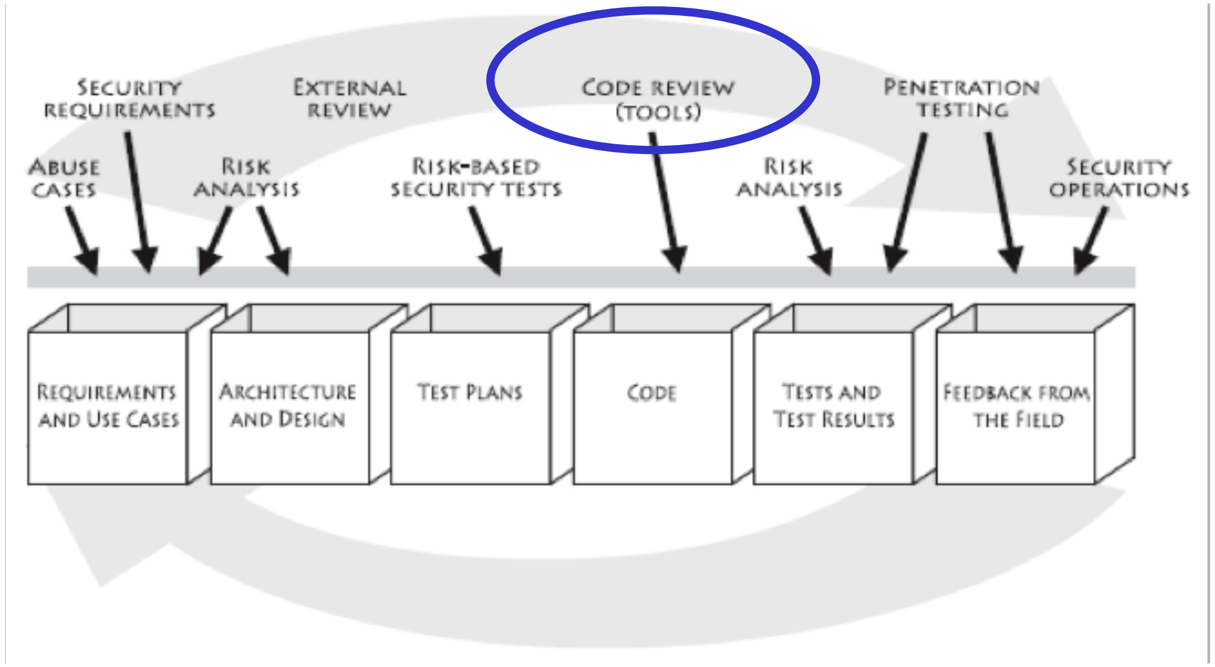
STATIC ANALYSIS FOR SECURITY

Fin ad ora ci siamo molto soffermati sul run time dei linguaggi di programmazione e abbiamo visto come viene cambiato, adesso ci soffermiamo sul backend del compilatore. Il frontend di un compilatore ha a che vedere con l'analisi sintattica e lessicale, dunque andare a verificare che il codice rispetta le regole sintattiche. Alla fine viene prodotto l'AST su cui vengono fatte molte analisi, dunque viene modificato a seguito di operazioni nel backend dei compilatori, come il controllo dei tipi o delle ottimizzazioni. Sono analisi statiche che hanno a che vedere con la struttura del programma dal punto di vista delle proprietà semantiche non sintattiche del programma.

SOURCE CODE ANALYSIS

1. Static Analysis = Automated analysis at compile time to find potential bugs
2. Broad range of techniques, from light- to heavyweight:
   - simple syntactic checks, e.g. grep on suitable patterns
   - type checking eg. adding an int and a bool
3. more advanced analyses taking into account semantics
   - dataflow analysis, control flow analysis, abstract interpretation, symbolic evaluation, constraint solving, taint analysis, program verification, model checking…
4. All compilers do some static analysis
5. List of tools
   1. https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

STATIC ANALYSIS AND SW DEV.



