# Compilers and Formal Languages (9)

| | |
|---|---|
| Email: | christian.urban at kcl.ac.uk |
| Office: | N7.07 (North Wing, Bush House) |
| Slides: | KEATS (also homework is there) |

# While Language

$$
\begin{array}{lcl}
\textbf{\textit{Stmt}} & ::= & \texttt{skip} \\
 & | & \textit{Id} \; \texttt{:=} \; \textbf{\textit{AExp}} \\
 & | & \texttt{if} \; \textbf{\textit{BExp}} \; \texttt{then} \; \textbf{\textit{Block}} \; \texttt{else} \; \textbf{\textit{Block}} \\
 & | & \texttt{while} \; \textbf{\textit{BExp}} \; \texttt{do} \; \textbf{\textit{Block}} \\
 & | & \texttt{read} \; \textit{Id} \\
 & | & \texttt{write} \; \textit{Id} \\
 & | & \texttt{write} \; \textit{String} \\
\textbf{\textit{Stmts}} & ::= & \textbf{\textit{Stmt}} \;\; ; \;\; \textbf{\textit{Stmts}} \mid \textbf{\textit{Stmt}} \\
\textbf{\textit{Block}} & ::= & \{ \, \textbf{\textit{Stmts}} \, \} \mid \textbf{\textit{Stmt}} \\
\textbf{\textit{AExp}} & ::= & \ldots \\
\textbf{\textit{BExp}} & ::= & \ldots
\end{array}
$$

# Fibonacci Numbers

```
write ”Fib”;
read n;
minus1 := 0;
minus2 := 1;
while n > 0 do {
       temp := minus2;
       minus2 := minus1 + minus2;
       minus1 := temp;
       n := n - 1
};
write ”Result”;
write minus2
```

# BF***

some big array, say `a`; 7 (8) instructions:

- `>` move `ptr++`
- `<` move `ptr--`
- `+` add `a[ptr]++`
- `-` subtract `a[ptr]--`
- `.` print out `a[ptr]` as ASCII
- `[` if `a[ptr] == 0` jump just after the corresponding `]`; otherwise `ptr++`
- `]` if `a[ptr] != 0` jump just after the corresponding `[`; otherwise `ptr++`

# Arrays in While

- `new arr[15000]`
- `x := 3 + arr[3 + y]`
- `arr[42 * n] := ...`

# New Arrays

```
new arr[number]
```

```
ldc number
newarray int
astore loc_var
```

# Array Update

```
arr[...] :=
```

```
aload loc_var
index_aexp
value_aexp
iastore
```

# Array Lookup in AExp

```
...arr[...]...
```

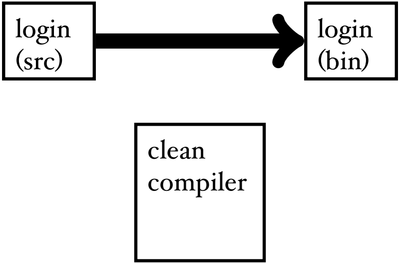
```
aload loc_var
index_aexp
iaload
```

**Using a compiler,
how can you mount the
perfect attack against a system?**

# What is a perfect attack?

1. you can potentially completely take over a target system
2. your attack is (nearly) undetectable
3. the victim has (almost) no chance to recover

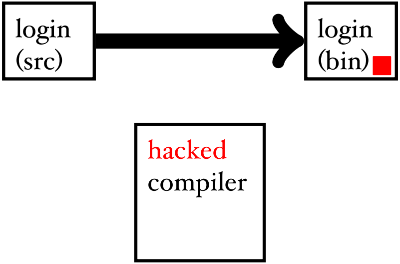
clean
compiler



login
(src)

login
(bin)

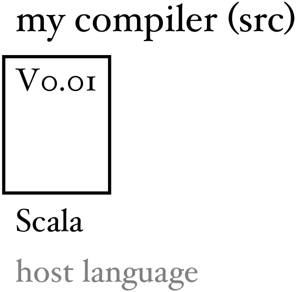
clean
compiler



login (src)

login (bin) 🟥

hacked compiler



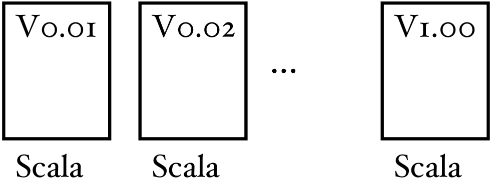
my compiler (src)

V0.01

Scala

host language

my compiler (src)



V0.01

V0.02

...

