EXERCISE #7

COMPUTABILITY REVIEW

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

Consider a simple bug-finding analysis that looks for null pointer deferences in C programs. The analysis raises an alert on any program that has ANY pointer operation, and does not raise an alert on any other program.
Is this analysis sound, complete, neither, or both? Justify your answer.

STATIC ANALYSIS

EECS 677: Software Security Evaluation

Drew Davidson

ADMINISTRIVIA AND ANNOUNCEMENTS

H1 - past due

H2 - coming out

LAST TIME: ANALYSIS DEFINITIONS

REVIEW: COMPUTABILITY

Analysis Target – The system being analyzed

- For us this will usually be a software program

Analysis Engine – The system doing analysis

- For us this will usually be a software program

Analysis Goal – The phenomenon we are detecting

- The existence of a certain (program) behavior?
- The absence of a certain (program) behavior?

LAST TIME: ANALYSIS LIMITS

REVIEW: COMPUTABILITY

The limits of computability

- The Halting Problem: No decision procedure for halting
- Rice’s Theorem: The Halting Problem implies no decision procedure for any reachability problem

Analysis without decision procedures

- Approximation
- How do we approximate? Soundness / Completeness

LAST TIME: ANALYSIS DESIGN

REVIEW: COMPUTABILITY

NO analysis can be both sound and complete

Building an analysis that is <u>either</u> sound <u>or</u> complete is trivial

- Complete – Always report positive, no false negatives
- Sound – Always report negative, no false positives

THIS TIME: WORKING WITH CONSTRAINTS

TODAY'S LECTURE

Given the limitations of analysis, how might we still provide useful tools for software security evaluation?

Static Analysis, get it?

LECTURE OUTLINE

- Contextualizing Rice’s Theorem
- Program Guarantees
- Analysis Specificity
- Dataflow analysis

RICE’S THEOREM (NON)ASSERTIONS

CONTEXTUALIZING RICE’S THEOREM

Excludes properties that are true or false for **every** program

Excludes properties that do not regard program's **behavior**

All non-trivial semantic properties of programs are undecidable.

Notably, the theorem ignores **syntactic** properties

j/k: Rice’s Theorem is named after Dr. Henry Gordon Rice

SYNTACTIC STATIC ANALYSIS

CONTEXTUALIZING RICE'S THEOREM

Some troubling behavior of a program may be discoverable via simply observing its text

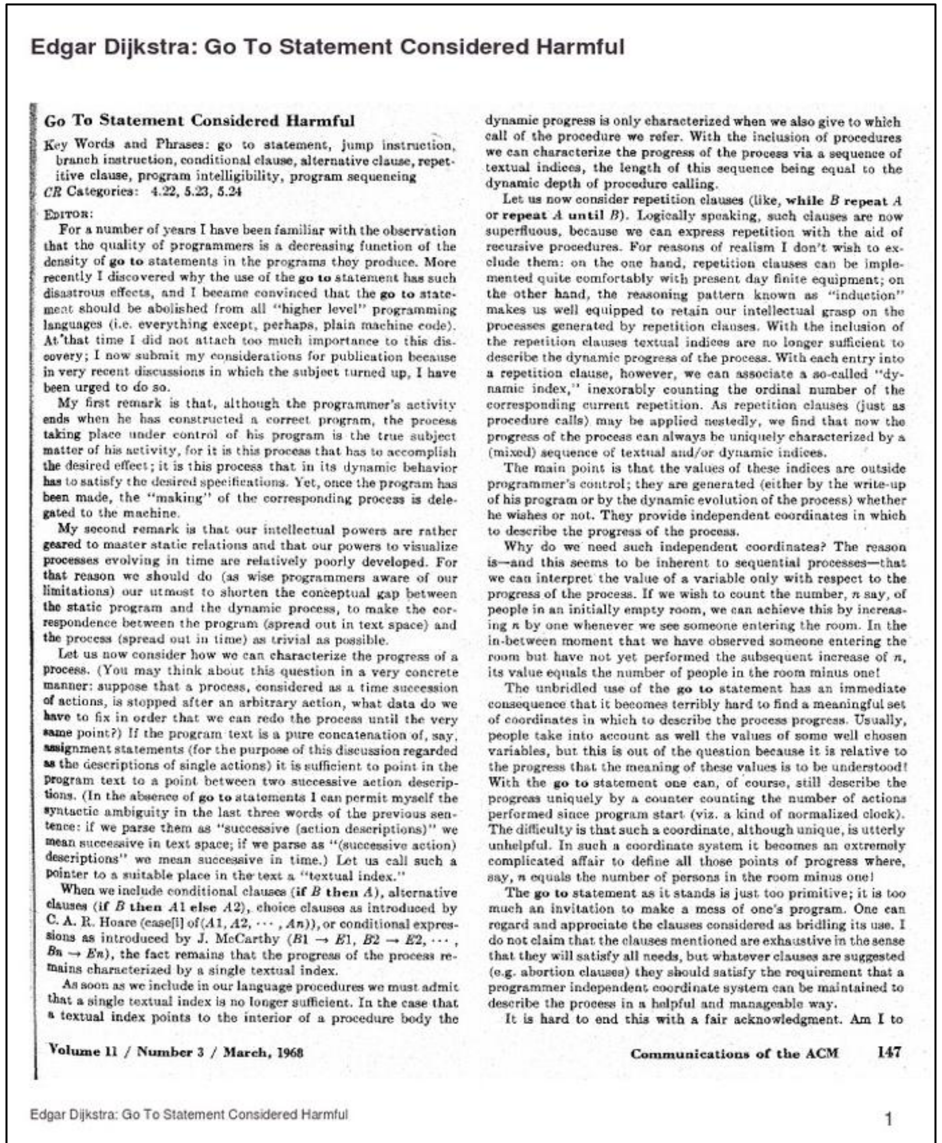
```
int main(int argc, const char * argv[]){
  const char * password = argv[1];
  int good;
  if (password == "supersecret"){
       good = 1;
  } else {
       good = 0;
  }
  authenticate(good);

}
```

