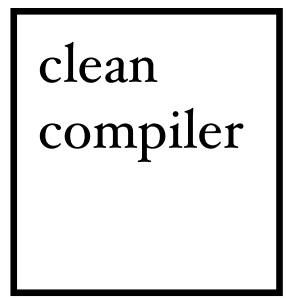
Automata and Formal Languages (9)

Email: christian.urban at kcl.ac.uk
Office: S1.27 (1st floor Strand Building)
Slides: KEATS (also home work is there)

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

What is a perfect attack?

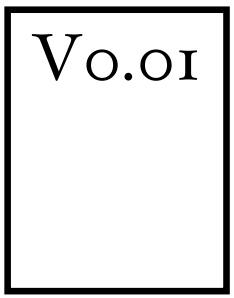
1. you can potentially completely take over a target system
2. your attack is (nearly) undetectable


clean
compiler


login
infected
...


clean
compiler

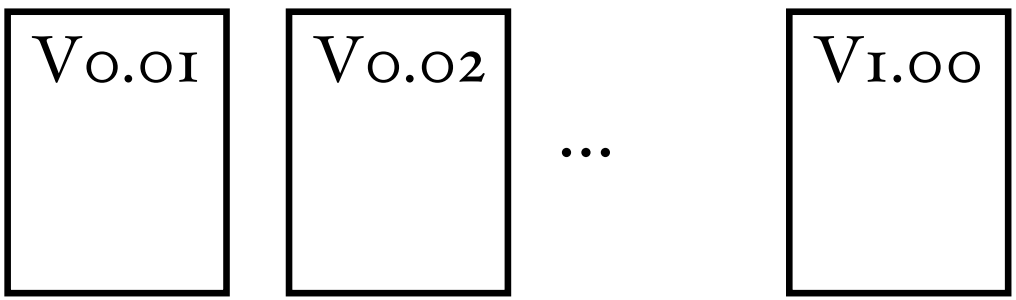
my compiler (src)