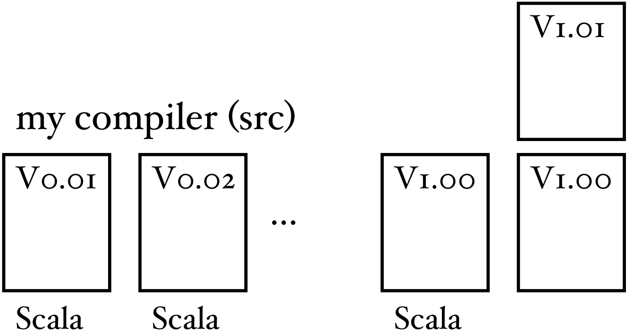
V1.00

Scala

Scala

Scala

host language



my compiler (src)

V0.01

V0.02

...

V1.00

V1.01

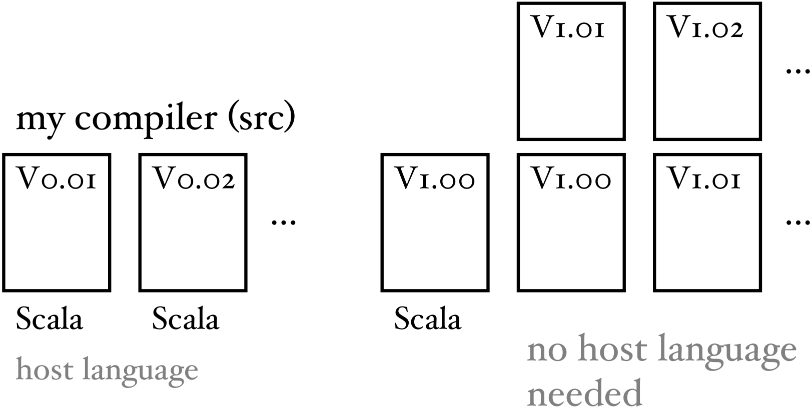
V1.00

Scala

Scala

Scala

host language



my compiler (src)

V0.01 V0.02 ...

Scala Scala

host language

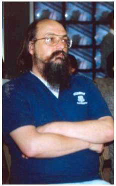
V1.01 V1.02 ...

V1.00 V1.00 V1.01 ...

Scala

no host language needed

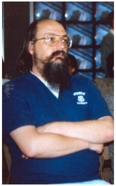
# Hacking Compilers


Ken Thompson
Turing Award, 1983

Ken Thompson showed how to hide a Trojan Horse in a compiler without leaving any traces in the source code.

No amount of source level verification will protect you from such Thompson-hacks.

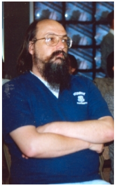
# Hacking Compilers



Ken Thompson
Turing Award, 198

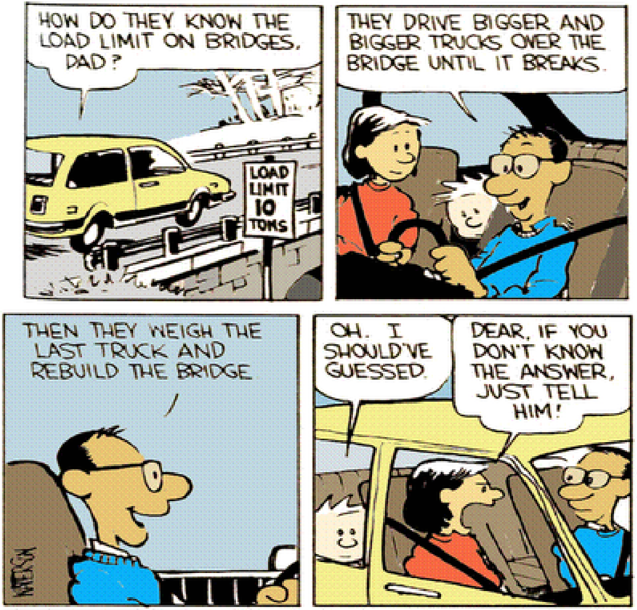
1) *Assume you ship the compiler as binary and also with sources.*
2) *Make the compiler aware when it compiles itself.*
3) *Add the Trojan horse.*
4) *Compile.*
5) *Delete Trojan horse from the sources of the compiler.*
6) *Go on holiday for the rest of your life. ;o)*

# Hacking Compilers


Ken Thompson
Turing Award, 1983

Ken Thompson showed how to hide a Trojan Horse in a compiler without leaving any traces in the source code.