INSIGHTS

CONTEXTUALIZING STATIC ANALYSIS

Software engineering "code smells" / stats

- Use of the forbidden / arcane constructs
- Cyclomatic complexity
- Long functions



Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** $A1$ **else** $A2$), choice clauses as introduced by C. A. R. Hoare (case[i] of $(A1, A2, \cdots, An)$), or conditional expressions as introduced by J. McCarthy $(B1 \rightarrow E1, B2 \rightarrow E2, \cdots, Bn \rightarrow En)$, the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** B **repeat** A or **repeat** A **until** B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

Volume 11 / Number 3 / March, 1968 Communications of the ACM 147

Edgar Dijkstra: Go To Statement Considered Harmful 1

RICE'S THEOREM (NON)ASSERTIONS

CONTEXTUALIZING RICE'S THEOREM

Excludes properties that are true or false for ***every*** program

Excludes properties that do not regard program's ***behavior***

All non-trivial semantic properties of programs are undecidable.

Notably, the theorem ignores ***syntactic*** properties

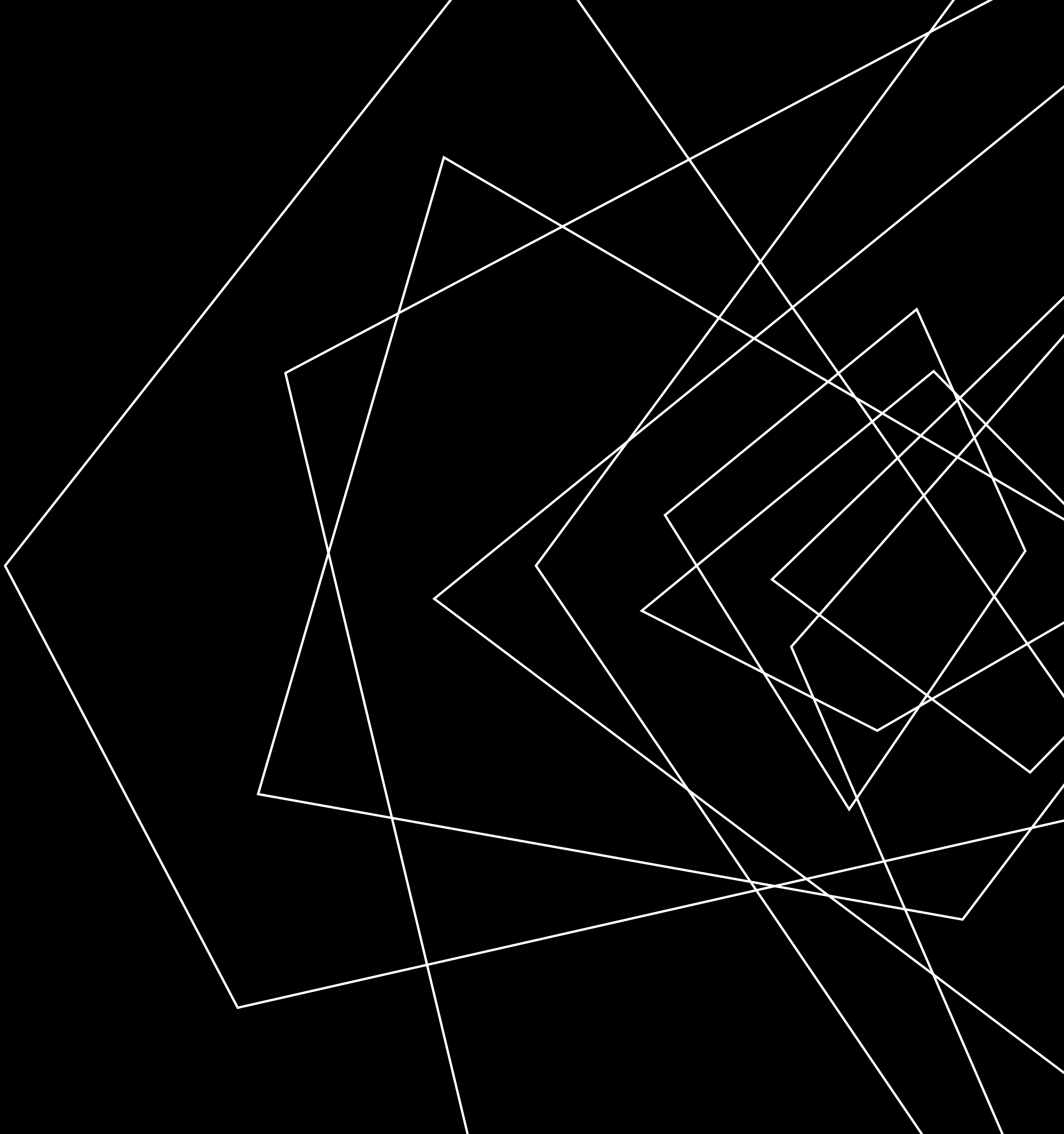
No decision procedure: that's a high bar!