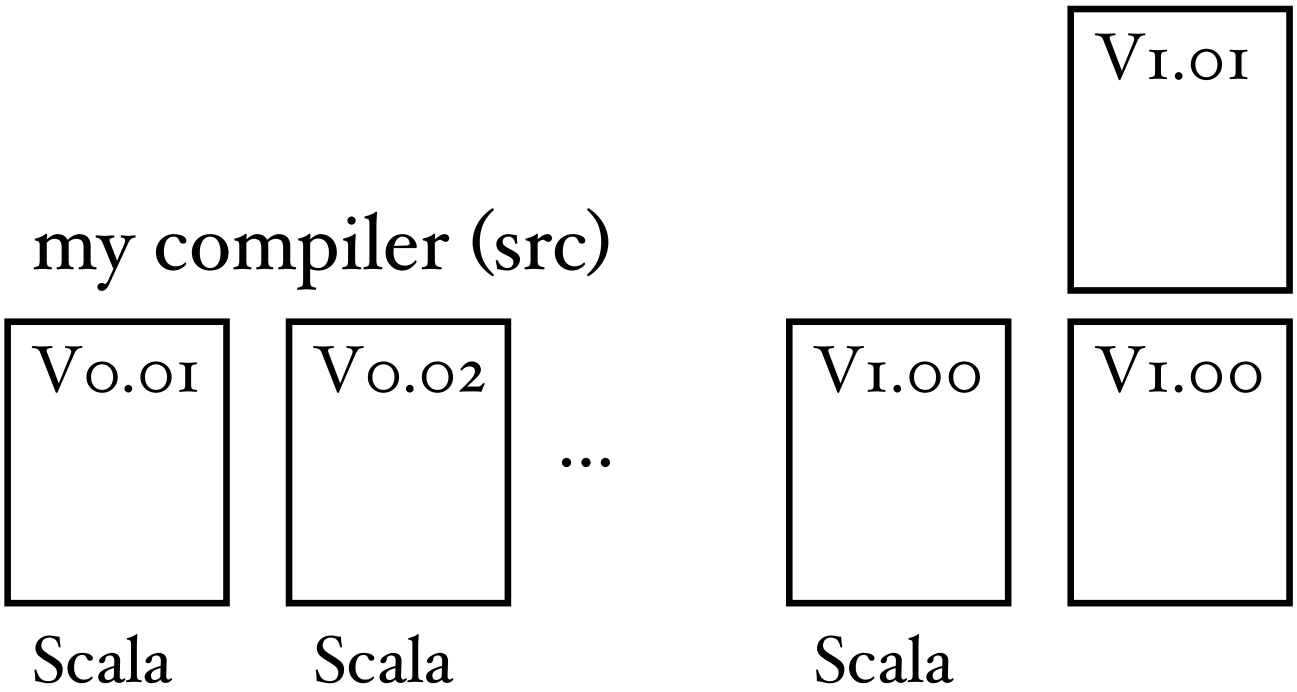
Scala

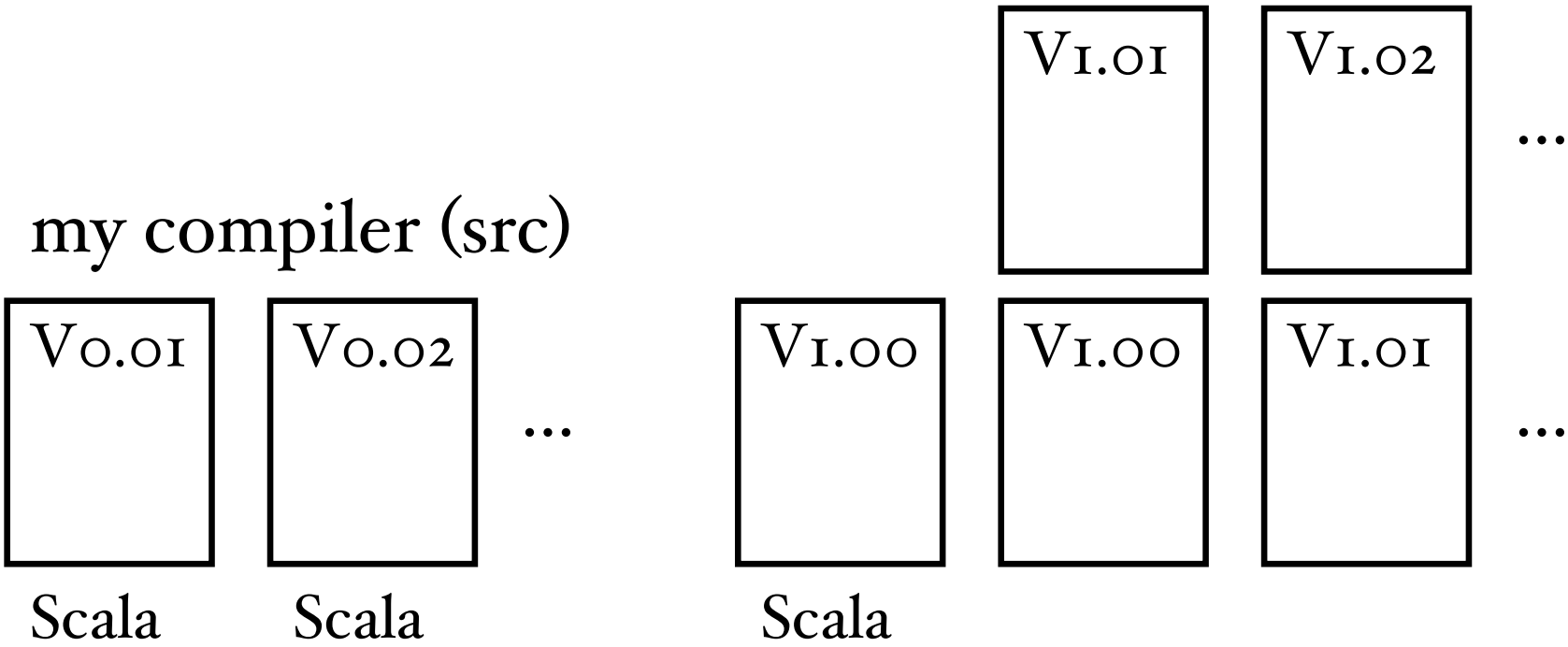
host language

my compiler (src)



host language


V1.01
my compiler (src)
V0.01
V0.02
...
V1.00
V1.00
Scala
Scala
Scala
host language

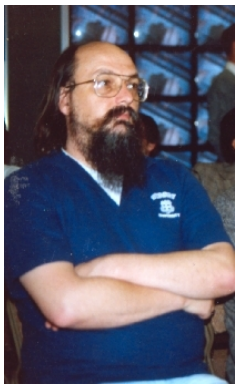

V1.01
V1.02
...
my compiler (src)
V0.01
V0.02
...
V1.00
V1.00
V1.01
...
Scala
Scala
Scala
host language

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.
- Therefore in safety-critical systems it is important to rely on only a very small TCB.

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.
- Therefore in safety-critical systems it is important to rely on only a very small TCB.

While-Language

$$
\begin{array}{lcl}
\mathit{Stmt} & \rightarrow & \text{skip} \\
 & | & \mathit{Id} := \mathit{AExp} \\
 & | & \text{if } \mathit{BExp} \text{ then } \mathit{Block} \text{ else } \mathit{Block} \\
 & | & \text{while } \mathit{BExp} \text{ do } \mathit{Block} \\
 & | & \text{write } \mathit{Id} \\
\mathit{Stmts} & \rightarrow & \mathit{Stmt} \,;\, \mathit{Stmts} \\
 & | & \mathit{Stmt} \\
\mathit{Block} & \rightarrow & \{\mathit{Stmts}\} \\
 & | & \mathit{Stmt} \\
\mathit{AExp} & \rightarrow & \ldots \\
\mathit{BExp} & \rightarrow & \ldots
\end{array}
$$

Fibonacci Numbers

```
1  /* Fibonnacci numbers implemented in
2     the WHILE language */
3
4  write "Input a number ";
5  read n;
6  x := 0;    // start values
7  y := 1;
8  while n > 0 do {
9    temp := y;
10   y := x + y;
11   x := temp;
12   n := n - 1   // decrement counter
13 };
14 write "Result ";
15 write y
```

Interpreter

$$
\begin{array}{lcl}
\text{eval}(n, E) & \stackrel{\text{def}}{=} & n \\
\text{eval}(x, E) & \stackrel{\text{def}}{=} & E(x) \quad \text{lookup } x \text{ in } E \\
\text{eval}(a_1 + a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) + \text{eval}(a_2, E) \\
\text{eval}(a_1 - a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) - \text{eval}(a_2, E) \\
\text{eval}(a_1 * a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) * \text{eval}(a_2, E) \\
 & & \\
\text{eval}(a_1 = a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) = \text{eval}(a_2, E) \\
\text{eval}(a_1 != a_2, E) & \stackrel{\text{def}}{=} & \neg(\text{eval}(a_1, E) = \text{eval}(a_2, E)) \\
\text{eval}(a_1 < a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) < \text{eval}(a_2, E)
\end{array}
$$

