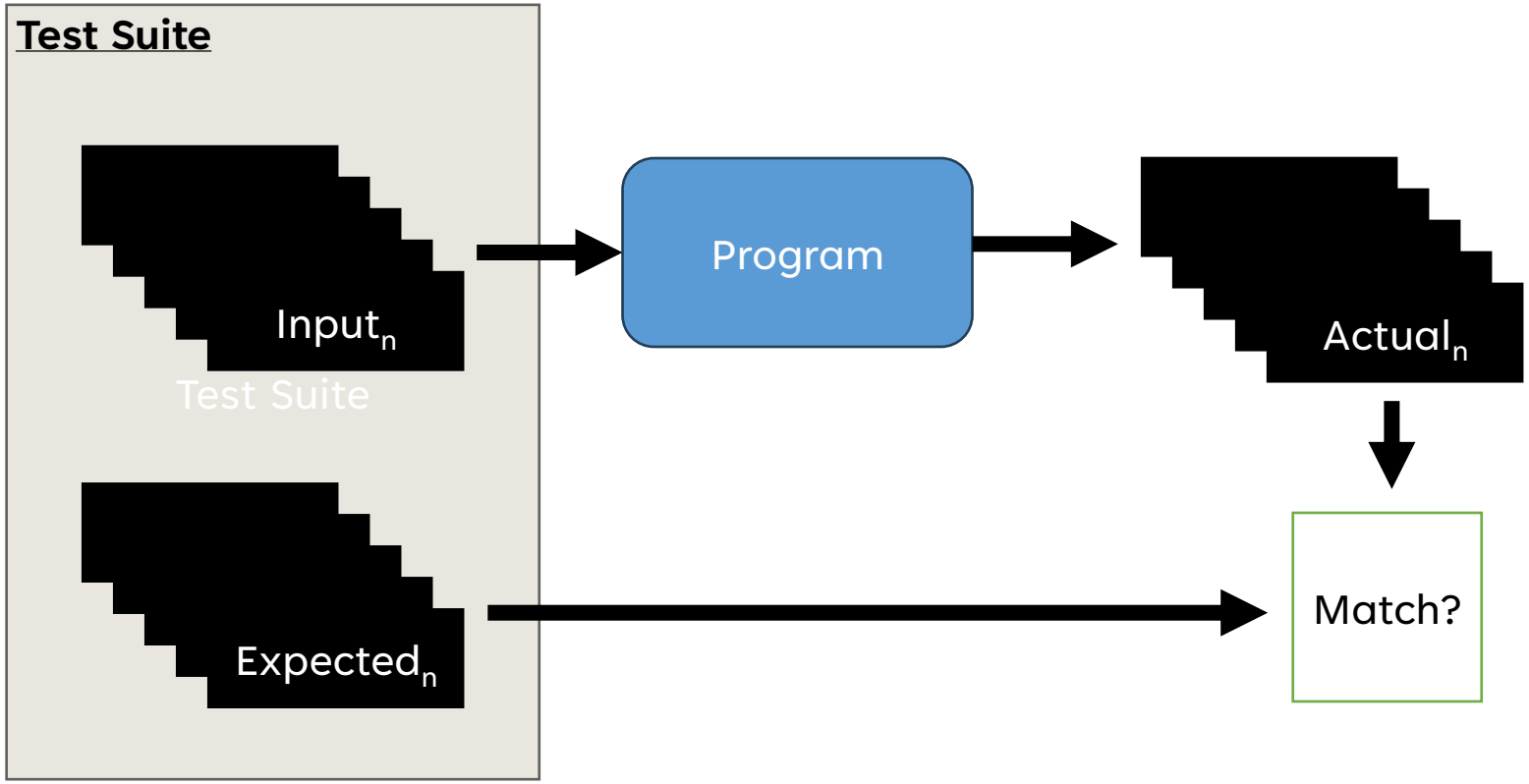
THIS LECTURE: SYMBOLIC EXECUTION

OUTLINE / OVERVIEW

A METHODICAL APPROACH TO “ABSTRACT” EXECUTION

RECALL: TEST CASE GENERATION

SYMBOLIC EXECUTION



THE PROBLEM OF COVERAGE

SYMBOLIC EXECUTION

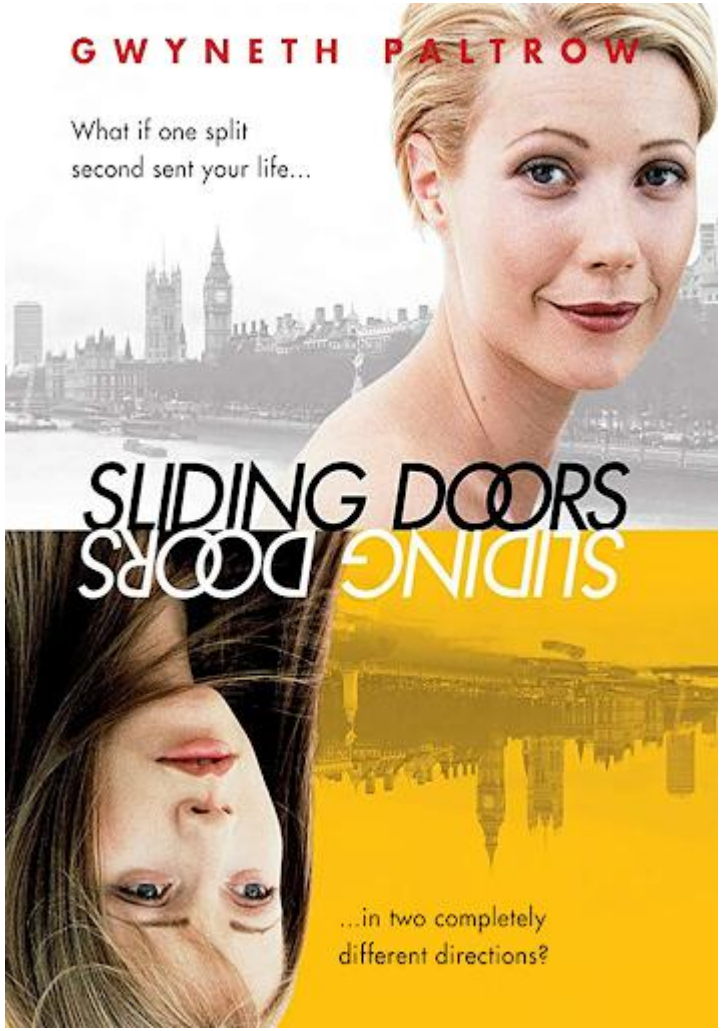
```
1: #include “stdlib.h”
2: int main(){
3:   int c = getchar();
4:   if (c == 12345){
5:     blowUpTheOcean();
6:   } else {
7:    return 0;
8:   }
9: }
```

A divergence, beyond which some behavior will be missed

BIG IDEA: EXPLORE BOTH OUTCOMES

SYMBOLIC EXECUTION

```
1: #include “stdlib.h”
2: int main(){
3:    int c = getchar();
4:    if (c == 12345){
5:      blowUpTheOcean();
6:    } else {
7:     return 0;
8:    }
9: }
```