WHY STATIC ANALYSIS?

Traditional methods of finding errors in SW development:

- Testing
- code inspection

Security errors can be hard to find by these methods, because they

- only arise in unusual circumstances
  - particular inputs uncommon execution paths, …

Code base is too large for a human code inspection

- Here static analysis can provide major improvement

- 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?
- …

QUESTIONS ABOUT PROGRAMS

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 program counter

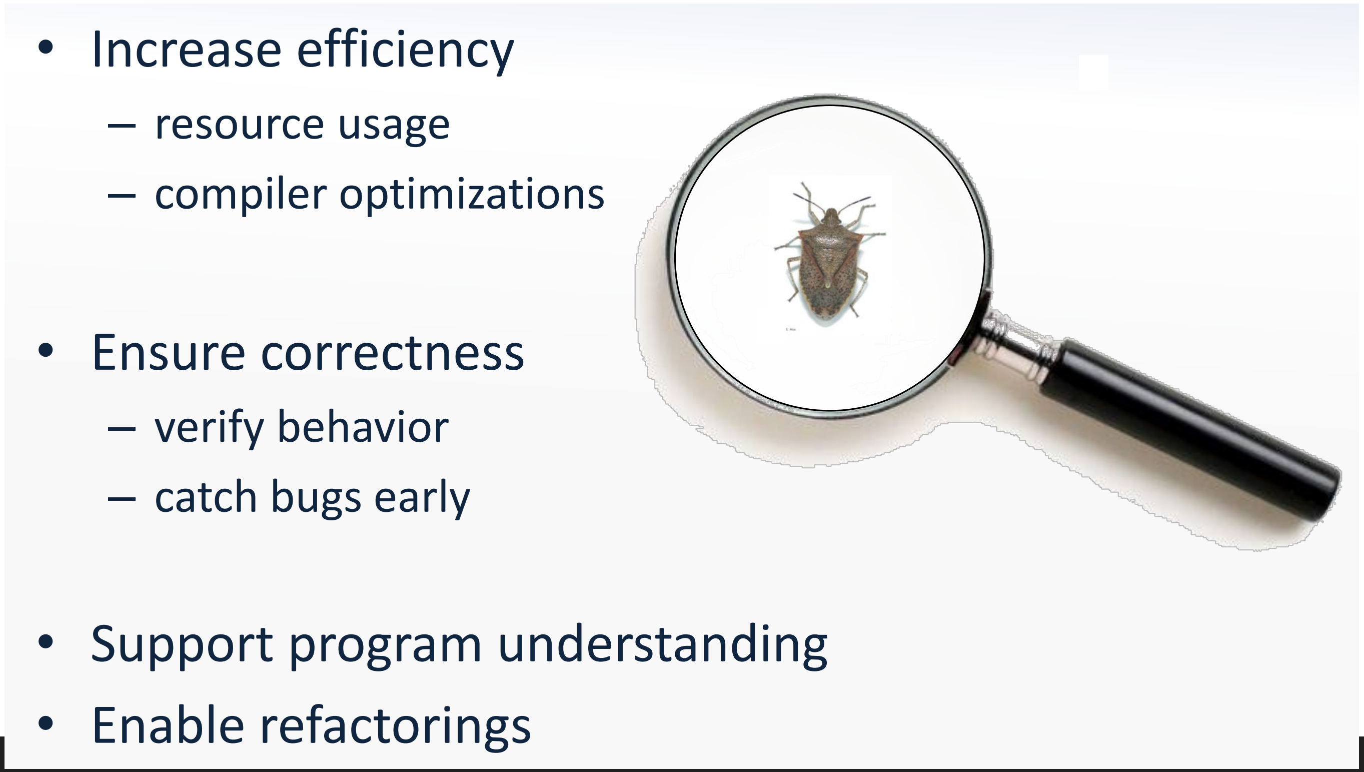