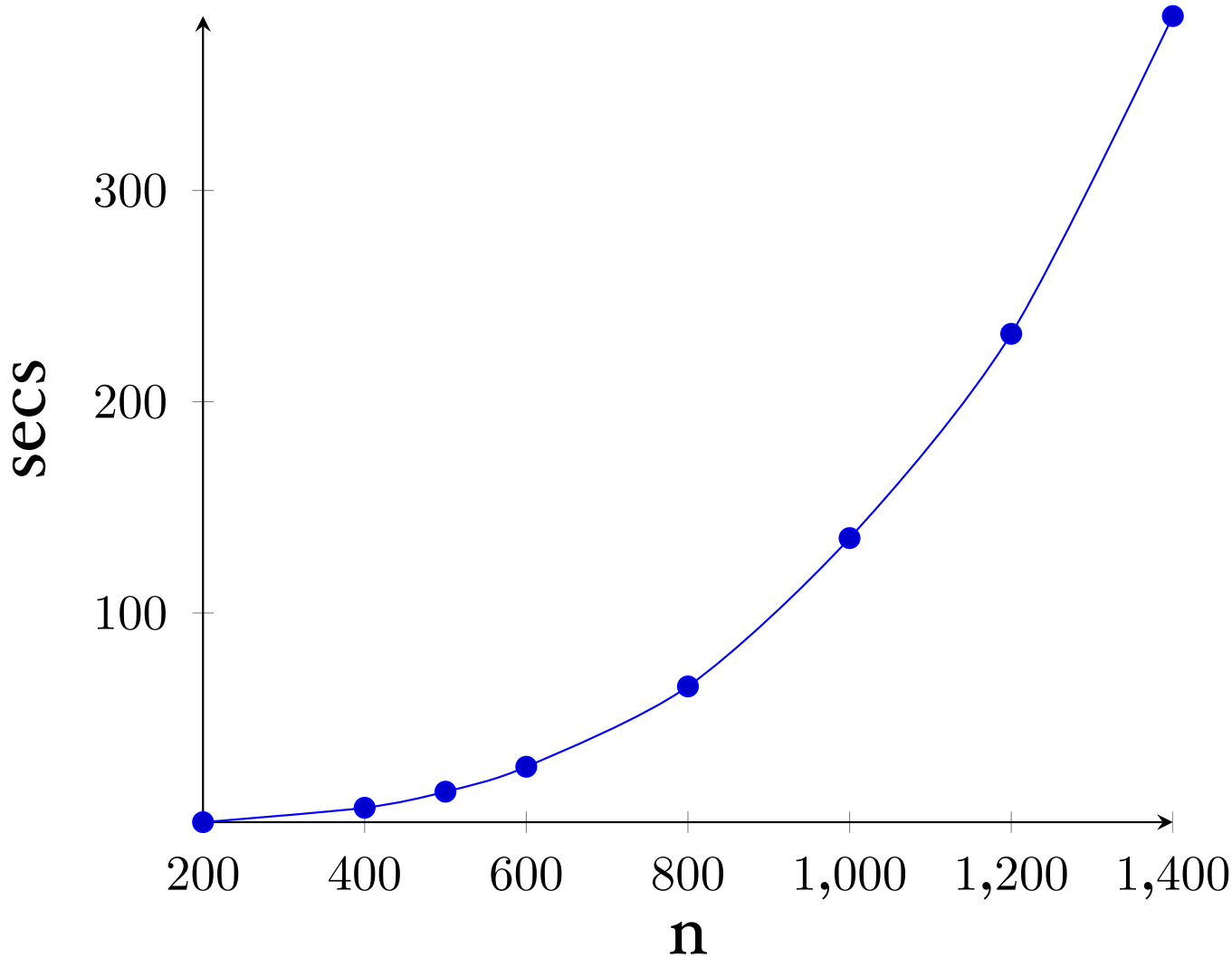
Interpreter (2)

$$
\begin{array}{lcl}
\text{eval}(\text{skip}, E) & \stackrel{\text{def}}{=} & E \\
\text{eval}(x := a, E) & \stackrel{\text{def}}{=} & E(x \mapsto \text{eval}(a, E)) \\
\text{eval}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) & \stackrel{\text{def}}{=} & \\
\quad \text{if eval}(b, E) \text{ then eval}(cs_1, E) & & \\
\quad \text{else eval}(cs_2, E) & & \\
\text{eval}(\text{while } b \text{ do } cs, E) & \stackrel{\text{def}}{=} & \\
\quad \text{if eval}(b, E) & & \\
\quad \text{then eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E)) & & \\
\quad \text{else } E & & \\
\text{eval}(\text{write } x, E) & \stackrel{\text{def}}{=} & \{\ \text{println}(E(x))\ ;\ E\ \}
\end{array}
$$

Test Program

```
1  start := 1000;    // start value
2  x := start;
3  y := start;
4  z := start;
5  while 0 < x do {
6   while 0 < y do {
7    while 0 < z do { z := z - 1 };
8    z := start;
9    y := y - 1
10  };
11  y := start;
12  x := x - 1
13 };
```

Interpreted Code



Java Virtual Machine

- introduced in 1995
- is a stack-based VM (like Postscript, CLR of .Net)
- contains a JIT compiler
- many languages take advantage of JVM's infrastructure (JRE)
- is garbage collected $\Rightarrow$ no buffer overflows
- some languages compiled to the JVM: Scala, Clojure...

