Questions about programs

- Does the program terminate on all inputs?
- How large can the heap become during execution?
- Can sensitive information leak to non-trusted users?
- Can non-trusted users affect sensitive information?
- Are buffer-overruns possible?
- Data races?
- SQL injections?
- XSS?
- …

Program points

```
foo(p,x) {
  var f,q;
  if (*p==0) { f=1; }
  else {
    q = alloc 10;
    *q = (*p)-1;
    f=(*p)*(x(q,x));
  }
  return f;
}
```

any point in the program
= any value of the PC

Invariants:

A property holds at a program point if it holds in any such state for any execution with any input

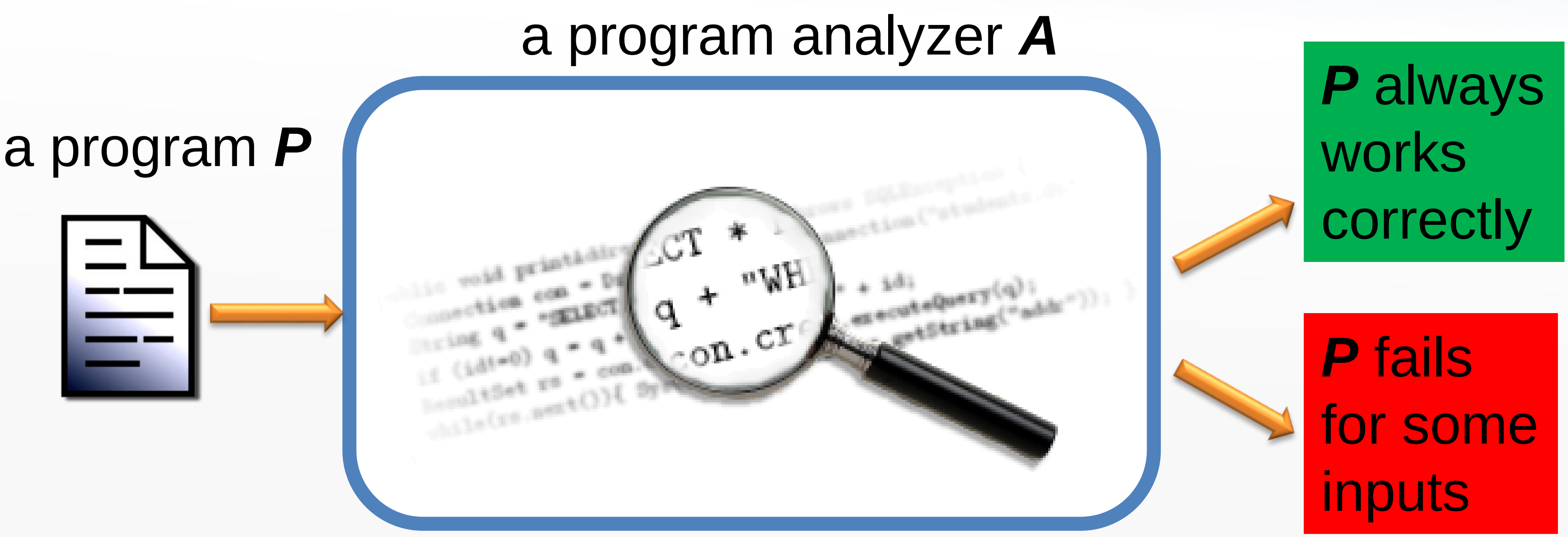
Questions about program points

- Will the value of `x` be read in the future?
- Can the pointer `p` be `null`?
- Which variables can `p` point to?
- Is the variable `x` initialized before it is read?
- What is a lower and upper bound on the value of the integer variable `x`?
- At which program points could `x` be assigned its current value?
- Do `p` and `q` point to disjoint structures in the heap?
- Can this `assert` statement fail?

Why are the answers interesting?

- Increase efficiency
  - resource usage
  - compiler optimizations
- Ensure correctness
  - verify behavior
  - catch bugs early
- Support program understanding
- Enable refactorings

Programs that reason about programs



Requirements to the perfect program analyzer

- **SOUNDNESS (don’t miss any errors)**
- **COMPLETENESS (don’t raise false alarms)**
- **TERMINATION (always give an answer)**

Rice’s theorem, 1953

CLASSES OF RECURSIVELY ENUMERABLE SETS AND THEIR DECISION PROBLEMS[1]

BY

H. G. RICE

1. **Introduction.** In this paper we consider classes whose elements are recursively enumerable sets of non-negative integers. No discussion of recursively enumerable sets can avoid the use of such classes, so that it seems desirable to know some of their properties. We give our attention here to the properties of complete recursive enumerability and complete recursiveness (which may be intuitively interpreted as decidability). Perhaps our most interesting result (and the one which gives this paper its name) is the fact that no nontrivial class is completely recursive.

