

Automata and Formal Languages (9)

Email: christian.urban at kcl.ac.uk

Office: SI.27 (1st floor Strand Building)

Slides: KEATS (also home work is there)

Functional Programming

```
def fib(n) = if n == 0 then 0
             else if n == 1 then 1
             else fib(n - 1) + fib(n - 2);
```

```
def fact(n) = if n == 0 then 1 else n * fact(n - 1);
```

```
def ack(m, n) = if m == 0 then n + 1
                 else if n == 0 then ack(m - 1, 1)
                 else ack(m - 1, ack(m, n - 1));
```

```
def gcd(a, b) = if b == 0 then a else gcd(b, a % b);
```

Fun Grammar

$\langle Exp \rangle ::= \langle Var \rangle \mid \langle Num \rangle$
 $\mid \langle Exp \rangle + \langle Exp \rangle \mid \dots \mid (\langle Exp \rangle)$
 $\mid \mathbf{if} \langle BExp \rangle \mathbf{then} \langle Exp \rangle \mathbf{else} \langle Exp \rangle$
 $\mid \mathbf{write} \langle Exp \rangle$
 $\mid \langle Exp \rangle ; \langle Exp \rangle$
 $\mid FunName (\langle Exp \rangle, \dots, \langle Exp \rangle)$

$\langle BExp \rangle ::= \dots$

$\langle Decl \rangle ::= \langle Def \rangle ; \langle Decl \rangle \mid \langle Exp \rangle$

$\langle Def \rangle ::= \mathbf{def} FunName (x_1, \dots, x_n) = \langle Exp \rangle$

Abstract Syntax Trees

abstract class Exp

abstract class BExp

abstract class Decl

case class Var(s: String) **extends** Exp

case class Num(i: Int) **extends** Exp

case class Aop(o: String, a1: Exp, a2: Exp) **extends** Exp

case class If(a: BExp, e1: Exp, e2: Exp) **extends** Exp

case class Write(e: Exp) **extends** Exp

case class Sequ(e1: Exp, e2: Exp) **extends** Exp

case class Call(name: String, args: List[Exp]) **extends** Exp

case class Bop(o: String, a1: Exp, a2: Exp) **extends** BExp

case class Def(name: String,
 args: List[String],
 body: Exp) **extends** Decl

case class Main(e: Exp) **extends** Decl

Arithmetic Functions

Compilation of some arithmetic functions:

`Aop("+", a1, a2) ⇒ ...iadd`

`Aop("-", a1, a2) ⇒ ...isub`

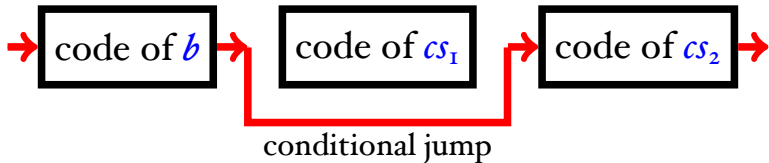
`Aop("*", a1, a2) ⇒ ...imul`

`Aop("/", a1, a2) ⇒ ...idiv`

`Aop("%", a1, a2) ⇒ ...irem`

Boolean Expressions

Compilation of Boolean expressions:



`Bop("==", a1, a2) ⇒ ...if_icmpne...`

`Bop("!=", a1, a2) ⇒ ...if_icmpeq...`

`Bop("<", a1, a2) ⇒ ...if_icmpge...`

`Bop("<=", a1, a2) ⇒ ...if_icmpgt...`

Sequences

Compiling `arg1 ; arg2`:

`...arg1...`

pop

`...arg1...`

Write

Compiling call to write(arg):

...arg...

dup

invokestatic XXX/XXX/write(I)V

needs a helper function

Function Definitions

```
.method public static write(I)V
  .limit locals 1
  .limit stack 2
  getstatic java/lang/System/out Ljava/io/PrintStream;
  iload 0
  invokevirtual java/io/PrintStream/println(I)V
  return
.end method
```

We will need for definitions

```
.method public static f (I...I)I
  .limit locals ??
  .limit stack ??
  ??
.end method
```

Stack Estimation

```
def max_stack_exp(e: Exp): Int = e match {  
  case Call(_, args) => args.map(max_stack_exp).sum  
  case If(a, e1, e2) => max_stack_bexp(a) +  
    (List(max_stack_exp(e1), max_stack_exp(e2))).max  
  case Write(e) => max_stack_exp(e) + 1  
  case Var(_) => 1  
  case Num(_) => 1  
  case Aop(_, a1, a2) =>  
    max_stack_exp(a1) + max_stack_exp(a2)  
  case Sequ(e1, e2) =>  
    List(max_stack_exp(e1), max_stack_exp(e2)).max  
}
```

```
def max_stack_bexp(e: BExp): Int = e match {  
  case Bop(_, a1, a2) =>  
    max_stack_exp(a1) + max_stack_exp(a2)  
}
```