Compiling AExps

1 + 2

ldc 1
ldc 2
iadd

Compiling AExps

1 + 2 + 3

```
ldc 1
ldc 2
iadd
ldc 3
iadd
```

Compiling AExps

$1 + (2 + 3)$

```
ldc 1
ldc 2
ldc 3
iadd
iadd
```

Compiling AExps

1 + (2 + 3)

```
ldc 1
ldc 2
ldc 3
iadd
iadd
```

dadd, fadd, ladd, ...

Compiling AExps

$$\begin{array}{lcl}
\text{compile}(n) & \stackrel{\text{def}}{=} & \text{ldc } n \\
\text{compile}(a_1 + a_2) & \stackrel{\text{def}}{=} & \\
\quad \text{compile}(a_1) \text{ @ compile}(a_2) \text{ @ iadd} & & \\
\text{compile}(a_1 - a_2) & \stackrel{\text{def}}{=} & \\
\quad \text{compile}(a_1) \text{ @ compile}(a_2) \text{ @ isub} & & \\
\text{compile}(a_1 * a_2) & \stackrel{\text{def}}{=} & \\
\quad \text{compile}(a_1) \text{ @ compile}(a_2) \text{ @ imul} & &
\end{array}$$

Compiling AExps

$$\begin{array}{lcl}
\text{compile}(n) & \stackrel{\text{def}}{=} & \text{ldc } n \\
\text{compile}(a_1 + a_2) & \stackrel{\text{def}}{=} & \\
\quad \text{compile}(a_1) \text{ @ compile}(a_2) \text{ @ iadd} & & \\
\text{compile}(a_1 - a_2) & \stackrel{\text{def}}{=} & \\
\quad \text{compile}(a_1) \text{ @ compile}(a_2) \text{ @ isub} & & \\
\text{compile}(a_1 * a_2) & \stackrel{\text{def}}{=} & \\
\quad \text{compile}(a_1) \text{ @ compile}(a_2) \text{ @ imul} & &
\end{array}$$

Compiling AExps

$1 + 2 * 3 + (4 - 3)$

```
ldc 1
ldc 2
ldc 3
imul
ldc 4
ldc 3
isub
iadd
iadd
```

Variables

$$\boldsymbol{x} := 5 + \boldsymbol{y} * 2$$

Variables

$$\boldsymbol{x} := 5 + \boldsymbol{y} * 2$$

- lookup: iload $\boldsymbol{index}$
- store: istore $\boldsymbol{index}$

Variables

$$x := 5 + y * 2$$

- lookup: iload $index$
- store: istore $index$

while compilating we have to maintain a map between our identifiers and the Java bytecode indices

$$\text{compile}(a, E)$$

Compiling AExps

$$\text{compile}(n, E) \stackrel{\text{def}}{=} \text{ldc } n$$

$$\text{compile}(a_1 + a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) \mathbin{@} \text{compile}(a_2.E) \mathbin{@} \text{iadd}$$

$$\text{compile}(a_1 - a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) \mathbin{@} \text{compile}(a_2, E) \mathbin{@} \text{isub}$$

$$\text{compile}(a_1 * a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) \mathbin{@} \text{compile}(a_2, E) \mathbin{@} \text{imul}$$

$$\text{compile}(x, E) \stackrel{\text{def}}{=} \text{iload } E(x)$$

Compiling AExps

$$\text{compile}(n, E) \stackrel{\text{def}}{=} \text{ldc } n$$

$$\text{compile}(a_1 + a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) \mathbin{@} \text{compile}(a_2.E) \mathbin{@} \text{iadd}$$

$$\text{compile}(a_1 - a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) \mathbin{@} \text{compile}(a_2, E) \mathbin{@} \text{isub}$$

$$\text{compile}(a_1 * a_2, E) \stackrel{\text{def}}{=} \text{compile}(a_1, E) \mathbin{@} \text{compile}(a_2, E) \mathbin{@} \text{imul}$$

$$\text{compile}(x, E) \stackrel{\text{def}}{=} \text{iload } E(x)$$

Compiling Statements

We return a list of instructions and an environment for the variables

$$\text{compile}(\text{skip}, E) \stackrel{\text{def}}{=} (Nil, E)$$

$$\text{compile}(x := a, E) \stackrel{\text{def}}{=} (\text{compile}(a, E) \mathbin{@} \text{istore } index, E(x \mapsto index))$$

where $index$ is $E(x)$ if it is already defined, or if it is not then the largest index not yet seen