We can design analyses that work perfectly in some cases, but admit uncertainty in others

j/k: Rice's Theorem is named after Dr. Henry Gordon Rice

LECTURE OUTLINE

- Contextualizing Rice's Theorem
- Program Guarantees
- Preciseness vs Efficiency
- Dataflow analysis



STATIC ANALYSIS – OPPORTUNITIES

STATIC ANALYSIS PHILOSOPHY

For security analysis, we want to lock out “bad” programs (even at the cost of locking out some “good” programs)

Provide assurances about what a program will NEVER or ALWAYS do

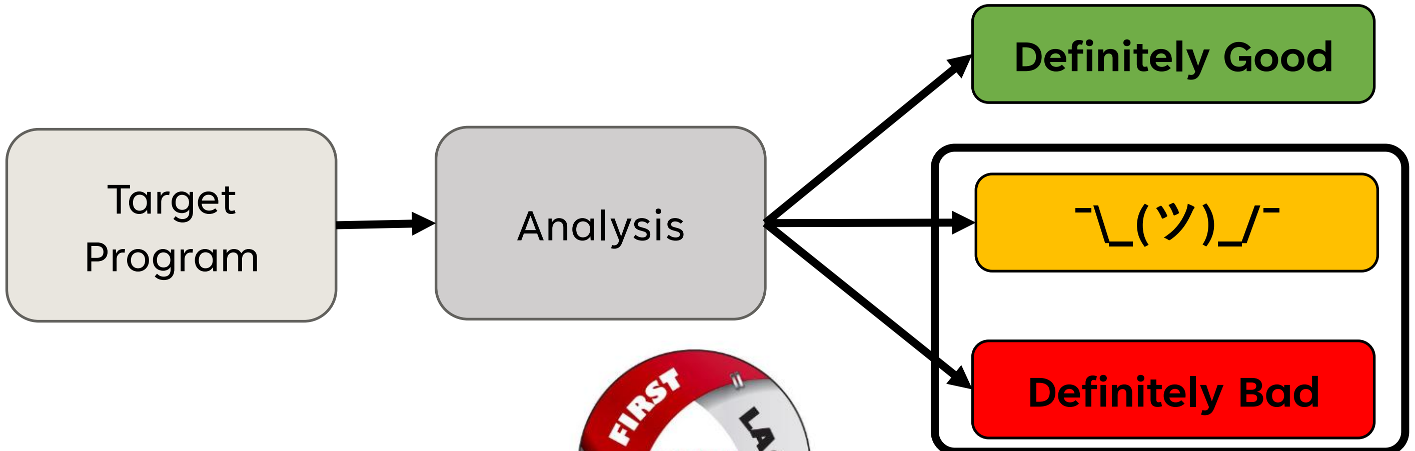
- Static analysis might report EVERY program that (possibly) has a problem
- Static analysis might certify EVERY program that (definitely) has no problem

STATIC ANALYSIS – OPPORTUNITIES

STATIC ANALYSIS PHILOSOPHY

Provide assurances about what a program will NEVER or ALWAYS do

- Static analysis might report EVERY program that (possibly) has a problem
- Static analysis might certify EVERY program that (definitely) has no problem



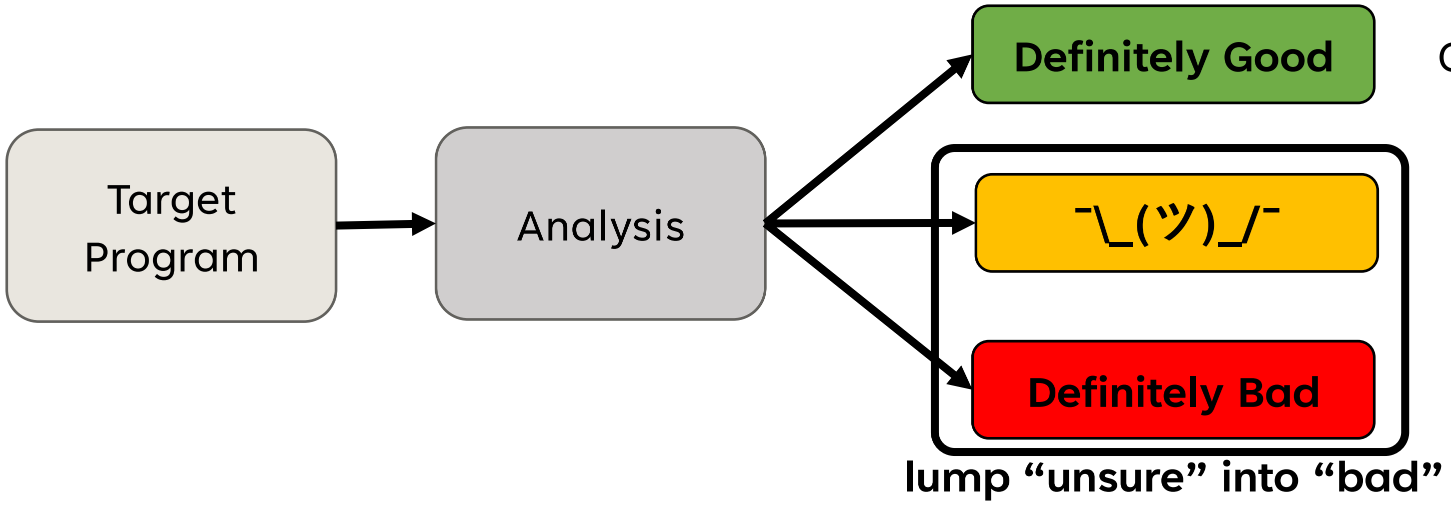
Complete bug finder – no flag

Complete bug finder - flag

lump “unsure” into “bad”