We assume familiarity with a paper of Kleene [5][2], and with ideas which are well summarized in the first sections of a paper of Post [7].

I. FUNDAMENTAL DEFINITIONS

2. **Partial recursive functions.** We shall characterize recursively enumer-

COROLLARY B. *There are no nontrivial c.r. classes by the strong definition.*

Rice's theorem

Any non-trivial property of the behavior of programs in a Turing-complete language is undecidable!

Reduction to the halting problem

- Can we decide if a variable has a constant value?

```
x = 17; if (TM(j)) x = 18;
```

- Here, `x` is constant if and only if the *j*'th Turing machine does not halt on empty input

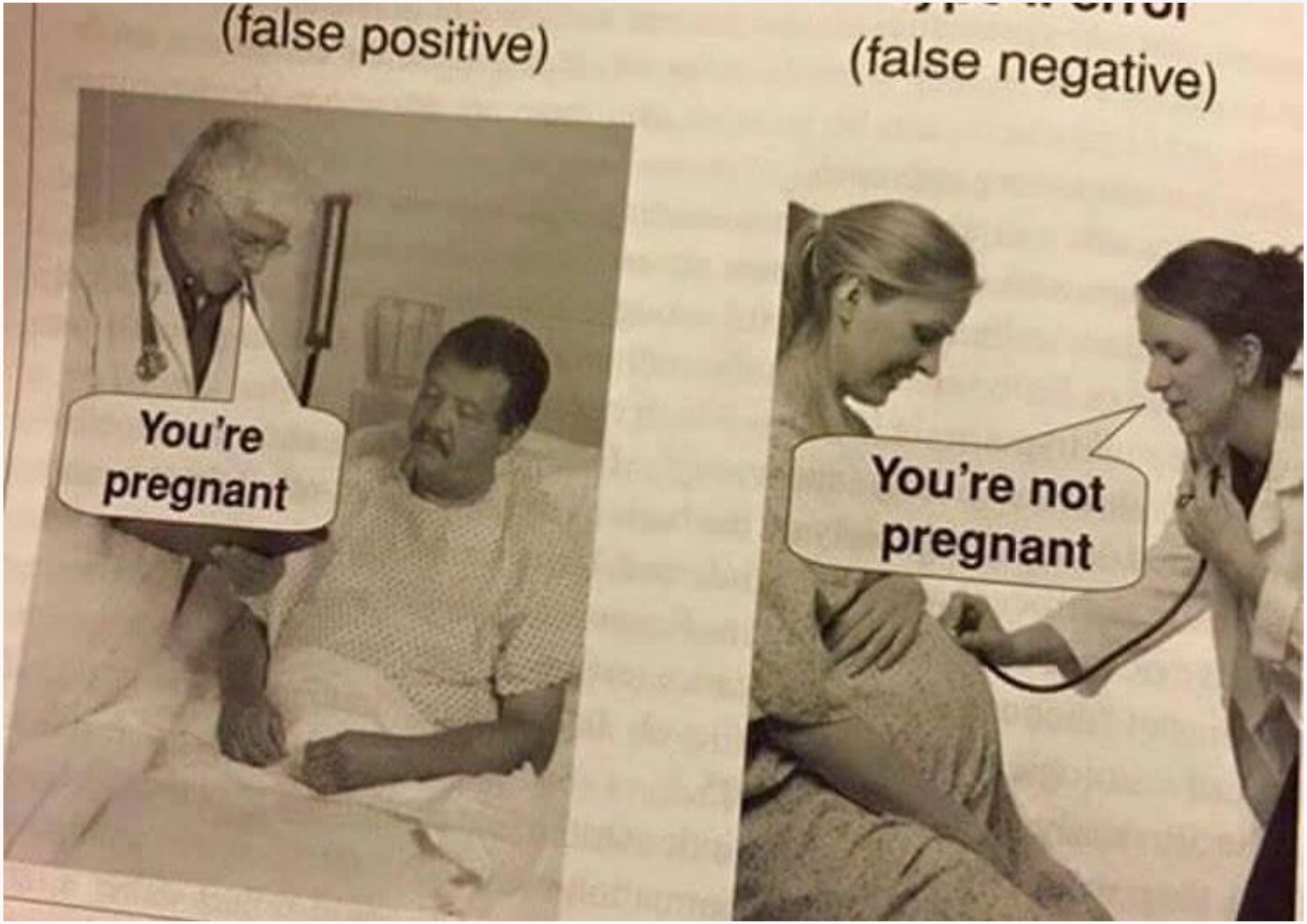
Approximation

- *Approximate* answers may be decidable!