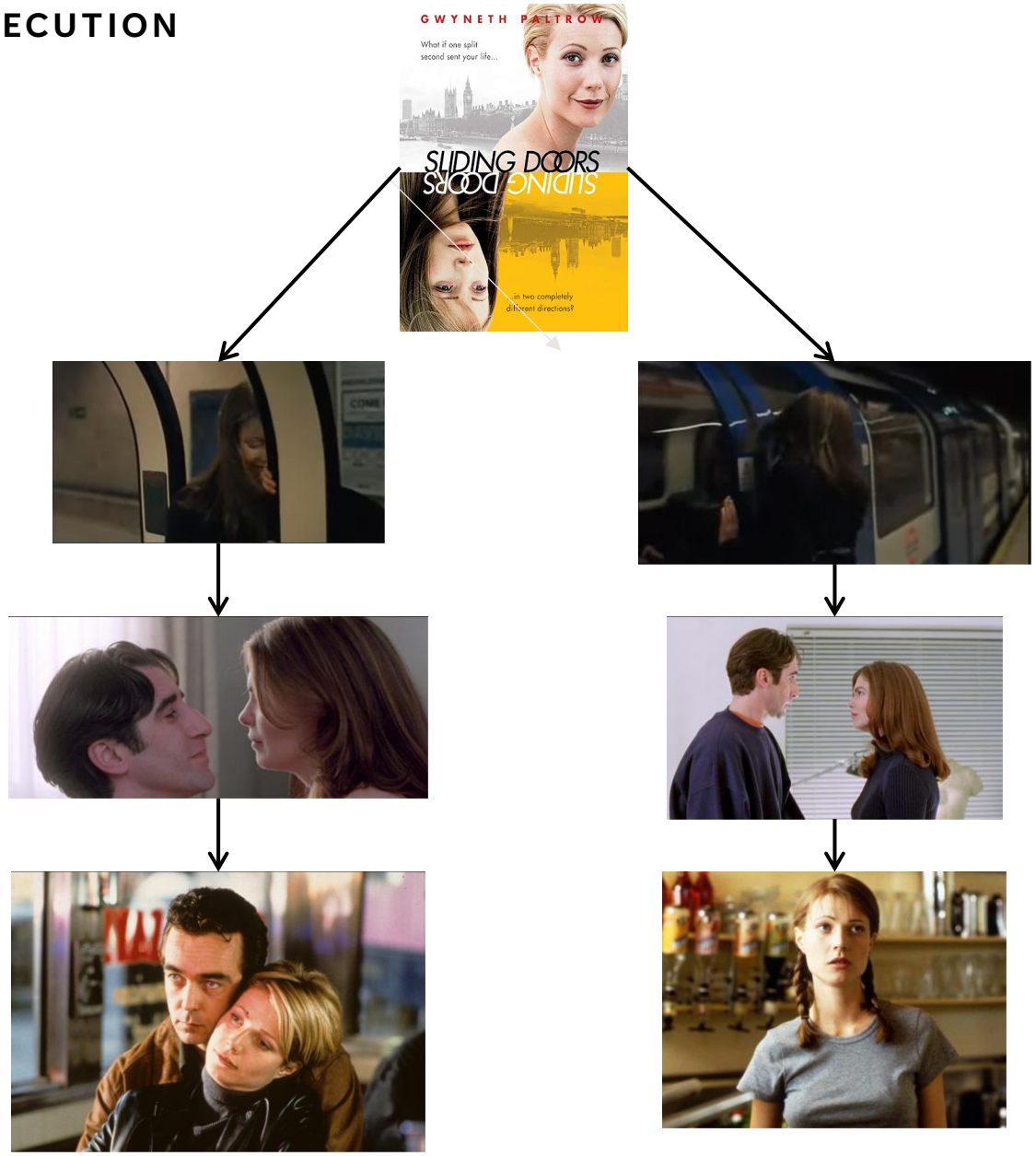


BIG IDEA: EXPLORE BOTH OUTCOMES

SYMBOLIC EXECUTION

```
goodBoyfriend = NULL;

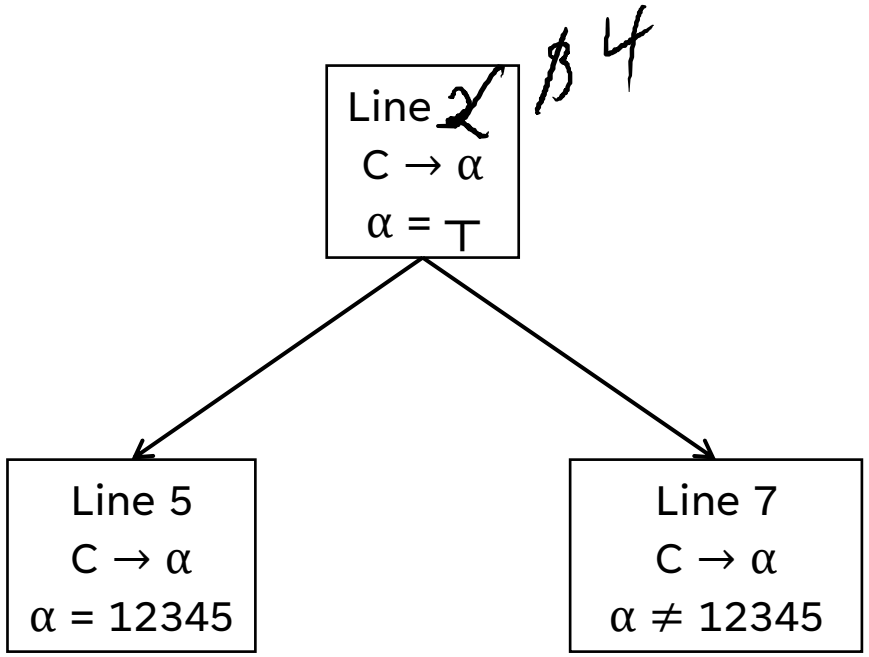
if (sheMakesTheSubway){
   goodBoyfriend->surprise();
   haircut = true;
   goodBoyfriend = new Boyfriend()
}
else {
  haircut = false;
  minimumWageJob(gwyneth);
}
```



BIG IDEA #1: *SYMBOLIC* INPUT

SYMBOLIC EXECUTION

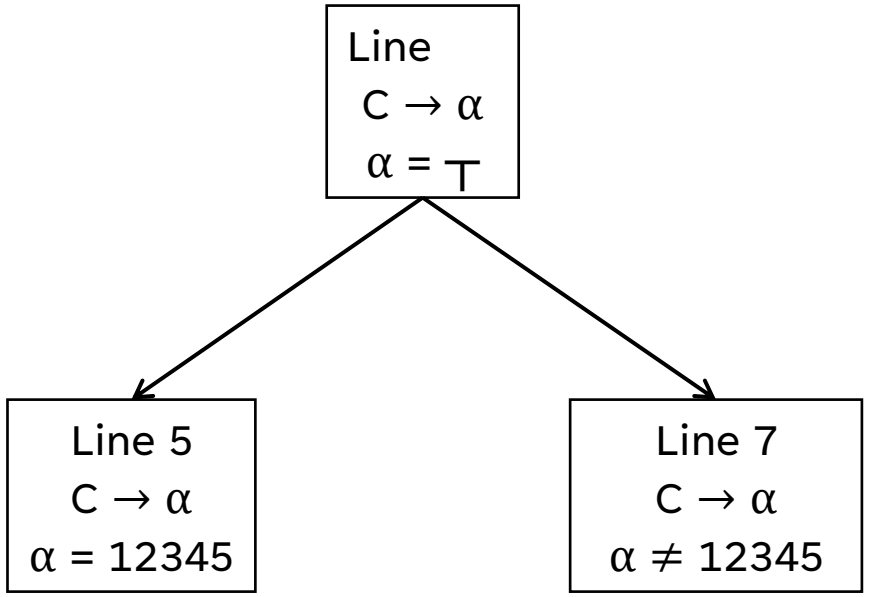
```
 1: #include “stdlib.h”
 2: int main(){
 3:   int c = getchar();
 4:   if (c == 12345){
 5:     blowUpTheOcean();
 6:   } else {
 7:     return 0;
 8:   }
 9: }
10:
```



RESPECT CONCRETE SEMANTICS

SYMBOLIC EXECUTION

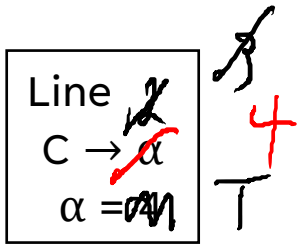
```
 1: #include “stdlib.h”
 2: int main(){
 3:   int c = getchar();
 4:   if (c == 12345){
 5:     blowUpTheOcean();
 6:   } else {
 7:     return 0;
 8:   }
 9: }
10:
```



RESPECT CONCRETE SEMANTICS

SYMBOLIC EXECUTION

```
 1: #include “stdlib.h”
 2: int main(){
 3:    int c = getchar();
 4:    c = 4;
 5:    if (c == 12345){
 6:       blowUpTheOcean();
 7:    } else {
 8:       return 0;
 9:    }
10: }
```