STATIC ANALYSIS - OPPORTUNITIES

STATIC ANALYSIS PHILOSOPHY



Complete bug finder – no flag

Complete bug finder - flag

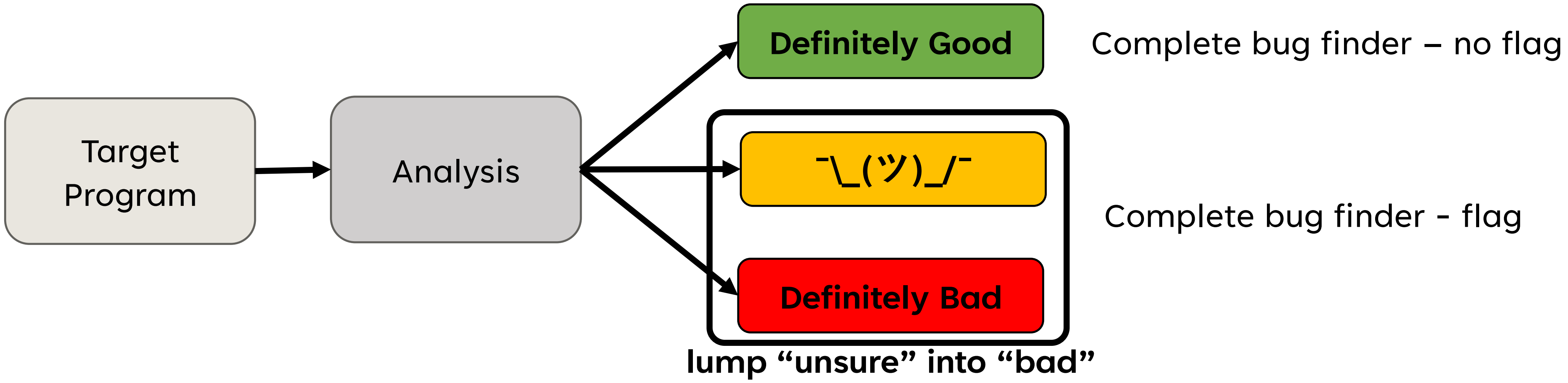
Goal: minimize the uncertainty

LECTURE OUTLINE

- Contextualizing Rice’s Theorem
- Program Guarantees
- Analysis over CFG
- Preciseness vs Efficiency

STATIC ANALYSIS - OPPORTUNITIES

STATIC ANALYSIS PHILOSOPHY



Goal: minimize the uncertainty (comes in part from reachability)

Key idea: Build the Control-Flow Graph, explore routes through the graph to (over)approximate reachability

BUILDING THE CFG

CONTROL FLOW GRAPH ANALYSIS

In general, an iterative process:

Pass over instructions, mark leaders/ terminators

Might create new leaders / terminators,
so keep doing passes until no more found

Build basic blocks by boundaries

Connect control source to control destination

Refine based on analyses

For LLVM Bitcode, even easier

All blocks (except possibly entry) have labels

Connect control source to control destination

One deviation from LLVM: split after a call!

Refine based on analyses

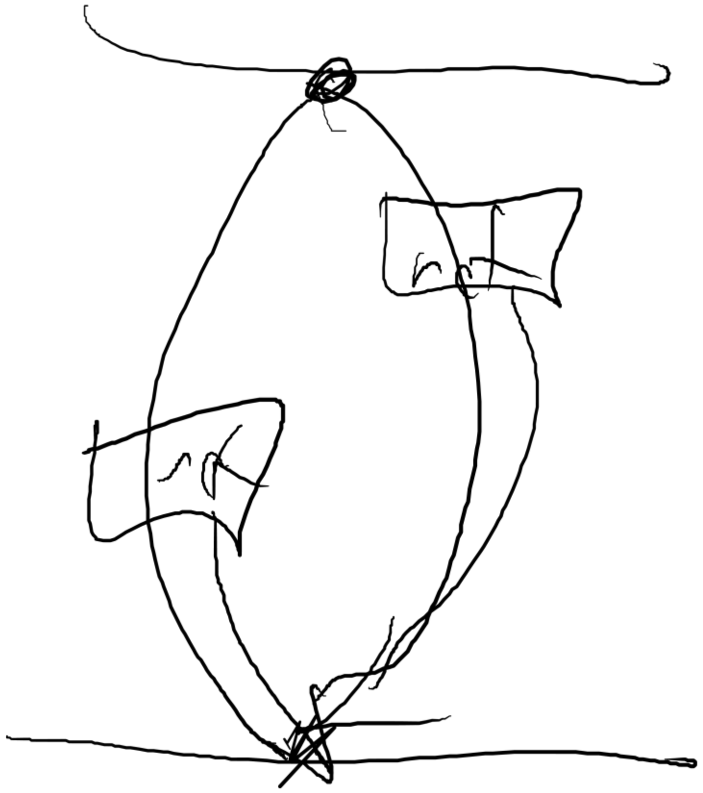
CFG NOTATION

STATIC ANALYSIS PHILOSOPHY

We'll enforce a couple of constraints on the CFG

It must be a hammock –
One entry point, one exit block

Call sites and return points are connected via a special link edge



VISUALIZING THE CFG: DOT

CONTROL FLOW GRAPH ANALYSIS

FLOW-SENSITIVE ANALYSIS RELIES HEAVILY ON THE CONTROL-FLOW GRAPH CONCEPT

It's pretty helpful to have a CFG in hand

Good news! You know how to automatically induce the CFG structure

Gooder news! There's a format to visualize CFGs

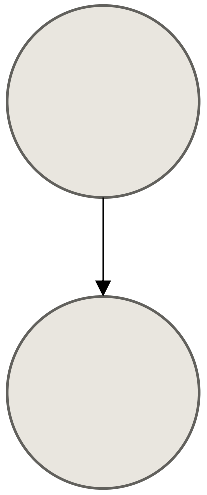
File graph.dot

```
digraph name {
  nodeA […];
  nodeB […];
  nodeA -> nodeB […];
}
```

cmdline

```
dot –Tpdf graph.dot –o graph.pdf
```

output



EXISTING CFG TOOLS

CONTROL FLOW GRAPH ANALYSIS

Good news! You know how to automatically induce the CFG structure

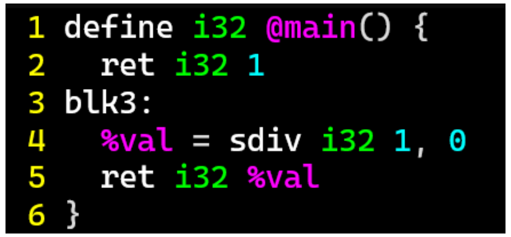
Gooder news! There’s a format to visualize CFGs