- The approximation must be *conservative*:
  - i.e. only err on “the safe side”
  - which direction depends on the *client application*

- We'll focus on decision problems
- More subtle approximations if not only “*yes*”/“*no*”
  - e.g. memory usage, pointer targets

False positives and false negatives



Example approximations

- Decide if a given function is ever called at runtime:
  - if “*no*”, remove the function from the code
  - if “*yes*”, don’t do anything
  - the “*no*” answer *must* always be correct if given
- Decide if a cast `(A)x` will always succeed:
  - if “*yes*”, don’t generate a runtime check
  - if “*no*”, generate code for the cast
  - the “*yes*” answer *must* always be correct if given

Beyond “yes”/“no” problems

- How much memory / time may be used in any execution?

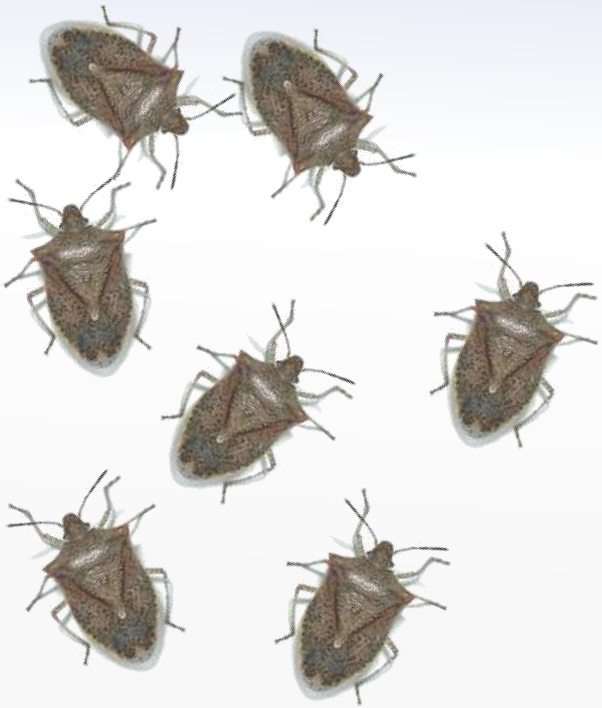
- Which variables may be the targets of a pointer variable p?

The engineering challenge

- A correct but trivial approximation algorithm may just give the useless answer every time
- The *engineering challenge* is to give the useful answer often enough to fuel the client application
- ... and to do so within reasonable time and space
- This is the hard (and fun) part of static analysis!

Bug finding

```
int main() {
  char *p,*q;
  p = NULL;
  printf("%s",p);
  q = (char *)malloc(100);
  p = q;
  free(q);
  *p = 'x';
  free(p);
  p = (char *)malloc(100);
  p = (char *)malloc(100);
  q = p;
  strcat(p,q);
}
```