PATH FEASIBILITY

SYMBOLIC EXECUTION

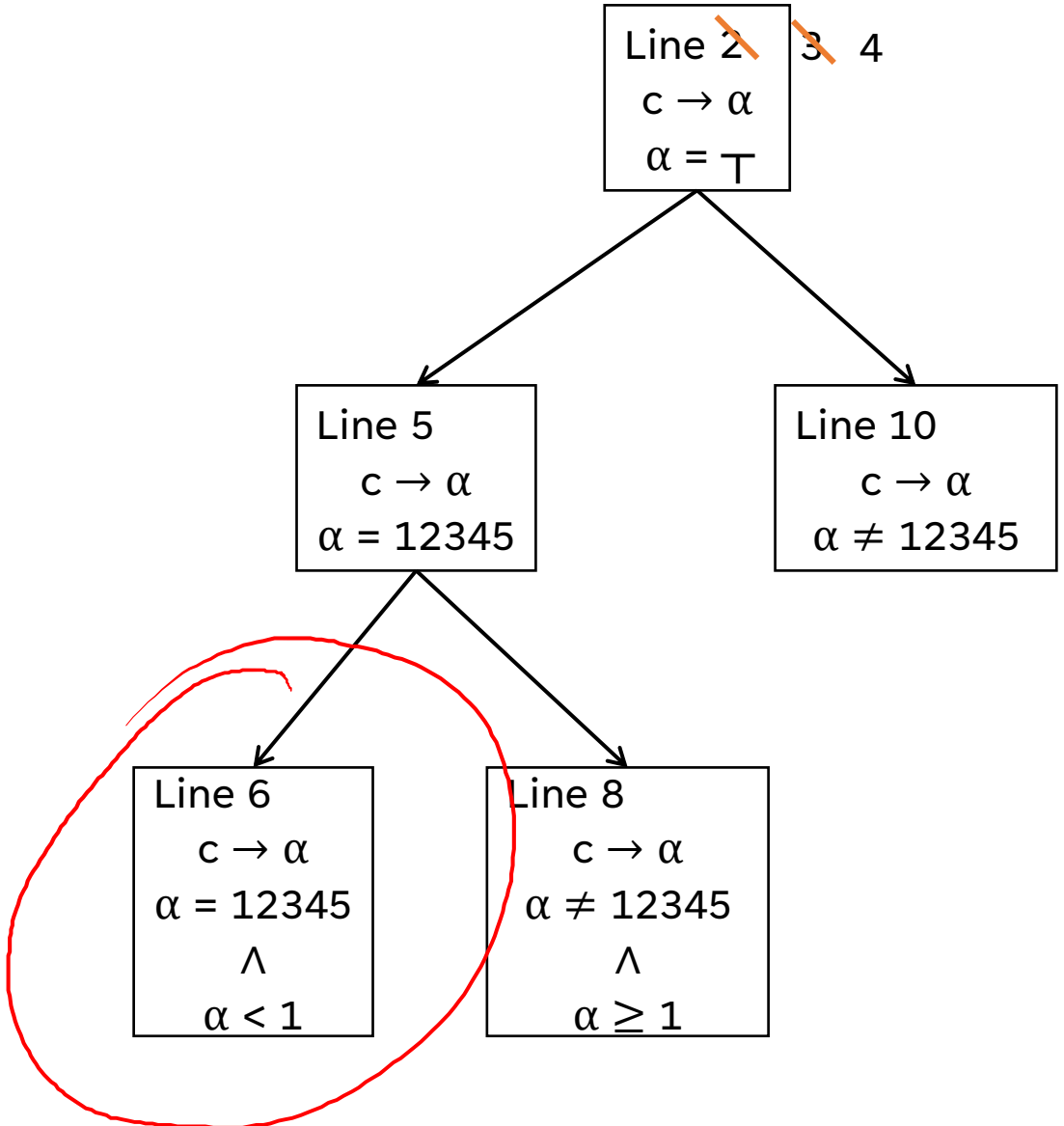
```
 1: #include “stdlib.h”
 2: int main(){
 3:   int c = getchar();
 4:   if (c == 12345){
 5:     if (c < 1){
 6:       blowUpTheOcean();
 7:     }
 8 :  } else {
10:     return 0;
11:   }
12: }
```

Even with symbolic variables,
it may be impossible to exercise both branches

PATH FEASIBILITY

SYMBOLIC EXECUTION

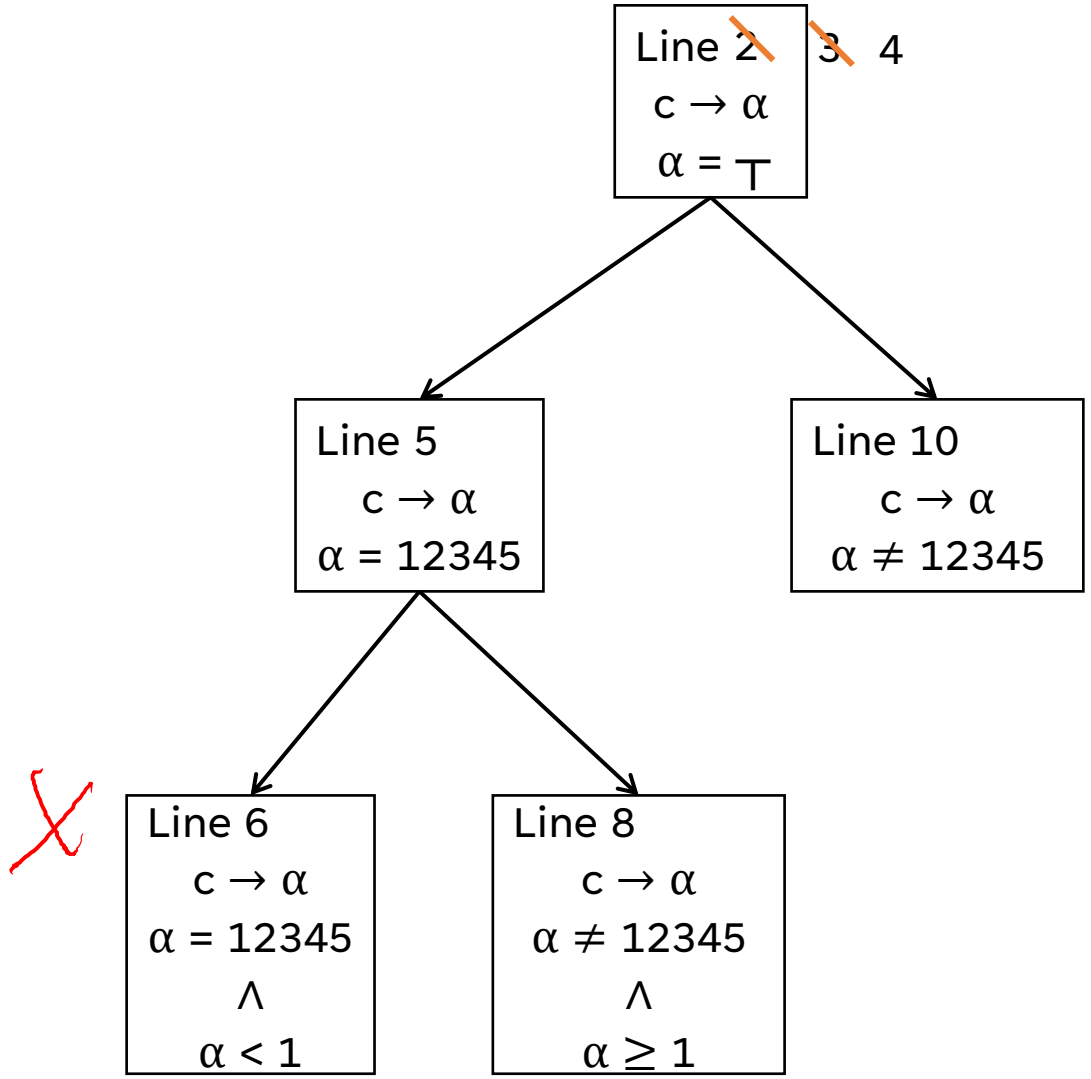
```
 1: #include “stdlib.h”
 2: int main(){
 3:    int c = getchar();
 4:    if (c == 12345){
 5:       if (c < 1){
 6:          blowUpTheOcean();
 7:       }
 8 :   } else {
10:       return 0;
11:    }
12: }
```