Goodest news! llvm can output a dot-format CFG for .ll-format code

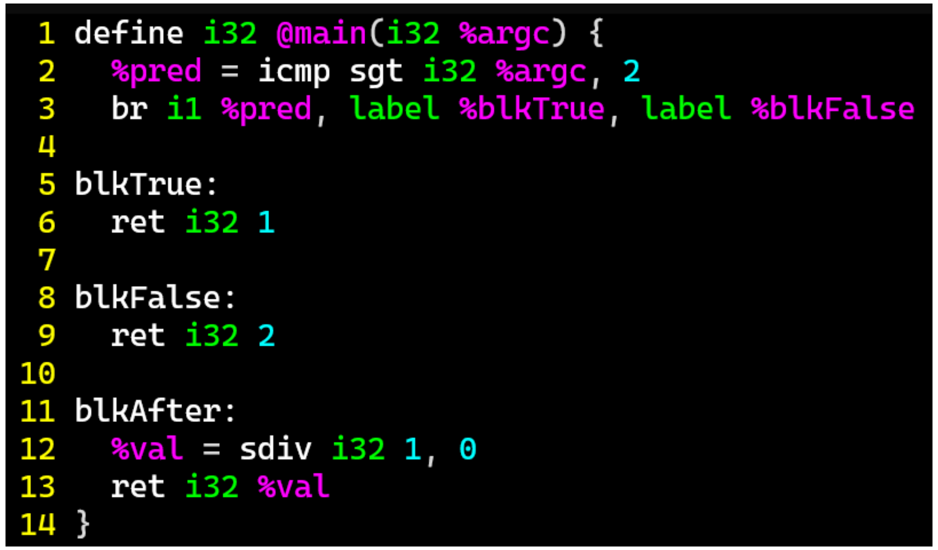
```
opt -dot-cfg prog.ll > /dev/null

opt -passes=dot-cfg prog.ll > /dev/null
```

OVERAPPROXIMATING REACHABILITY

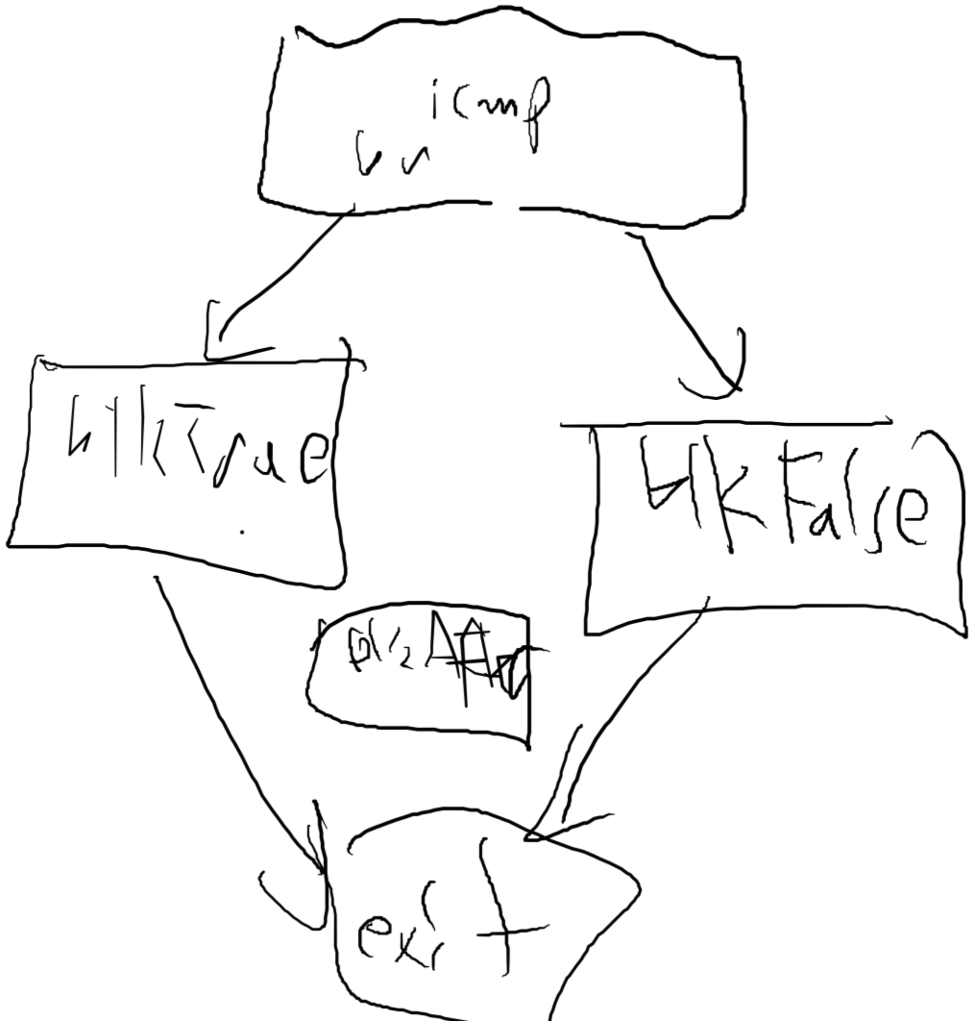
STATIC ANALYSIS PHILOSOPHY



```
define i32 @main() {
  ret i32 1
blk3:
  %val = sdiv i32 1, 0
  ret i32 %val
}
```



```
define i32 @main(i32 %argc) {
  %pred = icmp sgt i32 %argc, 2
  br i1 %pred, label %blkTrue, label %blkFalse

blkTrue:
  ret i32 1

blkFalse:
  ret i32 2

blkAfter:
  %val = sdiv i32 1, 0
  ret i32 %val
}
```