Compiling AExps

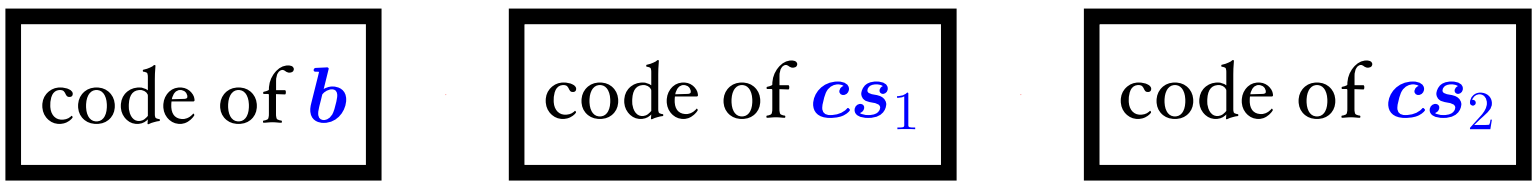
$x := x + 1$

iload n_x
ldc 1
iadd
istore n_x

where n_x is the index corresponding to the variable x

Compiling Ifs

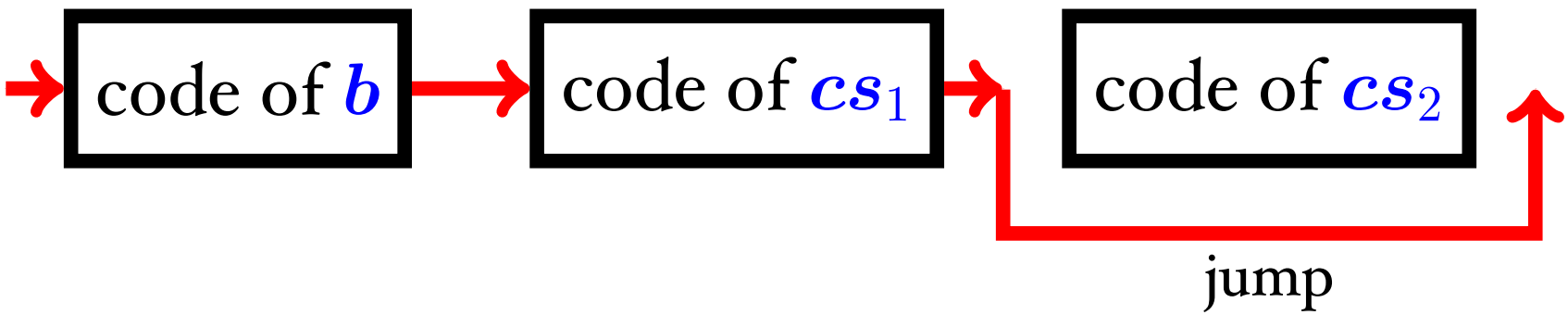
if b then cs_1 else cs_2



Compiling Ifs

if b then cs_1 else cs_2

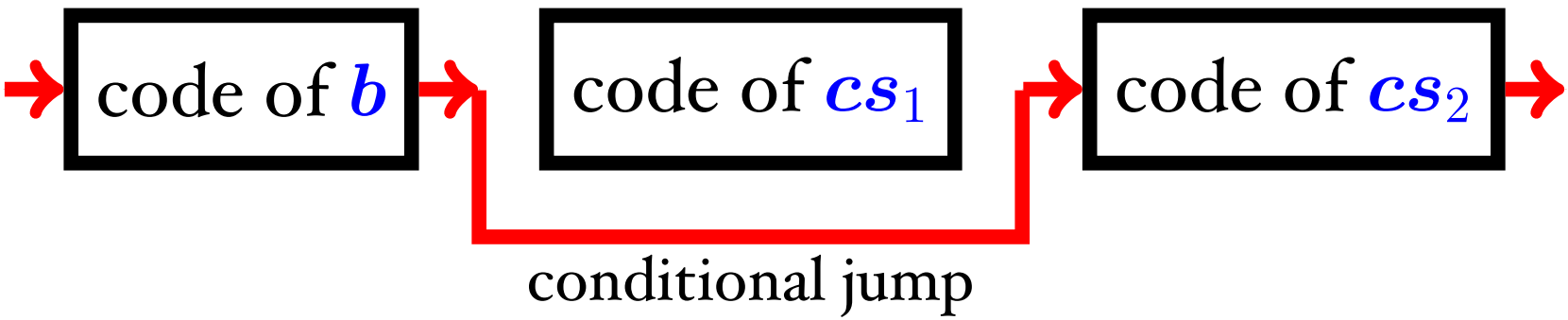
Case **True**:



Compiling Ifs

if b then cs_1 else cs_2

Case **False**:



Conditional Jumps

- if_icmpeq *label* if two ints are equal, then jump
- if_icmpne *label* if two ints aren't equal, then jump
- if_icmpge *label* if one int is greater or equal then another, then jump

  ...

Conditional Jumps

- if_icmpeq *label* if two ints are equal, then jump
- if_icmpne *label* if two ints aren't equal, then jump
- if_icmpge *label* if one int is greater or equal then another, then jump

...

L_1:
  if_icmpeq L_2
  iload 1
  ldc 1
  iadd
  if_icmpeq L_1
L_2:

Conditional Jumps

- if_icmpeq *label* if two ints are equal, then jump
- if_icmpne *label* if two ints aren't equal, then jump
- if_icmpge *label* if one int is greater or equal then another, then jump

...

```
L1:
   if_icmpeq L2
   iload 1
   ldc 1
   iadd
   if_icmpeq L1
L2:
```

labels must be unique

Compiling BExps

$$a_1 = a_2$$

$$\text{compile}(a_1 = a_2, E, lab) \stackrel{\text{def}}{=}$$
$$\text{compile}(a_1, E) \;@\; \text{compile}(a_2, E) \;@\; \text{if\_icmpne } lab$$