Successor Function

```
.method public static suc(I)I
.limit locals 1
.limit stack 3
  iload 0
  ldc 1
  iadd
  ireturn
.end method
```

```
def suc(x) = x + 1;
```

Addition Function

```
.method public static add(II)I
.limit locals 2
.limit stack 6
  iload 0
  ldc 0
  if_icmpne If_else
  iload 1
  goto If_end
If_else:
  iload 0
  ldc 1
  isub
  iload 1
  invokestatic XXX/XXX/add(II)I
  invokestatic XXX/XXX/suc(I)I
If_end:
  ireturn
.end method
```

```
def add(x, y) =
  if x == 0 then y
  else suc(add(x - 1, y));
```

Factorial

```
.method public static fact(II)I
.limit locals 2
.limit stack 6
  iload 0
  ldc 0
  if_icmpne If_else_2
  iload 1
  goto If_end_3
If_else_2:
  iload 0
  ldc 1
  isub
  iload 0
  iload 1
  imul
  invokestatic fact/fact/fact(II)I
If_end_3:
  ireturn
.end method
```

```
def fact(n, acc) =
  if n == 0 then acc
  else fact(n - 1, n * acc);
```

```
.method public static fact(II)I
```

```
.limit locals 2
```

```
.limit stack 7
```

```
fact_Start:
```

```
  iload 0
```

```
  ldc 0
```

```
  if_icmpne If_else_2
```

```
  iload 1
```

```
  goto If_end_3
```

```
If_else_2:
```

```
  iload 0
```

```
  ldc 1
```

```
  isub
```

```
  iload 0
```

```
  iload 1
```

```
  imul
```

```
  istore 1
```

```
  istore 0
```

```
  goto fact_Start
```

```
If_end_3:
```

```
  ireturn
```

```
.end method
```

```
def fact(n, acc) =  
  if n == 0 then acc  
  else fact(n - 1, n * acc);
```

Tail Recursion

A call to `f(args)` is usually compiled as

```
args onto stack  
invokestatic .../f
```

Tail Recursion

A call to $f(\text{args})$ is usually compiled as

```
args onto stack  
invokestatic .../f
```

A call is in tail position provided:

- if Bexp then **Exp** else **Exp**
- Exp ; **Exp**
- Exp op Exp

then a call $f(\text{args})$ can be compiled as

```
prepare environment  
jump to start of function
```


Tail Recursive Call

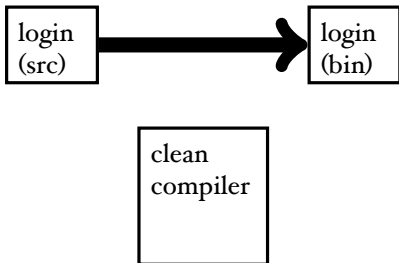
```
def compile_expT(a: Exp, env: Mem, name: String): Instrs =
  ...
  case Call(n, args) => if (name == n)
  {
    val stores = args.zipWithIndex.map
      { case (x, y) => "istore " + y.toString + "\n" }
    args.flatMap(a => compile_expT(a, env, "")) ++
    stores.reverse ++
    List ("goto " + n + "_Start\n")
  }
  else
  {
    val is = "I" * args.length
    args.flatMap(a => compile_expT(a, env, "")) ++
    List ("invokestatic XXX/XXX/" + n + "(" + is + ")I\n")
  }
```

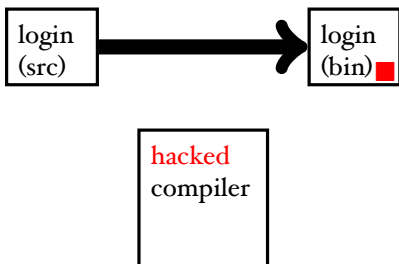
**Using a compiler,
how can you mount the
perfect attack against a system?**

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





my compiler (src)



Scala

host language

my compiler (src)

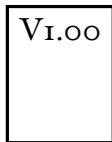


Scala



Scala

...



Scala

host language

my compiler (src)

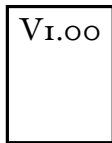


Scala

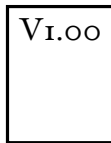
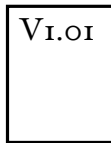


Scala

...

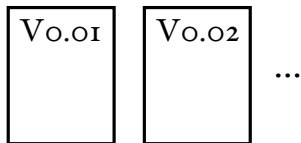


Scala



host language

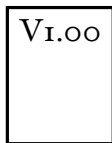
my compiler (src)



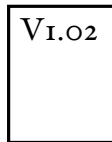
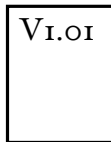
Scala

Scala

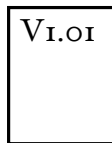
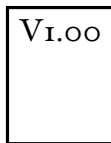
host language



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.

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

Hacking Compilers



Ken Thompson
Turing Award, 1983



- 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.