No amount of source level verification will protect you from such Thompson-hacks.


HOW DO THEY KNOW THE LOAD LIMIT ON BRIDGES, DAD?
LOAD LIMIT 10 TONS
THEY DRIVE BIGGER AND BIGGER TRUCKS OVER THE BRIDGE UNTIL IT BREAKS.
THEN THEY WEIGH THE LAST TRUCK AND REBUILD THE BRIDGE.
WATERSON
OH. I SHOULD'VE GUESSED.
DEAR, IF YOU DON'T KNOW THE ANSWER, JUST TELL HIM!

# Compilers & Boeings 777

First flight in 1994. They want to achieve triple redundancy in hardware faults.

They compile 1 Ada program to

- Intel 80486
- Motorola 68040 (old Macintosh's)
- AMD 29050 (RISC chips used often in laser printers)

using 3 independent compilers.

# Compilers & Boeings 777

First flight in 1994. They want to achieve triple redundancy in hardware faults.

They compile 1 Ada program to

- Intel 80486
- Motorola 68040 (old Macintosh's)
- AMD 29050 (RISC chips used often in laser printers)

using 3 independent compilers.

Airbus uses C and static analysers. Recently started using CompCert.

# Goal

Remember the Bridges example?

- Can we look at our programs and somehow ensure they are bug free/correct?

# Goal

Remember the Bridges example?

- Can we look at our programs and somehow ensure they are bug free/correct?

- Very hard: Anything interesting about programs is equivalent to the Halting Problem, which is undecidable.

# Goal

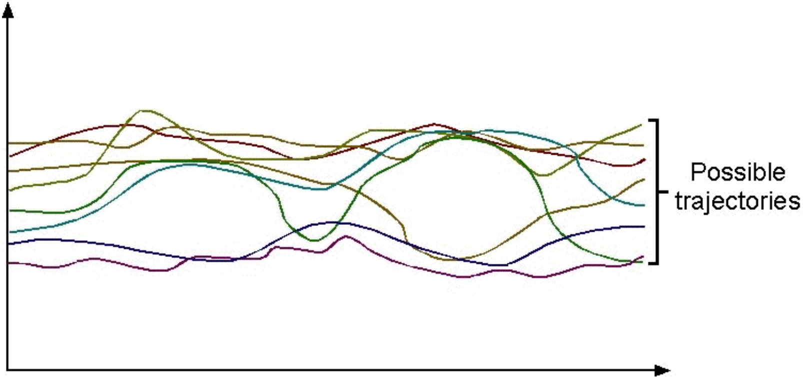
Remember the Bridges example?

- Can we look at our programs and somehow ensure they are bug free/correct?

- Very hard: Anything interesting about programs is equivalent to the Halting Problem, which is undecidable.

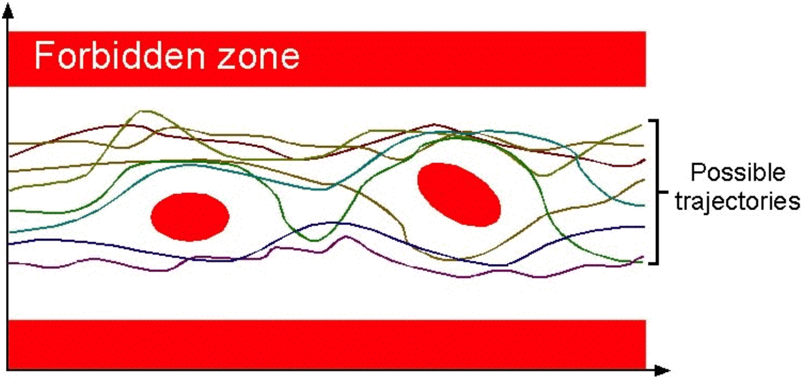
- Solution: We avoid this “minor” obstacle by being as close as possible of deciding the halting problem, without actually deciding the halting problem. ⇒ yes, no, don’t know (static analysis)