LECTURE OUTLINE

- Contextualizing Rice’s Theorem
- Program Guarantees
- Analysis over CFG
- Preciseness vs Efficiency

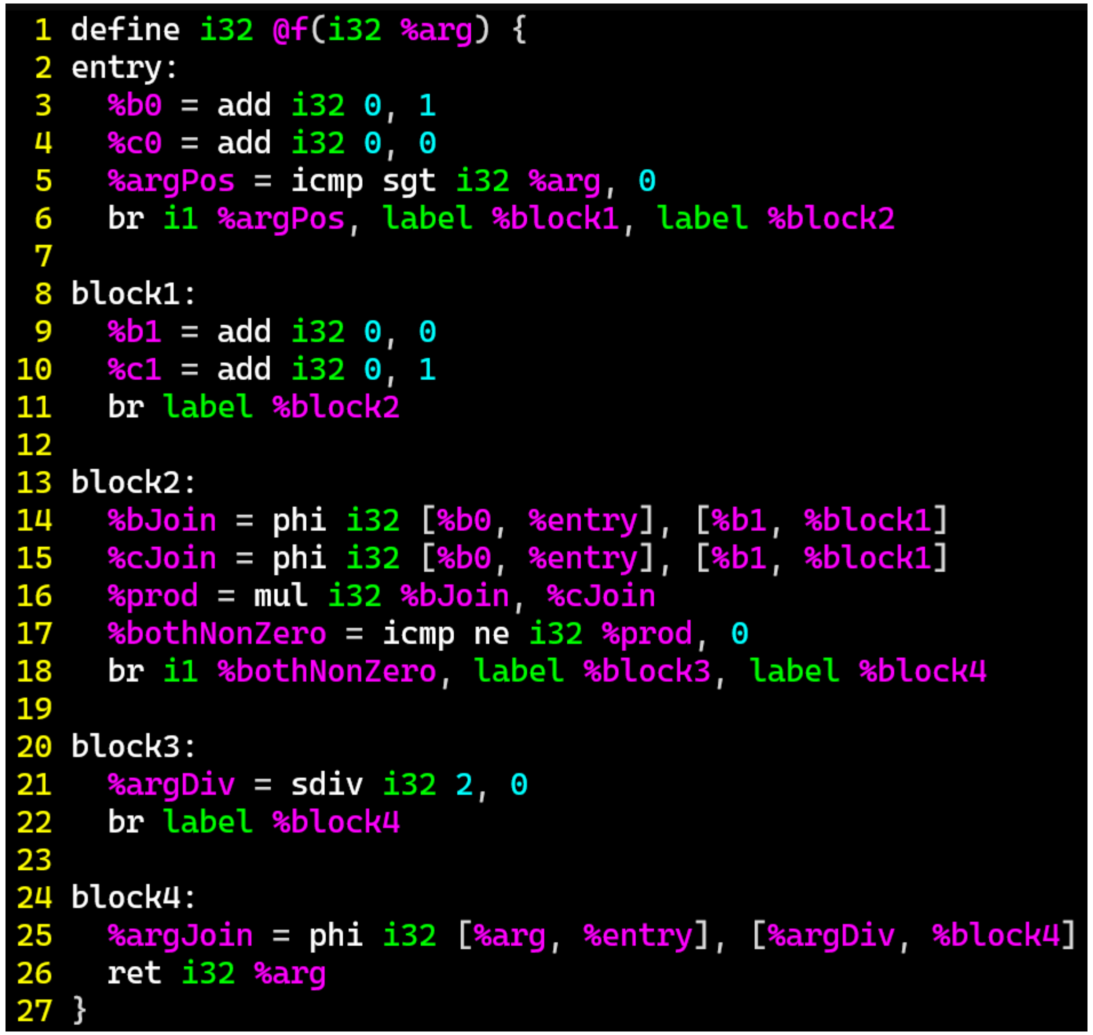
INFEASIBLE PATHS

STATIC ANALYSIS: DATAFLOW

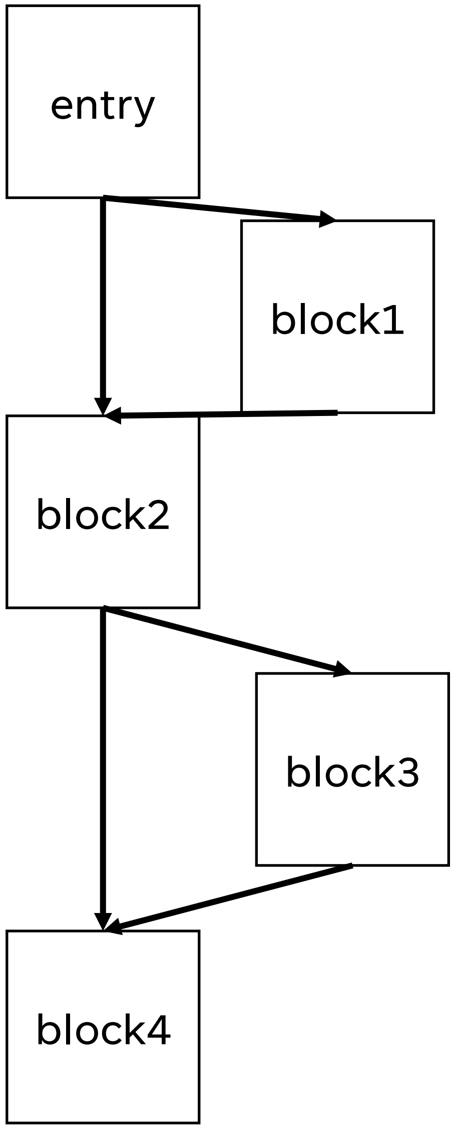
Is this program buggy?