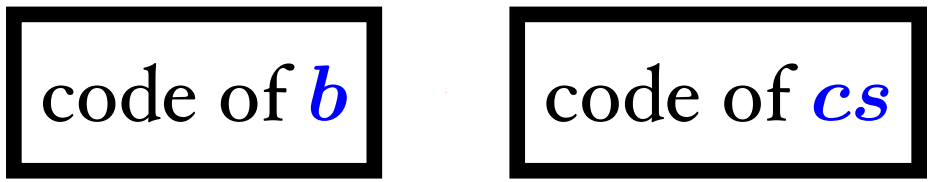
Compiling Ifs

if b then cs_1 else cs_2

$$\begin{array}{l}
\text{compile}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) \stackrel{\text{def}}{=} \\
\quad l_{ifelse} \text{ (fresh label)} \\
\quad l_{ifend} \text{ (fresh label)} \\
\quad (is_1, E') = \text{compile}(cs_1, E) \\
\quad (is_2, E'') = \text{compile}(cs_2, E') \\
\quad (\text{compile}(b, E, l_{ifelse}) \\
\quad\ @\ is_1 \\
\quad\ @\ \text{goto } l_{ifend} \\
\quad\ @\ l_{ifelse} : \\
\quad\ @\ is_2 \\
\quad\ @\ l_{ifend} :, E'')
\end{array}$$

Compiling Whiles

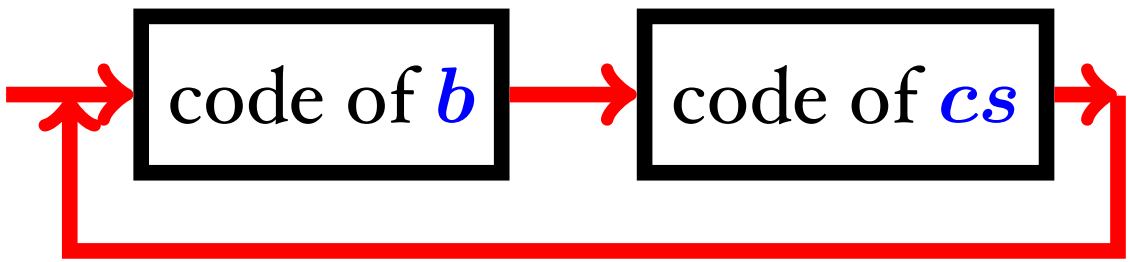
while b do cs



Compiling Whiles

while b do cs

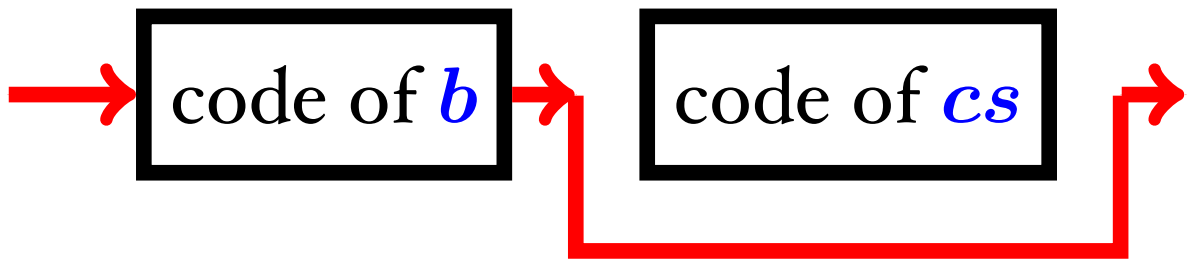
Case **True**:



Compiling Whiles

while b do cs

Case **False**:



Compiling Whiles

while b do cs

$$\begin{array}{l}\text{compile}(\text{while } b \text{ do } cs, E) \stackrel{\text{def}}{=} \\ \quad l_{wbegin} \text{ (fresh label)} \\ \quad l_{wend} \text{ (fresh label)} \\ \quad (is, E') = \text{compile}(cs_1, E) \\ \quad (l_{wbegin} : \\ \quad\ @\ \text{compile}(b, E, l_{wend}) \\ \quad\ @\ is \\ \quad\ @\ \text{goto } l_{wbegin} \\ \quad\ @\ l_{wend} :, E')\end{array}$$

Compiling Writes

write x

```
.method public static write(I)V        (library function)
  .limit locals 5
  .limit stack 5
  iload 0
  getstatic java/lang/System/out Ljava/io/PrintStream;
  swap
  invokevirtual java/io/PrintStream/println(I)V
  return
.end method
```