BIG IDEA #2: PATH CONSTRAINTS

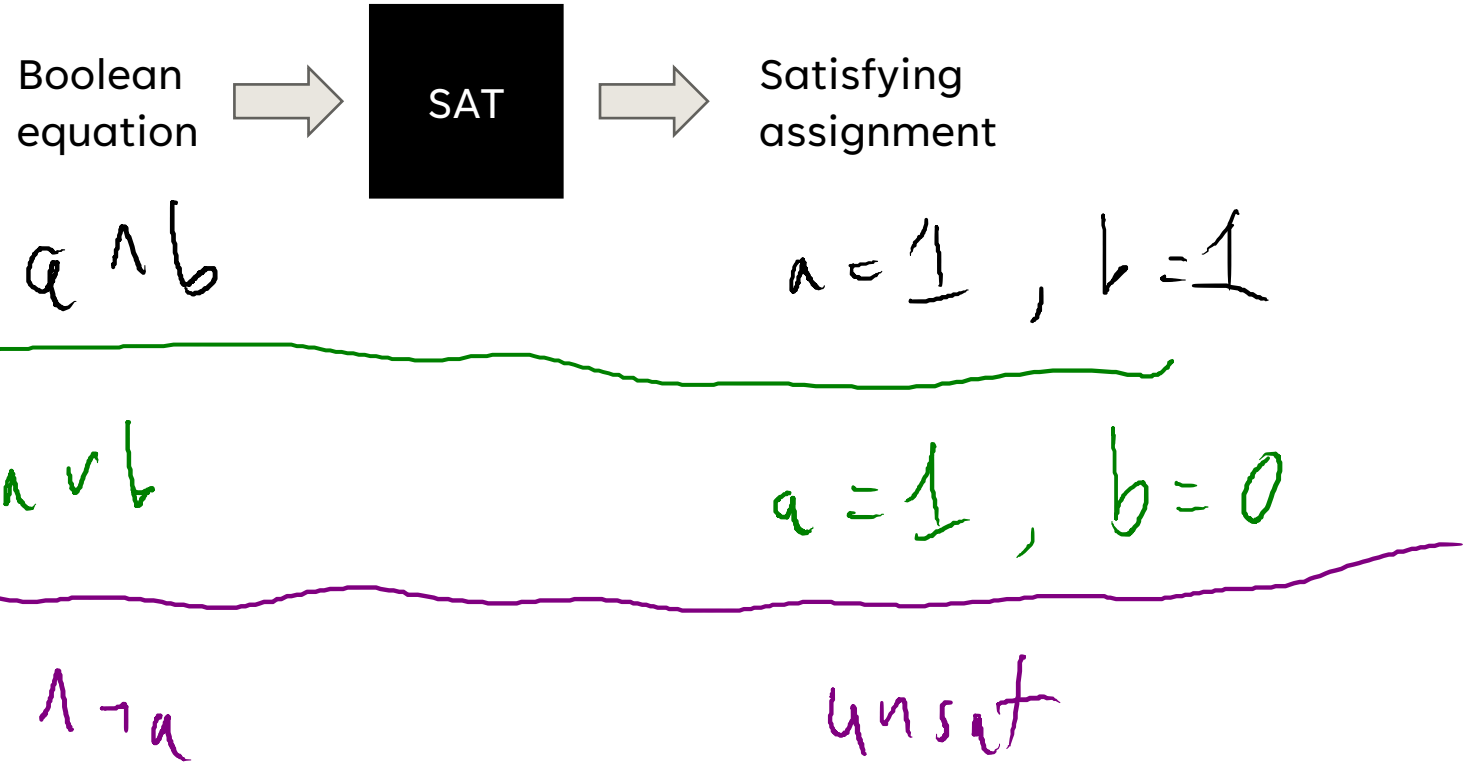
SYMBOLIC EXECUTION

```
 1: #include “stdlib.h”
 2: int main(){
 3:   int c = getchar();
 4:   if (c == 12345){
 5:     if (c < 1){
 6:       blowUpTheOcean();
 7:     }
 8 :  } else {
10:     return 0;
11:   }
12: }
```



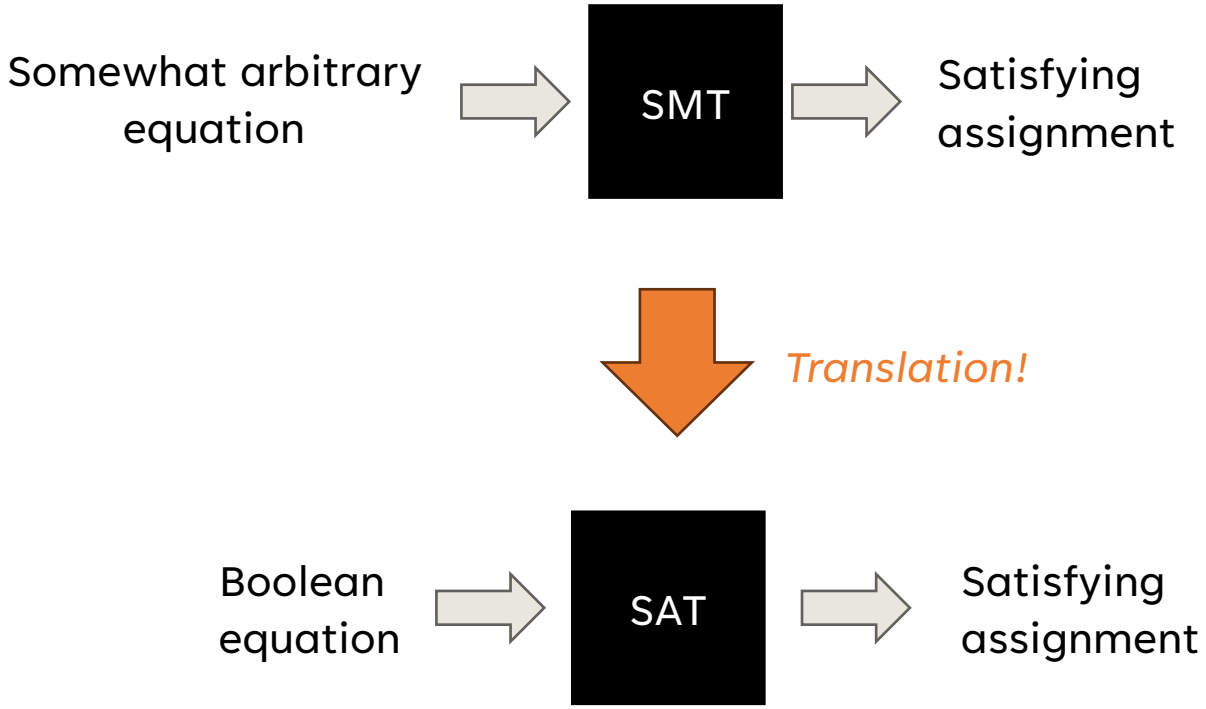
THE MAGIC OF THE SOLVER

SYMBOLIC EXECUTION



THE MAGIC OF THE SOLVER

SYMBOLIC EXECUTION



THE SYMBOLIC EXECUTION TREE

SYMBOLIC EXECUTION

```
 1: #include “stdlib.h”
 2: int main(){
 3:   int c = getchar();
 4:   if (c == 12345){
 5:     if (c < 1){
 6:       blowUpTheOcean();
 7:     }
 8 :  } else {
10:     return 0;
11:   }
12: }
```

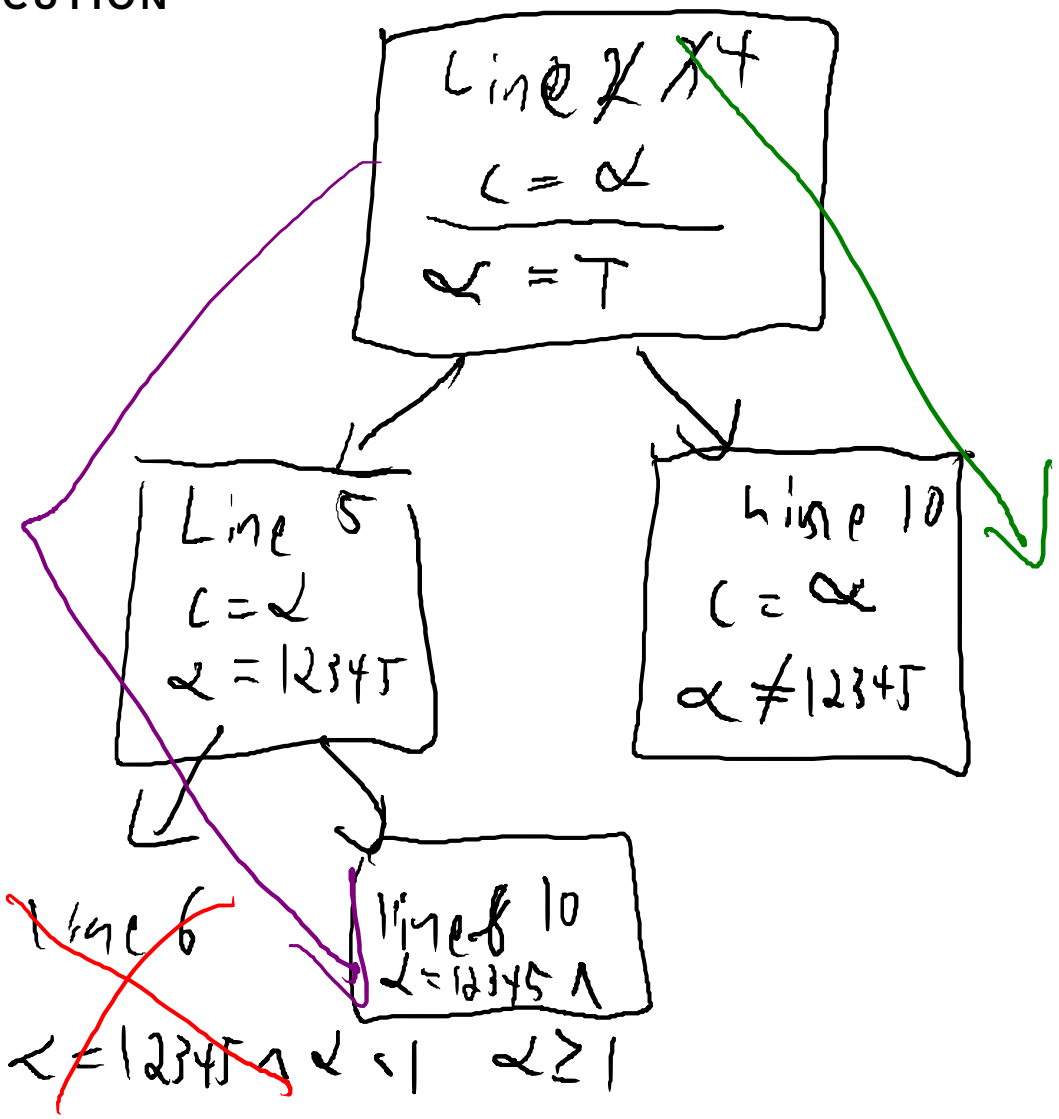
At each line of the program:

- Advance a symbolic program state
- When you hit a branch split the symbolic state into 2 versions
  - 1) a state that satisfies the predicate
  - 2) a state that does not satisfy the predicate

TEST SUITE GENERATION

SYMBOLIC EXECUTION

```
 1: #include “stdlib.h”
 2: int main(){
 3:    int c = getchar();
 4:    if (c == 12345){
 5:       if (c < 1){
 6:          blowUpTheOcean();
 7:       }
 8 :   } else {
10:       return 0;
11:    }
12: }
```



SOUNDNESS / COMPLETENESS

SYMBOLIC EXECUTION

Soundness



Never generate a state that is unreachable

Completeness



Never miss a state



Termination

Maybe not!

```
1. int main(){
2.  int c = getchar();
3.  while ( c ){
4.     if (c == 12345){
5.         blowUpTheOcean();
6.     }
7.   }
8.   return 4;
9. }
```

SOUNDNESS / COMPLETENESS

SYMBOLIC EXECUTION

```
1. int main(){
2.  int c = getchar();
3.  while ( c ){
4.     if (c == 12345){
5.         blowUpTheOcean();
6.     }
7.   }
8.   return 4;
9. }
```

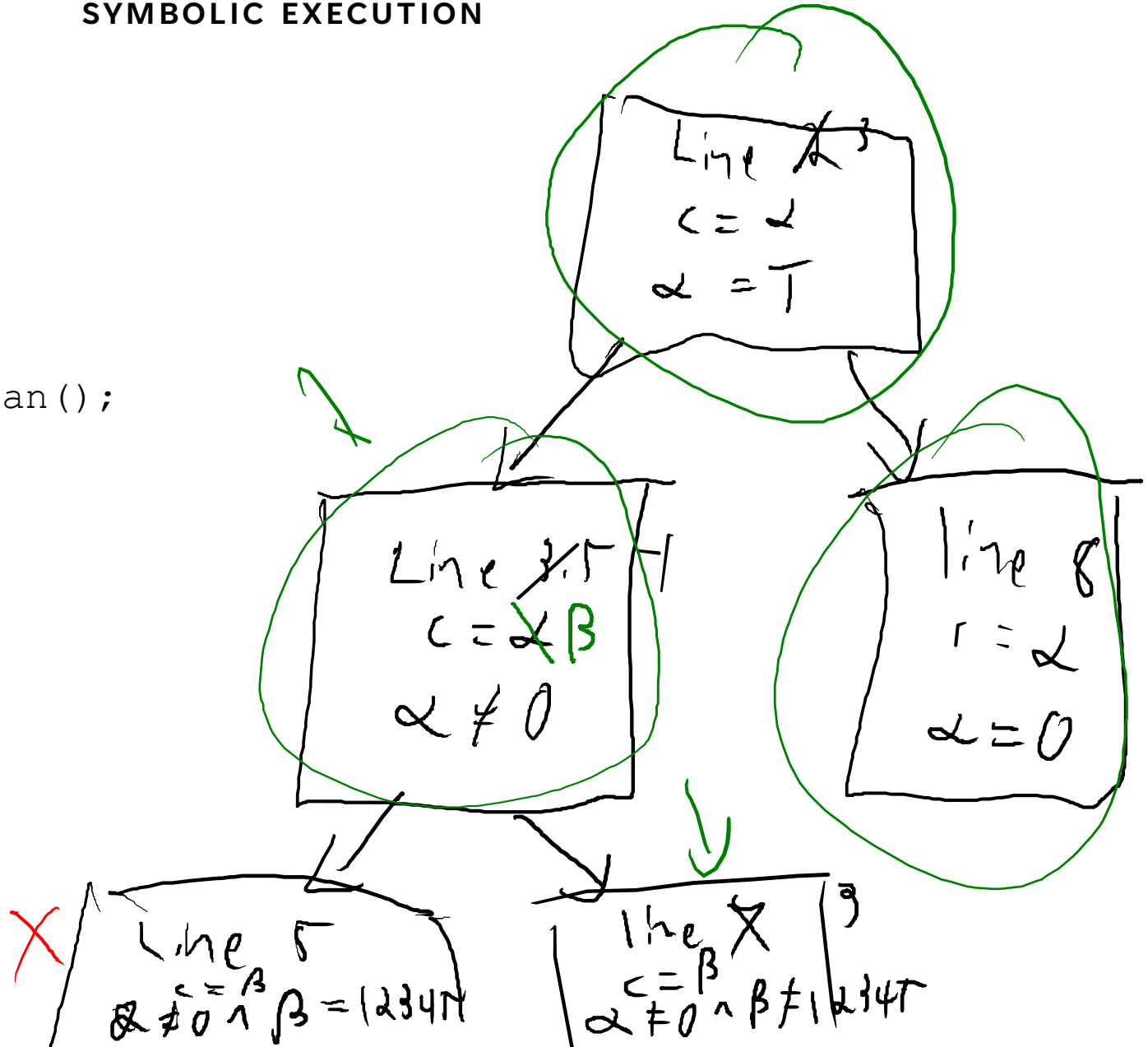
Termination

Maybe not!

LIMITATION: COVERAGE

SYMBOLIC EXECUTION

```
 1. int main(){
 2.     int c = getchar();
 3.     while ( c ){
3.5.      c = getchar();
 4.       if (c == 12345){
 5.           blowUpTheOcean();
 6.       }
 7.     }
 8.     return 4;
 9. }
```



LIMITATION: ENVIRONMENT INPUT

SYMBOLIC EXECUTION

```
1. int main(){
2.    int c = getchar();
3.    c = ioctl(...);
4.    return 4;
5. }
```