INVARIANT

è una proprietà che vale per tutti i punti di esecuzione del programma

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

QUESTIONS ABOUT PROGRAM POINTS

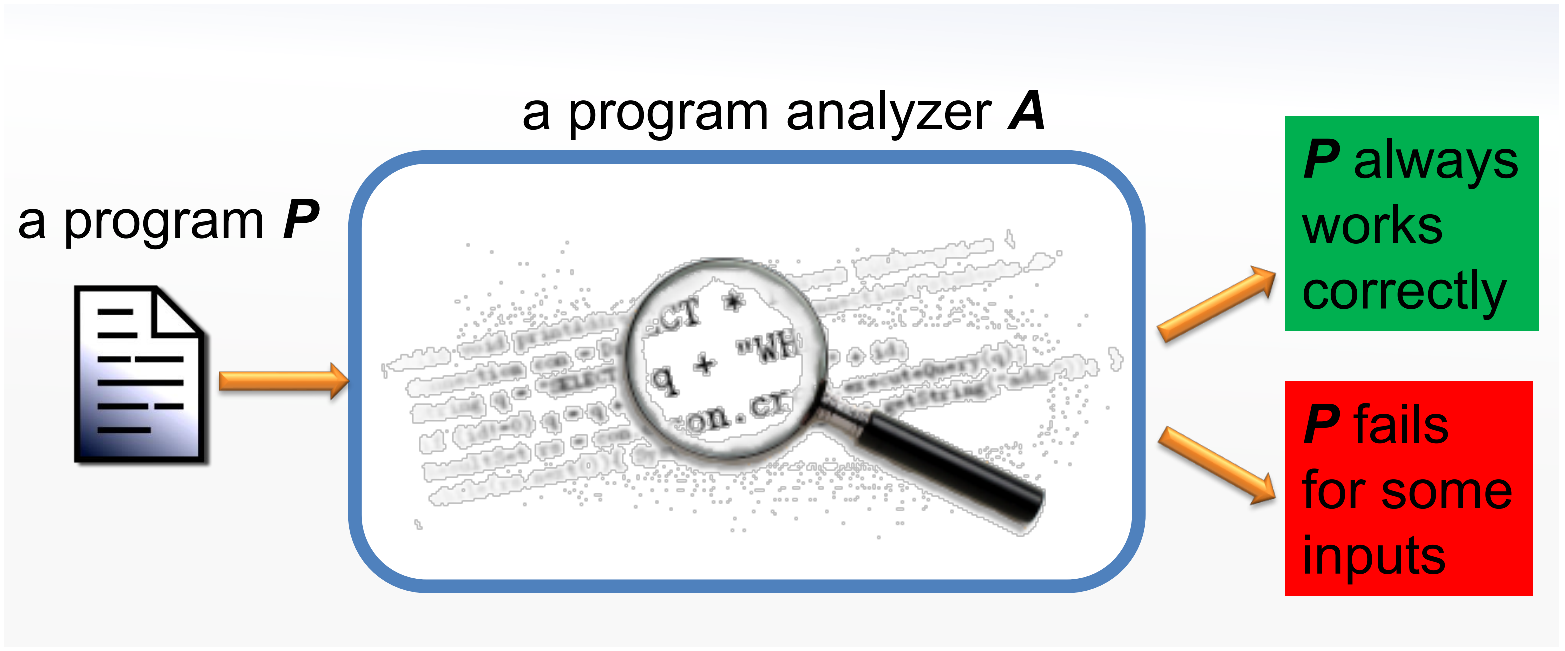
- Will the value of variable x be read in the future?
- Can the pointer p be null?
- Which variables can p point to?
- Is the variable x initialized before it is read?
- What is a lower and upper bound on the value of the integer variable x?
- At which program points could x be assigned its current value?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