# What is Static Analysis?

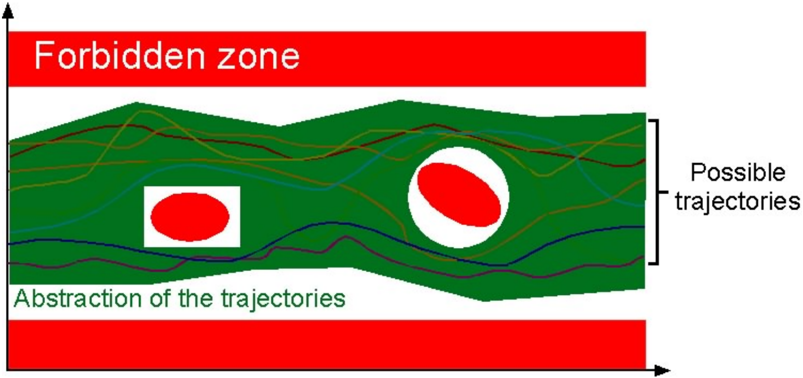


- depending on some initial input, a program (behaviour) will “develop” over time.

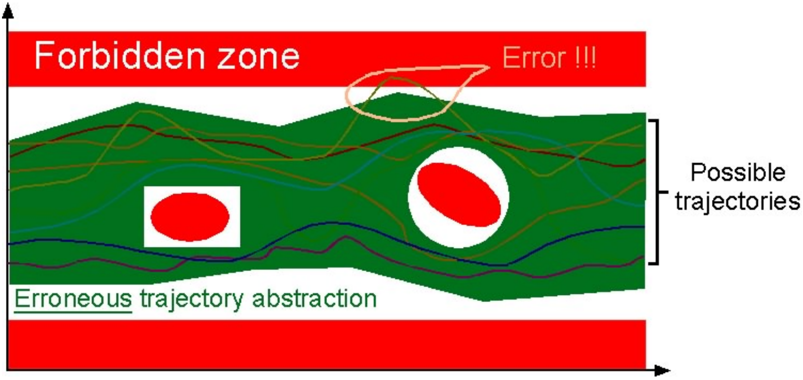
# What is Static Analysis?



# What is Static Analysis?

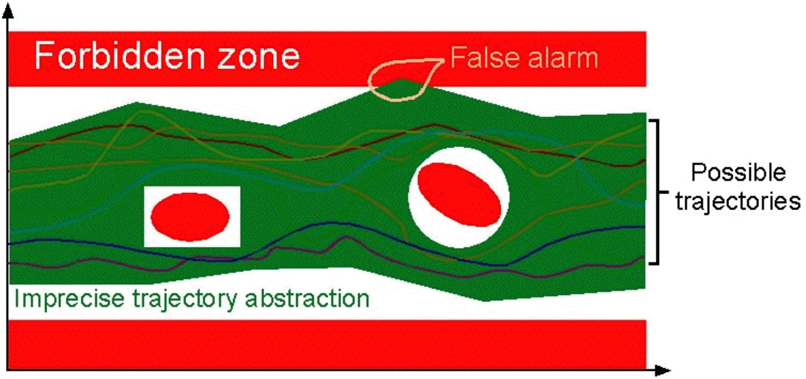


# What is Static Analysis?



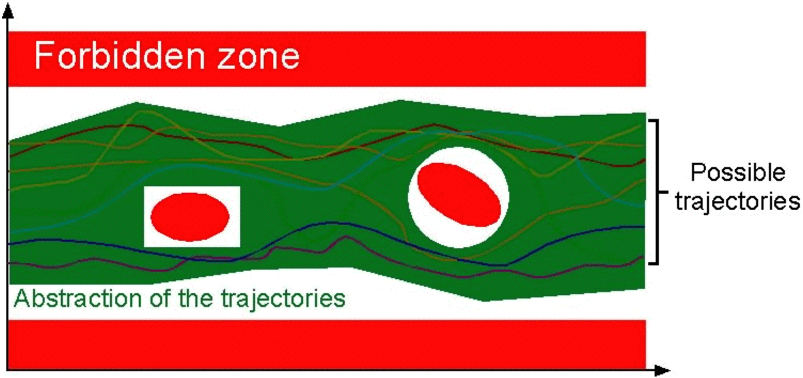
- to be avoided

# What is Static Analysis?



- this needs more work

# What is Static Analysis?



# Concrete Example: Are Vars Definitely Initialised?

Assuming x is initialised, what about y?