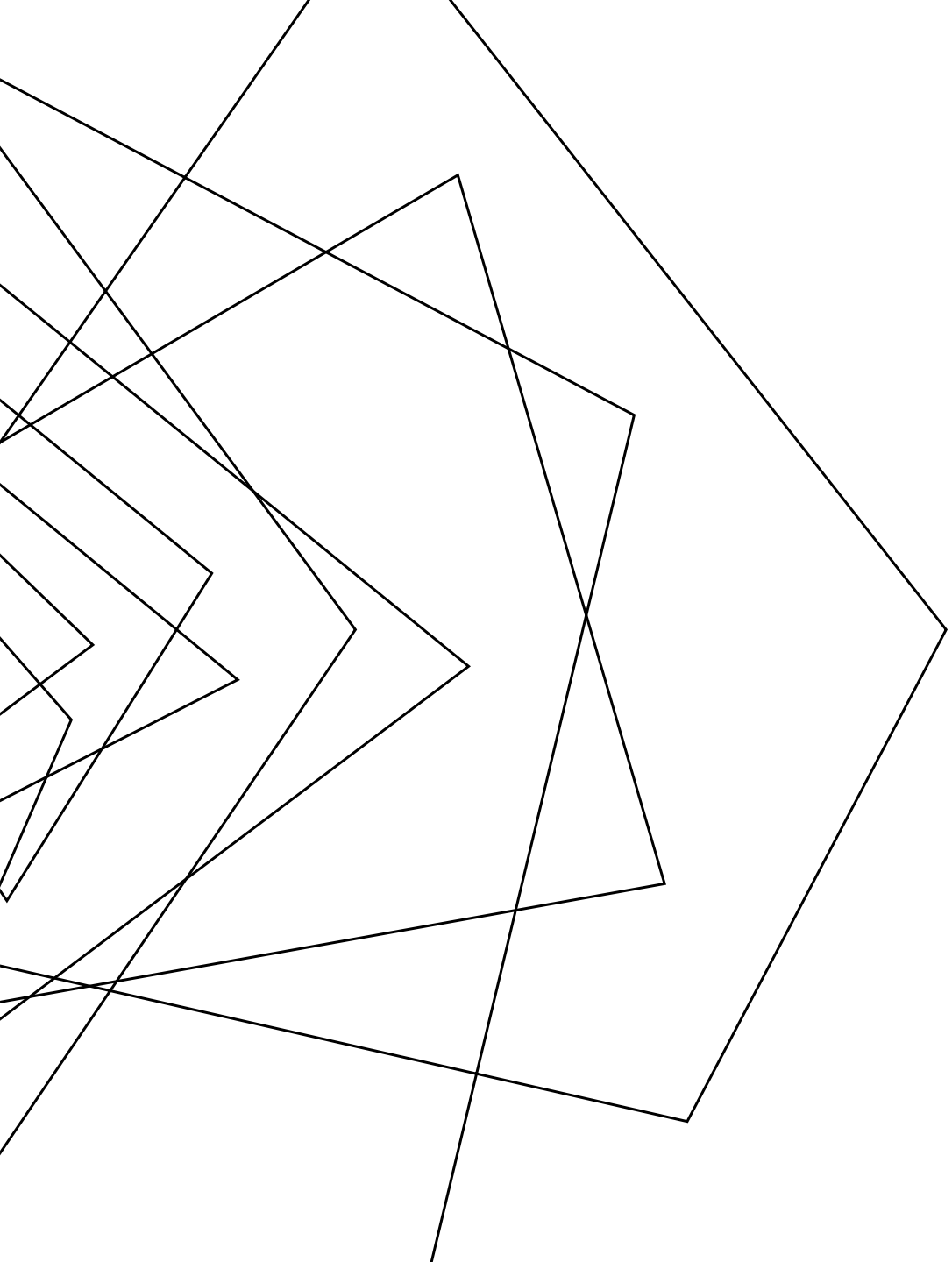
c

a = time()

b = time()