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

… THE ANSWER ARE INTERESTING

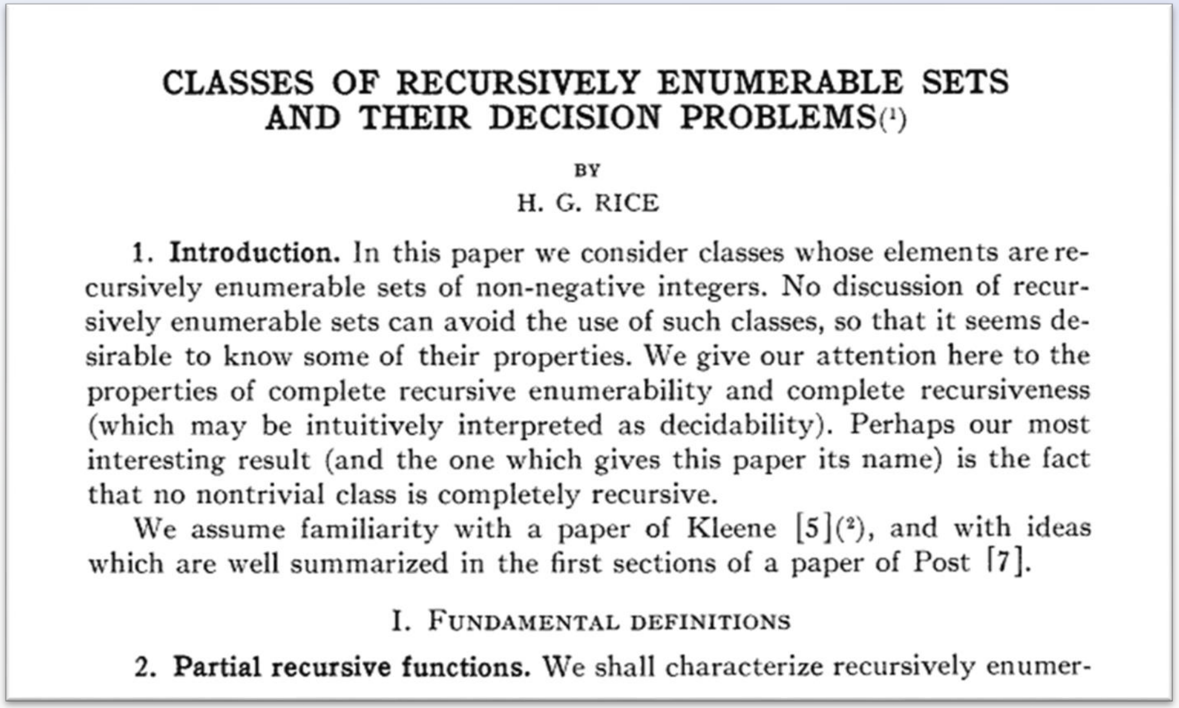
PROGRAMS THAT REASON ABOUT PROGRAMS


a program analyzer ***A***
a program ***P***
P always works correctly
P fails for some inputs

THE PERFECT PROGRAM ANALYZER

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

RICE 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-

Ogni qualunque proprietà ragionevole di programmi è in generale non decidibile. Non si hanno mai programmi che dicono si o no relativamente al comportamento di altri programmi per classi di proprietà generali.

RICE THEOREM

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

SOLUTION: APPROXIMATION

- *Approximate* answers may be decidable!
- The approximation must be *conservative*:

EXAMPLE

- 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

AN 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!

un programma è giudicato corretto ma in realtà è corretto

FALSE POSITIVE AND FALSE NEGATIVE

importante avere un bilanciamento tra i 2 falsi

- Important quality measures for a static analysis techniques (and tools)
- rate of false positives
  - tool complains about non-error
- rate of false negatives
  - tool fails to complain about error

IN FORMAL TERMS

- A static analysis is sound if it only finds real bugs,
  - no false positives
- A static analysis is complete if it finds all bugs,
  - no false negatives

STATIC ANALYSIS YOU ALREADY KNOW

- Warning about unused variables
- Warning about dead/unreachable code
- Warning about missing initialisation
  - possibly as part of language definition (eg in Java) and checked by compiler