iload $E(x)$
invokestatic write(I)V

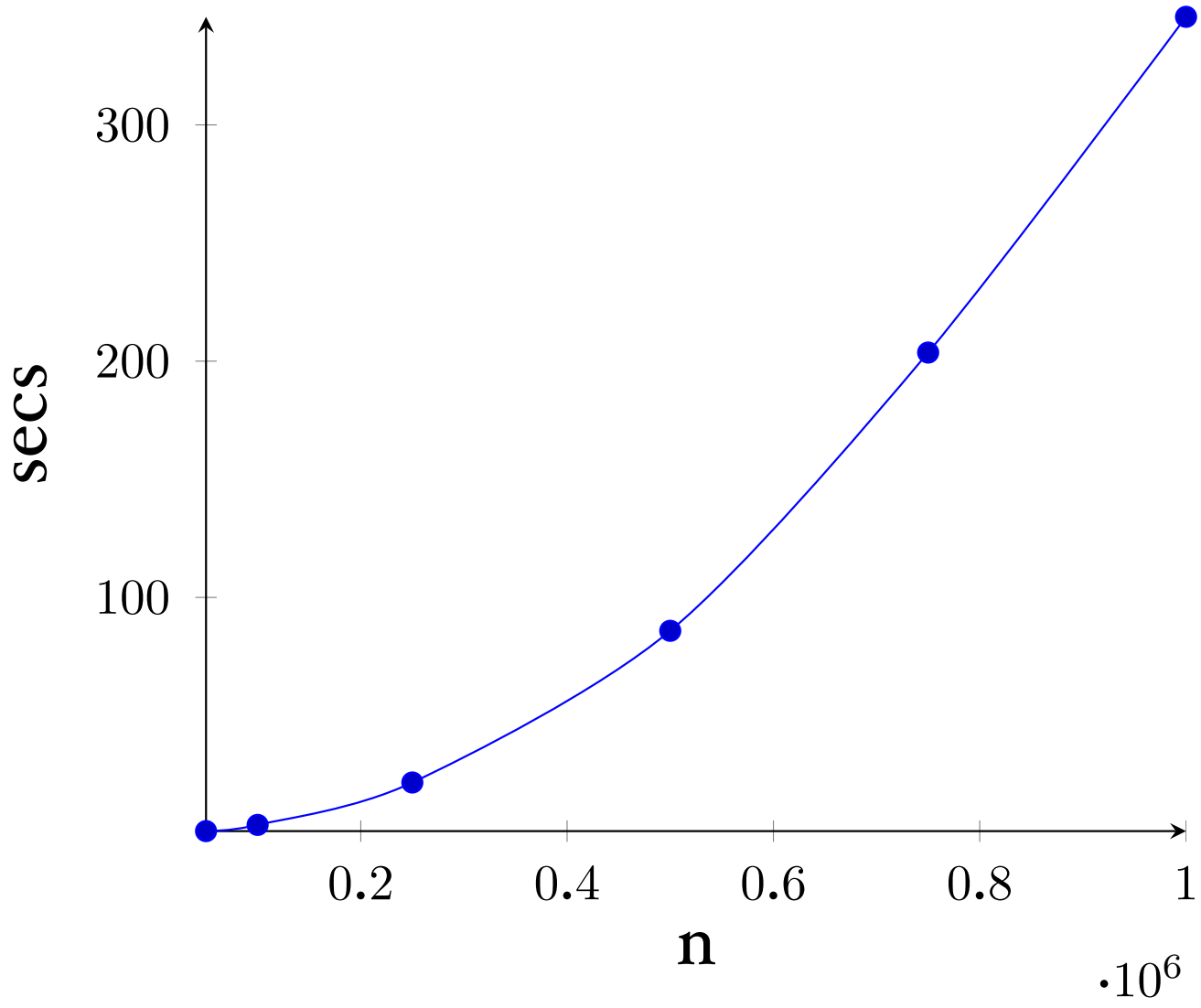
```
.class public XXX.XXX
.super java/lang/Object

.method public <init>()V
  aload_0
  invokenonvirtual java/lang/Object/<init>()V
  return
.end method

.method public static main([Ljava/lang/String;)V
  .limit locals 200
  .limit stack 200

(here comes the compiled code)

  return
.end method
```

Next Compiler Phases

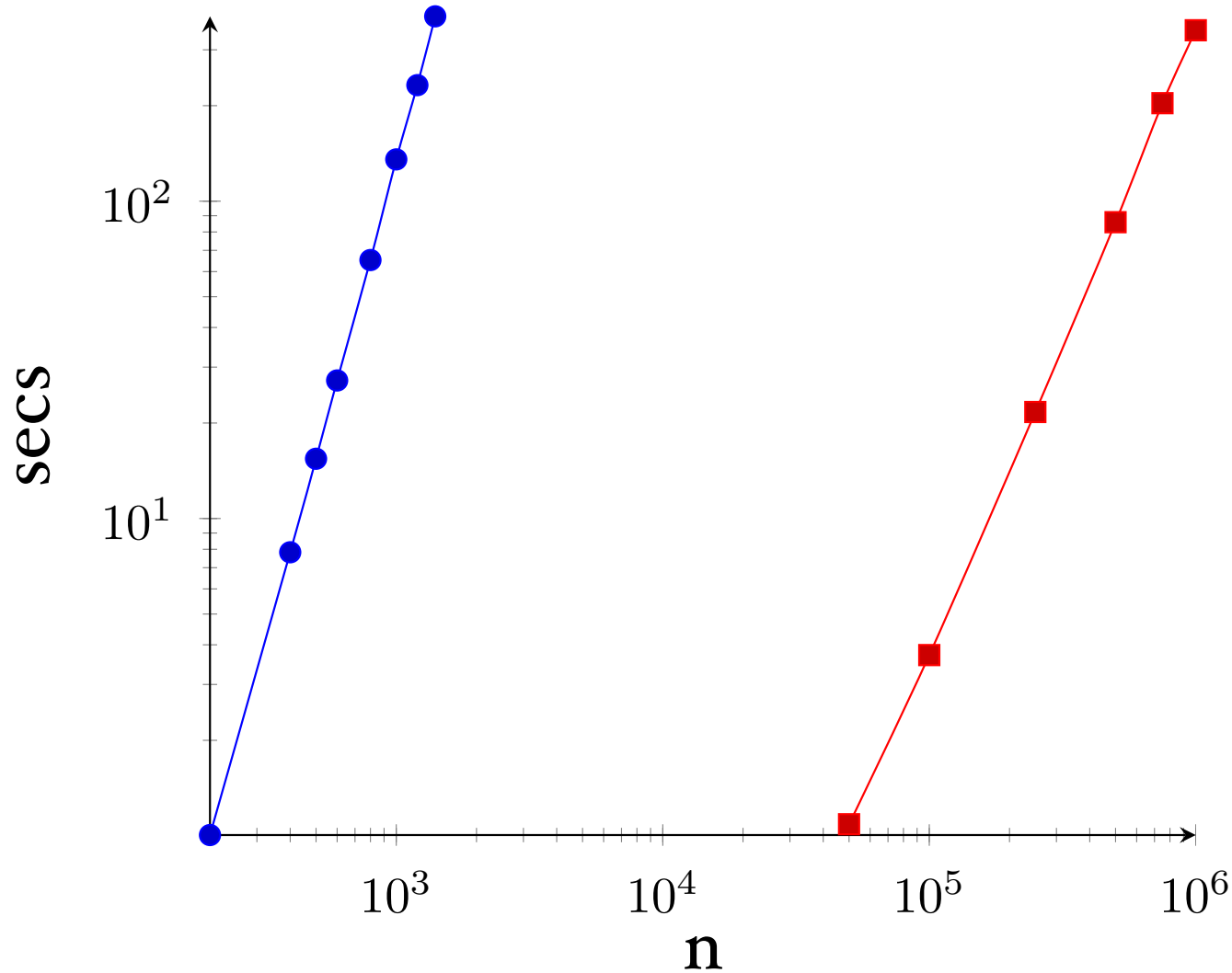
- assembly $\Rightarrow$ byte code (class file)
- labels $\Rightarrow$ absolute or relative jumps