if (a < b) {

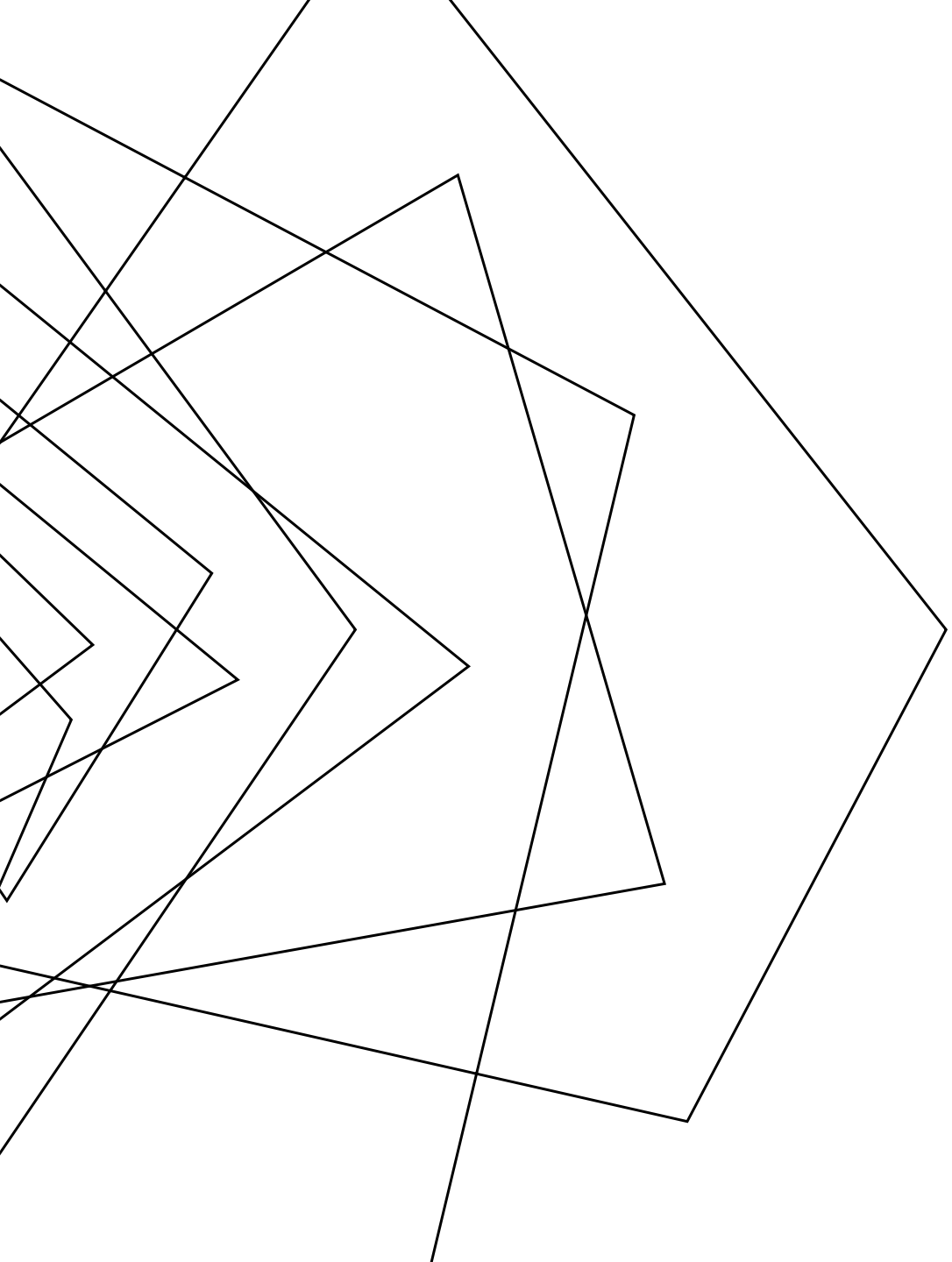
}



SUMMARY

SYMBOLIC EXECUTION

A simple, elegant idea



NEXT TIME

CONCOLIC EXECUTION

An extension to symbolic execution

Give up on completeness AND soundness to increase practical coverage

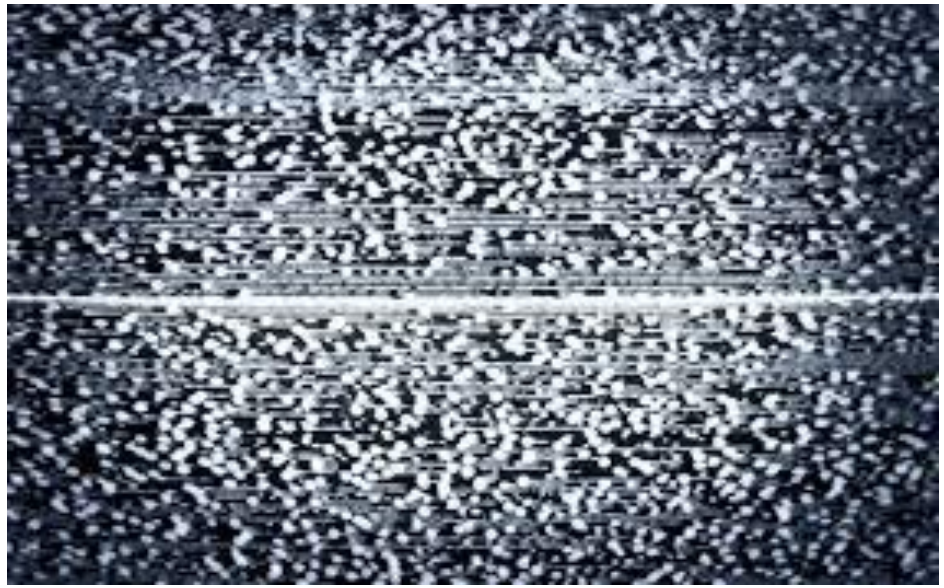
PREVIOUSLY: FUZZING

OUTLINE / OVERVIEW

GENERATING RANDOM TEST CASES

Surprisingly effective in practice

Main challenge is exploring “new” behavior



The random “fuzz” of white noise

THIS LECTURE: SYMBOLIC EXECUTION

OUTLINE / OVERVIEW

A METHODICAL APPROACH TO “ABSTRACT” EXECUTION

RECALL: TEST CASE GENERATION

SYMBOLIC EXECUTION

THE PROBLEM OF COVERAGE

SYMBOLIC EXECUTION

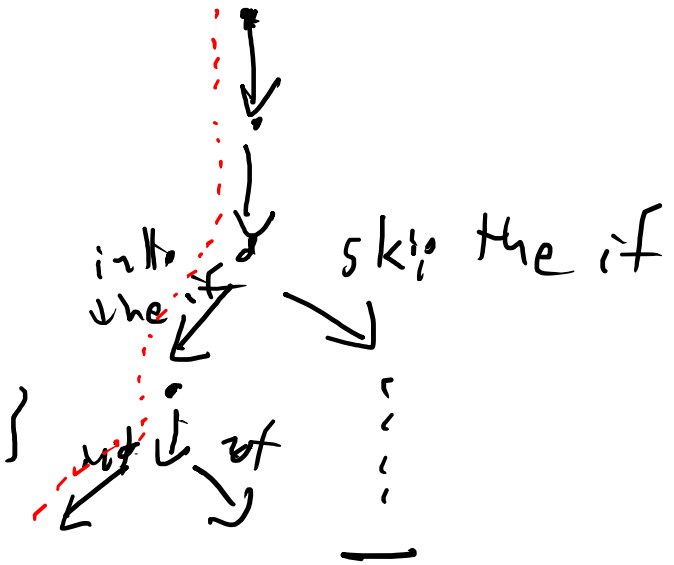
PREDICATES GET IN THE WAY!

SYMBOLIC EXECUTION

```
#include "stdlib.h"

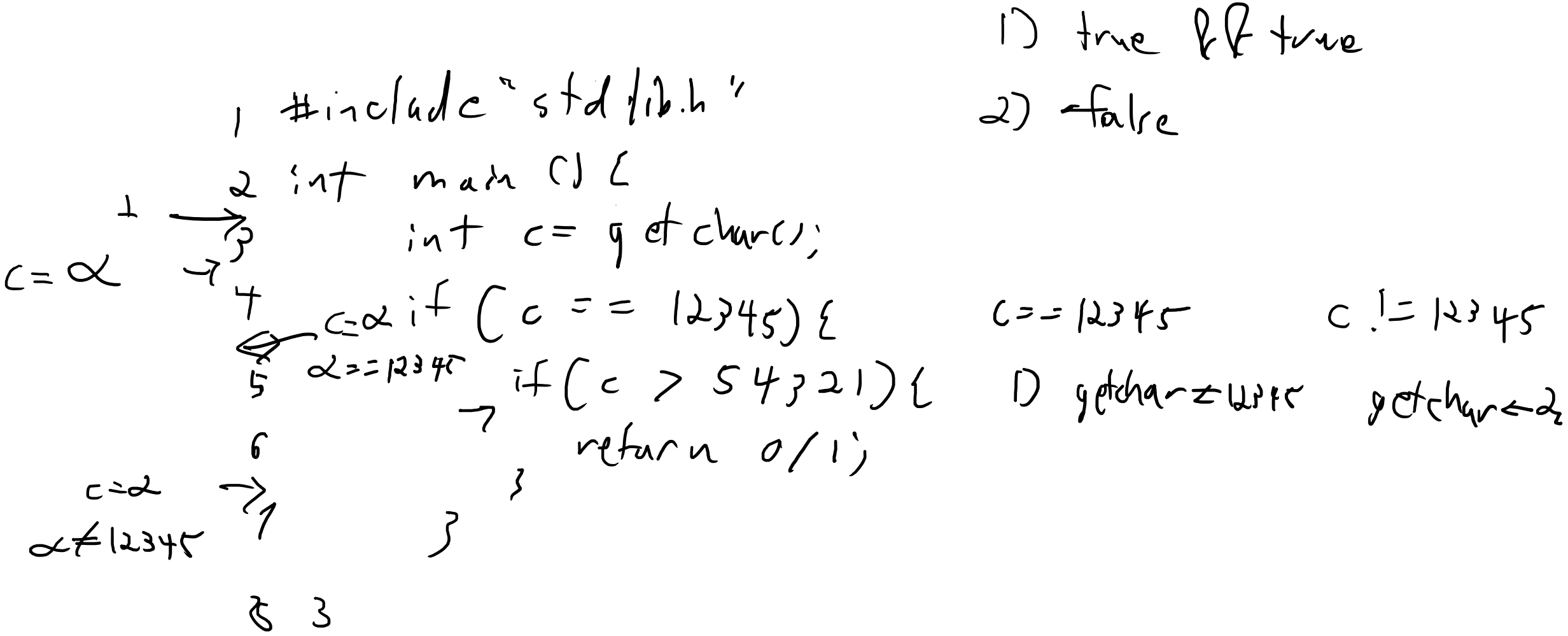
int main() {
    int c = getchar();
    if (c == 12345){
        c=getchar();
        if(c==54321){ return 1/0;}
    }

}
```