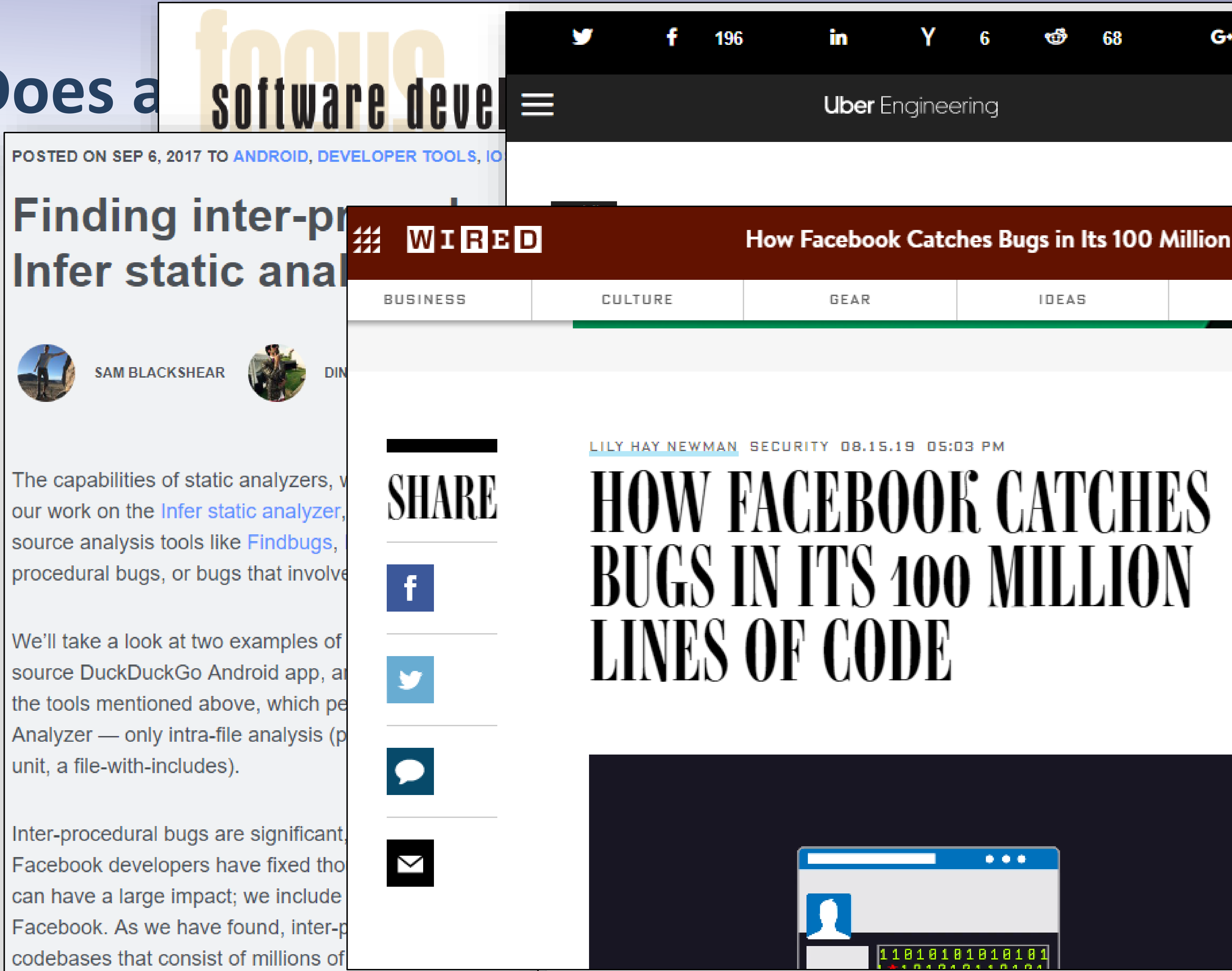


```
gcc -Wall foo.c
lint foo.c

No errors!
```

https://en.wikipedia.org/wiki/Lint_(software)

Does a



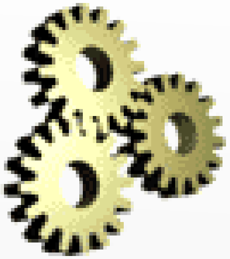
focus
software devel
196
in
Y
6
68
Uber Engineering
POSTED ON SEP 6, 2017 TO ANDROID, DEVELOPER TOOLS, IO
Finding inter-pr
Infer static anal
SAM BLACKSHEAR
DIN
The capabilities of static analyzers, w
our work on the Infer static analyzer,
source analysis tools like Findbugs,
procedural bugs, or bugs that involve
We'll take a look at two examples of
source DuckDuckGo Android app, ar
the tools mentioned above, which pe
Analyzer — only intra-file analysis (p
unit, a file-with-includes).
Inter-procedural bugs are significant,
Facebook developers have fixed tho
can have a large impact; we include
Facebook. As we have found, inter-p
codebases that consist of millions of
WIRED
How Facebook Catches Bugs in Its 100 Million
BUSINESS
CULTURE
GEAR
IDEAS
SHARE
LILY HAY NEWMAN SECURITY 08.15.19 05:03 PM
HOW FACEBOOK CATCHES BUGS IN ITS 100 MILLION LINES OF CODE

A constraint-based approach

```java
public class Matrix {
public static void main(String[] args) {
        int arr[][]=new int[3][3];
        System.out.println("Enter nine elements");
        Scanner sc=new Scanner(System.in);
        for(int i=0;i<arr.length;i++)
        {
            for(int j=0;j<arr.length;j++)
            {
                arr[i][j]=sc.nextInt();
            }
        }
        int sum=0;
        for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j++) {
        if (i == j)
        sum = sum + arr[i][j];
        }}
       System.out.println(sum);
    }
}
```

program to analyze

mathematical constraints



constraint solver

$[\![p]\!]$ = &int
$[\![q]\!]$ = &int
$[\![alloc]\!]$ = &int
$[\![x]\!] = \phi$
$[\![foo]\!] = \phi$
$[\![\&n]\!]$ = &int
$[\![main]\!]$ = ()->int

solution

Challenging features in modern programming language

- Higher-order functions
- Mutable records or objects, arrays
- Integer or floating-point computations
- Dynamic dispatching
- Inheritance
- Exceptions
- Reflection
- …