- javap is a disassembler for class files

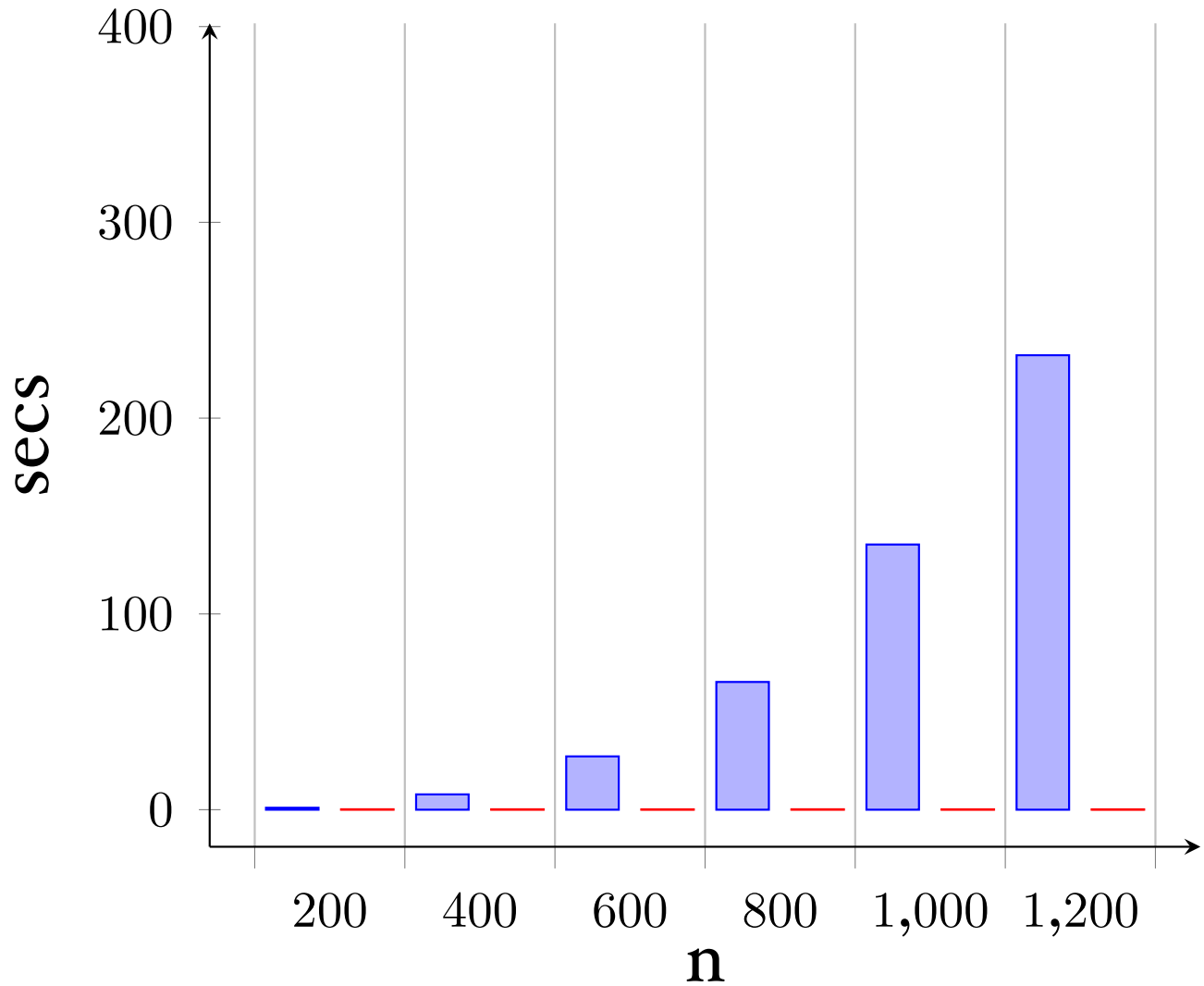
Compiled Code



Compiled vs. Interpreted Co



Compiled vs. Interpreted Co



What Next

- register spilling
- dead code removal
- loop optimisations
- instruction selection
- type checking
- concurrency
- fuzzy testing
- verification

- GCC, LLVM, tracing JITs