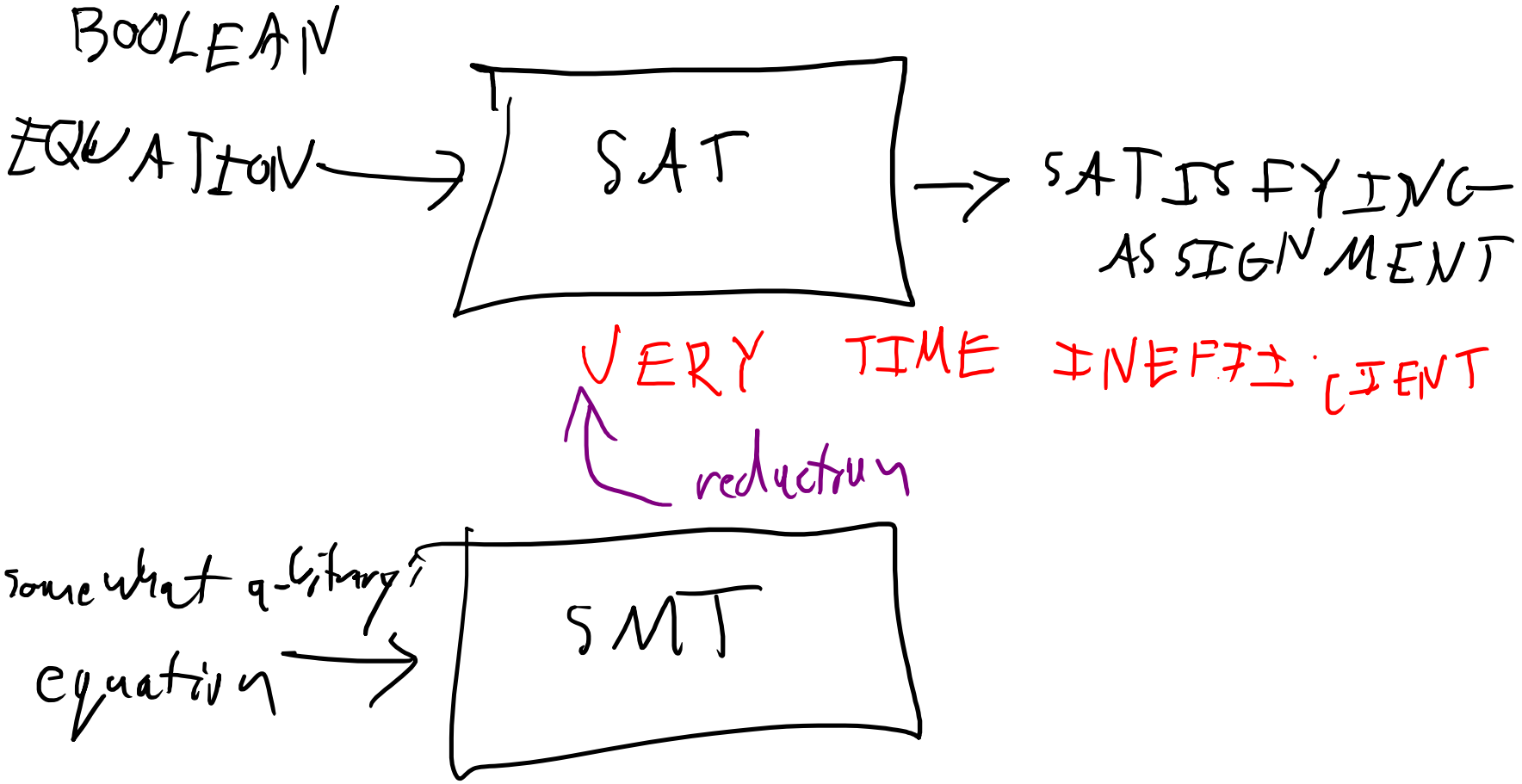
ELIMINATING INFEASIBLE PATHS

SYMBOLIC EXECUTION



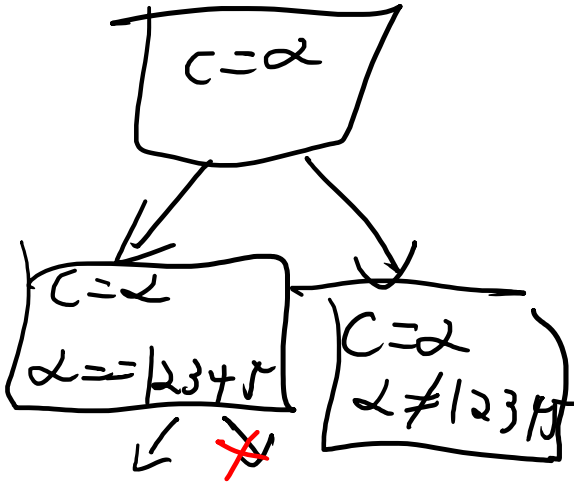
THE MAGIC OF THE SOLVER

SYMBOLIC EXECUTION



THE SYMBOLIC EXECUTION TREE

SYMBOLIC EXECUTION



At each line of the program:

- advance the symbolic program state
- when you hit a branch,
  split the symbolic state into 2 versions:
  1) satisfies the branch predicate
  2) does not satisfy the branch predicate

SOUNDNESS / COMPLETENESS

SYMBOLIC EXECUTION

✓ Soundness:

- Never generate a state that violates the constraints

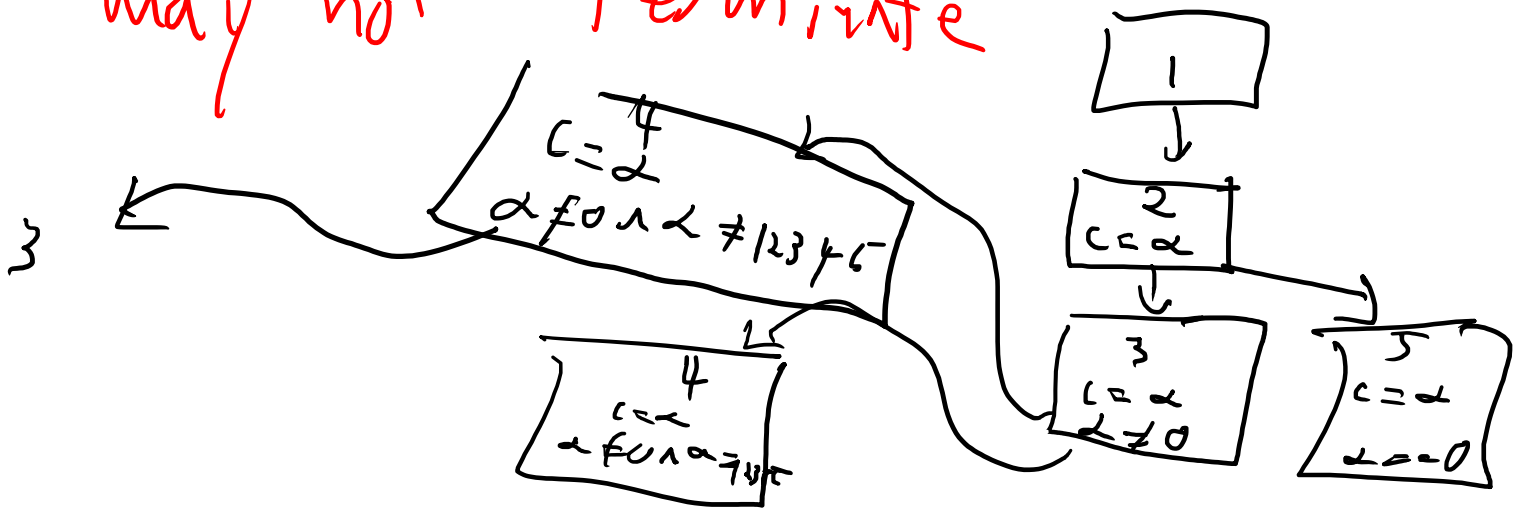
✓ Completeness:

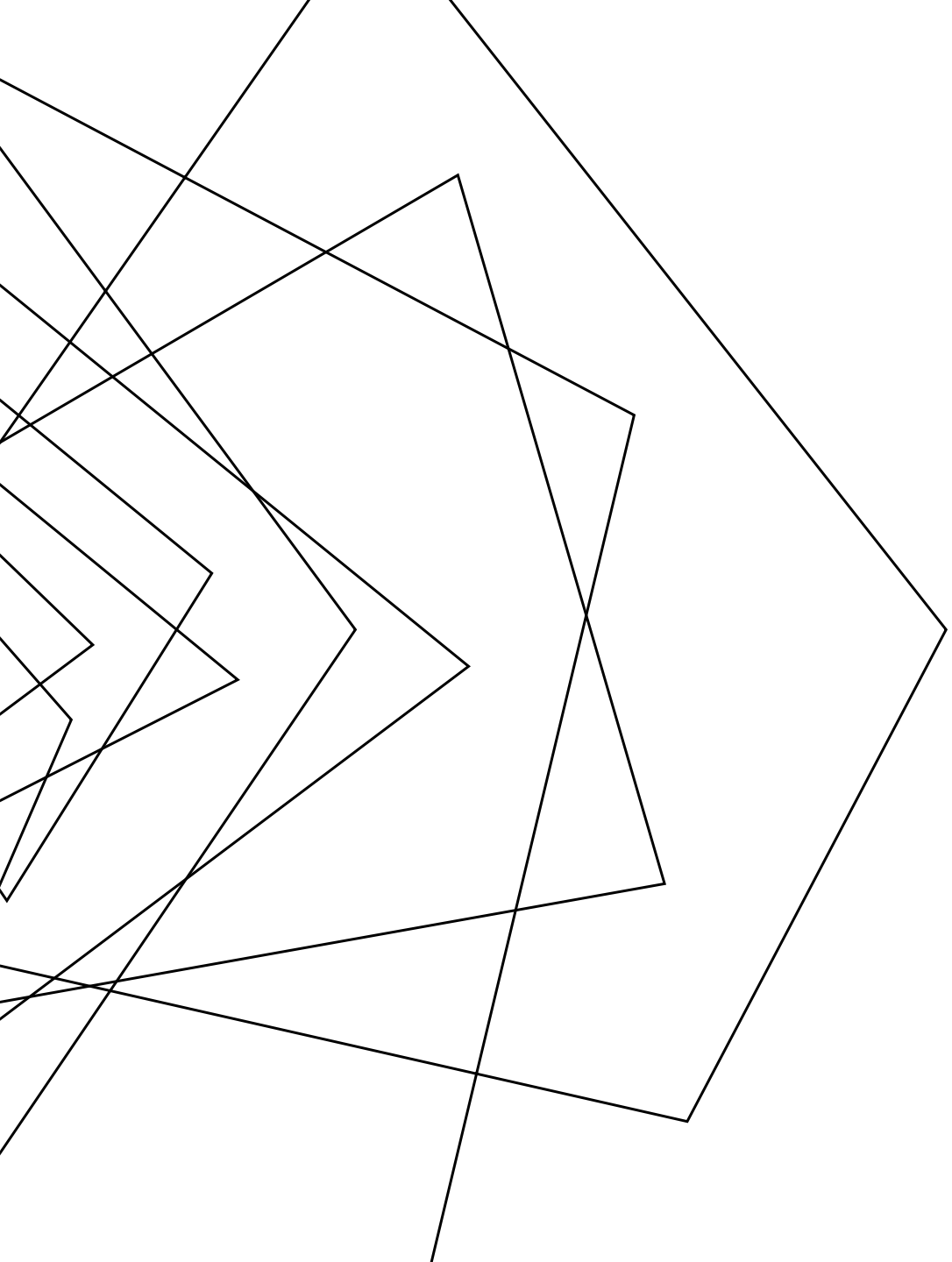
- Never a state we miss

may not terminate

```
1 int main() {
2     int c = getchar();
3     while (c ) {
4         if (c == 123+5)
5 }
```

c = getchar();





WRAP-UP

SYMBOLIC EXECUTION

A simple, elegant idea