Just check every path(?!?!?!)

```
int f(int arg) {
    int b = 1;
    int c = 0;
    if (arg > 0){
      b = 0;
      c = 1;
    }
    if (b && c) {
        arg = 2 / 0;
    }
    return arg;
}
```



```
define i32 @f(i32 %arg) {
entry:
  %b0 = add i32 0, 1
  %c0 = add i32 0, 0
  %argPos = icmp sgt i32 %arg, 0
  br i1 %argPos, label %block1, label %block2

block1:
  %b1 = add i32 0, 0
  %c1 = add i32 0, 1
  br label %block2

block2:
  %bJoin = phi i32 [%b0, %entry], [%b1, %block1]
  %cJoin = phi i32 [%b0, %entry], [%b1, %block1]
  %prod = mul i32 %bJoin, %cJoin
  %bothNonZero = icmp ne i32 %prod, 0
  br i1 %bothNonZero, label %block3, label %block4

block3:
  %argDiv = sdiv i32 2, 0
  br label %block4

block4:
  %argJoin = phi i32 [%arg, %entry], [%argDiv, %block4]
  ret i32 %arg
}
```



WRASSLIN' WITH STATE SPACE

PRECISENESS VS EFFICIENCY

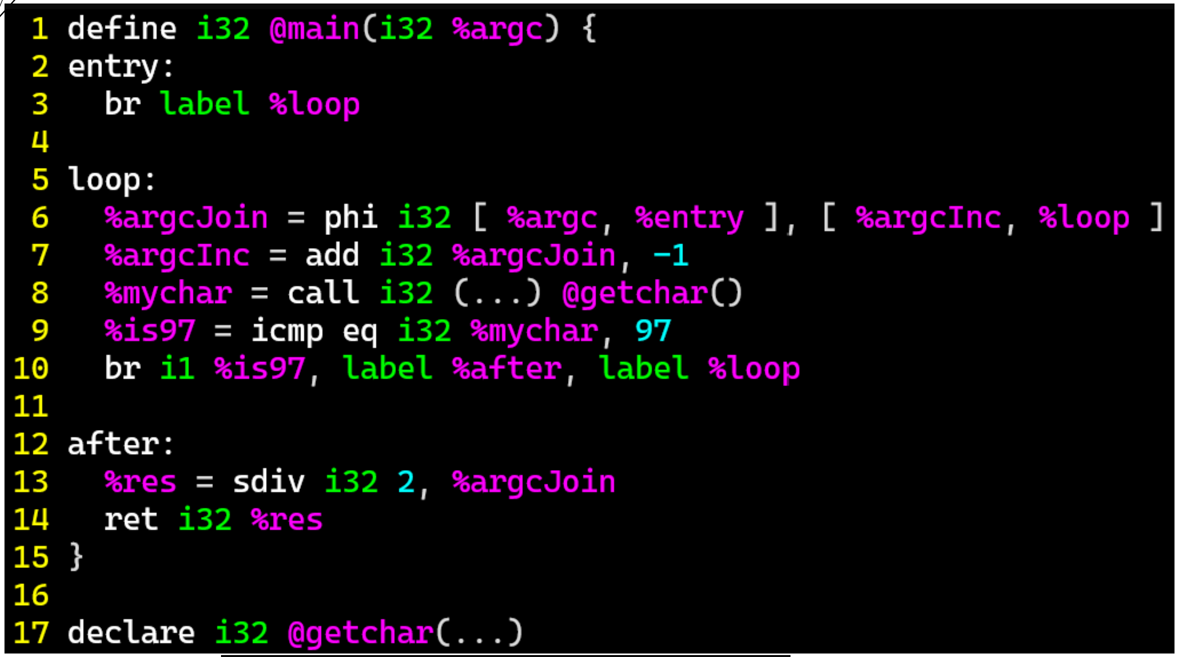
State space: the set of all possible configurations of the target model