Prog. 1:

```
if x < 1 then y := x else y := x + 1;
y := y + 1
```

Prog. 2:

```
if x < x then y := y + 1 else y := x;
y := y + 1
```

# Concrete Example: Are Vars Definitely Initialised?

What should the rules be for deciding when a variable is initialised?

# Concrete Example: Are Vars Definitely Initialised?

What should the rules be for deciding when a variable is initialised?

- variable `x` is definitely initialized after `skip` iff `x` is definitely initialized before `skip`.

# $A$ is the set of definitely defined variables:

$$\frac{}{A \; \texttt{skip} \; A} \qquad \frac{vars(a) \subseteq A}{A \;\; (\texttt{x := a}) \;\; \{x\} \cup A}$$

# $A$ is the set of definitely defined variables:

$$\frac{}{A\ \texttt{skip}\ A} \qquad \frac{vars(a)\ \subseteq A}{A\ \ (\texttt{x := a})\ \ \{x\} \cup A}$$

$$\frac{A_1\ s_1\ A_2 \quad A_2\ s_2\ A_3}{A_1\ (s_1; s_2)\ A_3}$$

## $A$ is the set of definitely defined variables:

$$\frac{}{A\ \texttt{skip}\ A} \qquad \frac{vars(a) \subseteq A}{A\ \ (\texttt{x := a})\ \ \{x\} \cup A}$$

$$\frac{A_1\ s_1\ A_2 \quad A_2\ s_2\ A_3}{A_1\ (s_1;s_2)\ A_3}$$

$$\frac{vars(b) \subseteq A \quad A\ s_1\ A_1 \quad A\ s_2\ A_2}{A\ (\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\ A_1 \cap A_2}$$

## $A$ is the set of definitely defined variables:

$$\frac{}{A\ \texttt{skip}\ A} \qquad \frac{vars(a) \subseteq A}{A\ \ (\texttt{x := a})\ \ \{x\} \cup A}$$

$$\frac{A_1\ s_1\ A_2 \quad A_2\ s_2\ A_3}{A_1\ (s_1;s_2)\ A_3}$$

$$\frac{vars(b) \subseteq A \quad A\ s_1\ A_1 \quad A\ s_2\ A_2}{A\ (\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\ A_1 \cap A_2}$$

$$\frac{vars(b) \subseteq A \quad A\ s\ A'}{A\ (\texttt{while}\ b\ \texttt{do}\ s)\ A}$$

$A$ is the set of definitely defined variables:

$$\frac{}{A\ \texttt{skip}\ A} \qquad \frac{vars(a) \subseteq A}{A\ \ (\texttt{x := a})\ \ \{x\} \cup A}$$

$$\frac{A_1\ s_1\ A_2 \quad A_2\ s_2\ A_3}{A_1\ (s_1;s_2)\ A_3}$$

$$\frac{vars(b) \subseteq A \quad A\ s_1\ A_1 \quad A\ s_2\ A_2}{A\ (\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\ A_1 \cap A_2}$$

$$\frac{vars(b) \subseteq A \quad A\ s\ A'}{A\ (\texttt{while}\ b\ \texttt{do}\ s)\ A}$$

we start with $A = \{\}$

# Dijkstra on Testing

“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.”

What is good about compilers: the either seem to work, or go horribly wrong (most of the time).

# Proving Programs to be Correct

**Theorem:** There are infinitely many prime numbers.

**Proof** ...

similarly

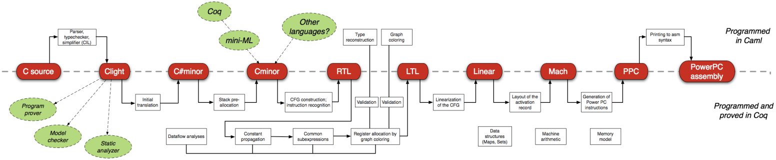
**Theorem:** The program is doing what it is supposed to be doing.

**Long, long proof** ...

This can be a gigantic proof. The only hope is to have help from the computer. 'Program' is here to be understood to be quite general (compiler, OS, ...).

# Can This Be Done?

- in 2008, verification of a small C-compiler
  - “if my input program has a certain behaviour, then the compiled machine code has the same behaviour”
  - is as good as `gcc -O1`, but much, much less buggy



# Fuzzy Testing C-Compilers

- tested GCC, LLVM and others by randomly generating C-programs
- found more than 300 bugs in GCC and also many in LLVM (some of them highest-level critical)
- about CompCert:

> “The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task.”