EXAMPLE

qui si vede che esiste un flusso di controllo, ovvero un comportamento del programma, in cui c non è inizializzato

control flow analysis

```
if (b) { c = 5; } else { c = 6; }    initialises c
if (b) { c = 5; } else { d = 6; }    does not
```

data flow analysis

```
d = 5;  c = d;          initialises c
c = d;  d = 5;          does not
```

analisi riguardante come i dati fluiscono

```
static OSStatus SSLVerifySignedServerKeyExchange (SSLContext
*ctx, bool isRsa, SSLBuffer signedParams, uint8_t *signature,
UInt16 signatureLen)
{ OSStatus  err;
   ..
 if((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
 if((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        ~~goto fail;~~
 if((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
   ...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
}
```

Infamous goto bug in iOS implementation of TLS

- Dead code analysis would easily reveal this flaw!
- Or simply code style that insists you always use { } for branches

LIMITS OF STATIC ANALYSES

Does

```
    if (i < 5 ) { c = 5; }
    if ((i < 0) || (i*i > 20 )){ c = 6; }
```

initialise c?

LIMITS

Many analyses become hard – or undecidable - at some stage

Analysis tools can then:

- report that they “DON’T KNOW”
- give a (possible) false positive
- give a (possible) false negative

SUMMARY ON SA TOOLS

1. Acceptable level of false positives
2. Not too many warnings
3. Good error reporting
4. Bugs should be easy to fix
5. One should be able to teach the tool to suppress false positives

READING

http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/

STATIC ANALYSIS AND SECURITY

Static analyses are used by compilers to carry out a transformation of a high level, abstract program to low level executable code.

In today's world, it is important to guarantee that compilation does not weaken security properties.

L'idea dell'analisi statica è quella di prendere il programma e fare delle analisi rispetto alla semantica del programma stesso il che vuol dire che affronta delle trasformazioni della rappresentazione del codice che sono compatibili con i comportamenti. La trasformazione del codice è un qualcosa che trasforma un programma semanticamente corretto in un altro programma semanticamente corretto che rispetta le stesse regole di comportamento. Ciò consente di ottenere programmi più semplici o che rispettano maggiori proprietà in modo tale da preservare il comportamento originale e quindi garantire l'efficienza preservando il comportamento. Lo sviluppo dei linguaggi di programmazione moderni deve tenere conto non solo della correttezza ma anche della sicurezza, cosa che gli strumenti di analisi statica normalmente non fanno.

CORRECTNESS VS SECURITY

Correctness and security turn out to be distinct issues.

Illustrating example: the dead-store elimination optimization

esempio del TLS in iOS

This optimization removes an assignment statement from a program if the value stored is never used.

QUESTO E' UN ESEMPIO SANO DI UTILIZZO DAL PUNTO DI VISTA DELLA SICUREZZA

metto la password a 0 perchè poi dopo uso una funzione untrusted, se avessi lasciato la password non pulita, questa funzione untrusted avrebbe potuto provare a ottenerla

```
void foo()
{
   int x;

   x = read_password();
   use(x);
   x = 0; // clear password
   untrusted();
   return;
}
```

AN EXAMPLE

```
void foo()
{
  int x;

  x = read_password();
  use(x);
  x = 0; // clear password
  untrusted();
  return;
}
```

As the value 0 stored to x is never used,
The assignment (x = 0) is removed (silently)
by the dead-store optimization

AN EXAMPLE

THE TRANFORMATION

void foo()

```
void foo()
{
   int x;

   x = read_password();
   use(x);
   // skip
   untrusted();
   return;
}
```

The dead store optimization leaves the password on the stack or in a register longer than was originally intended, making it possible for an attack elsewhere in the program to obtain the password.

The optimization is **correct** in that it preserves the input-output behavior of the original program; however, it is **insecure**.

THE CHALLENGE

- Along with functional preservation, the static technuques (compiler backend) would like to ensure the preservation of security properties:
  - the final executable should be at least as secure against attack as the original program.