Naïve state space representation: Consider each path separately

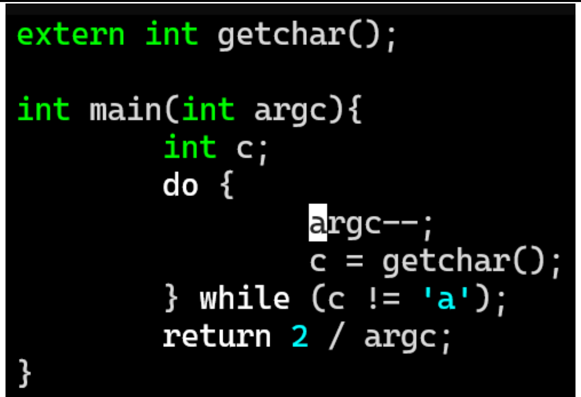
PATH-SENSITIVE DATAFLOW ANALYSIS

STATIC ANALYSIS: DATAFLOW

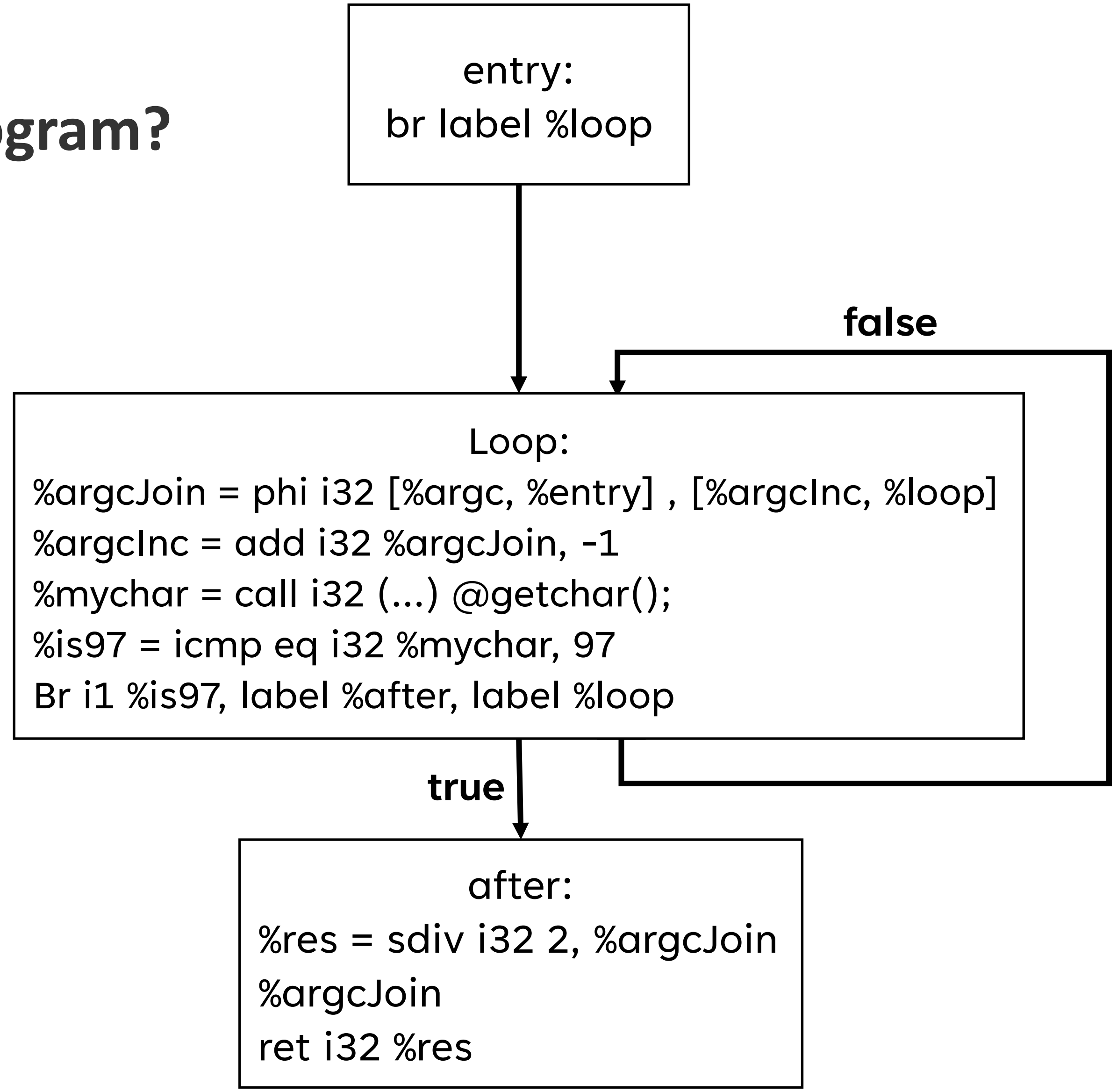
How many paths are there in this program?



```
define i32 @main(i32 %argc) {
entry:
  br label %loop

loop:
  %argcJoin = phi i32 [ %argc, %entry ], [ %argcInc, %loop ]
  %argcInc = add i32 %argcJoin, -1
  %mychar = call i32 (...) @getchar()
  %is97 = icmp eq i32 %mychar, 97
  br i1 %is97, label %after, label %loop

after:
  %res = sdiv i32 2, %argcJoin
  ret i32 %res
}

declare i32 @getchar(...)
```



```
extern int getchar();

int main(int argc){
        int c;
        do {
                argc--;
                c = getchar();
        } while (c != 'a');
        return 2 / argc;
}
```


entry:
br label %loop
false
Loop:
%argcJoin = phi i32 [%argc, %entry] , [%argcInc, %loop]
%argcInc = add i32 %argcJoin, -1
%mychar = call i32 (...) @getchar();
%is97 = icmp eq i32 %mychar, 97
Br i1 %is97, label %after, label %loop
true
after:
%res = sdiv i32 2, %argcJoin
%argcJoin
ret i32 %res

WRASSLIN' WITH STATE SPACE

PRECISENESS VS EFFICIENCY

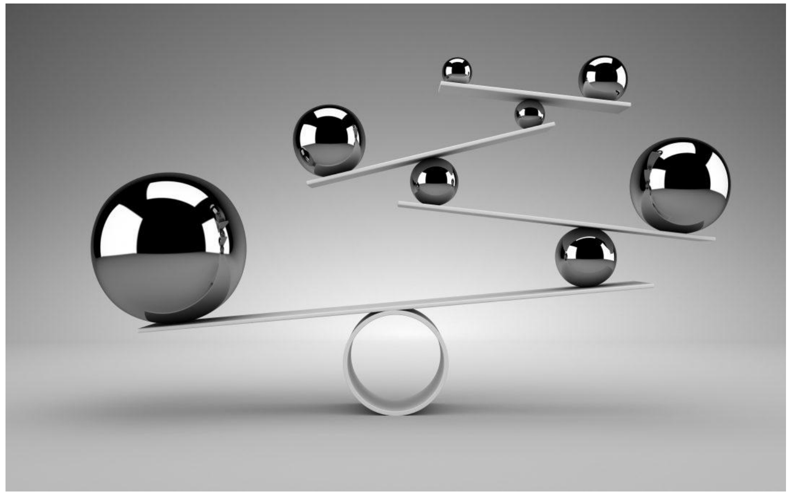
State space: the set of all possible configurations of the analysis target

Naïve state space representation: enumerate all configurations of a program

- For ~~n bits of memory: 2^n states~~

  n branches 2^n paths

Practical state space representation: Summarize sets of configurations of a program



THE POWER OF STATIC ANALYSIS

ANALYSIS SPECIFICITY

The power of static analysis:
You can see beyond the code that is executed in an individual trace

The responsibility of static analysis:
You need to consider the conditions/circumstances/context in which code *would be* executed while keeping state space small

OVERAPPROXIMATING REACHABILITY

STATIC ANALYSIS PHILOSOPHY

NEXT TIME

FLOW-SENSITIVE ANALYSIS