Compilers and Formal Languages

Email: christian.urban at kcl.ac.uk
Slides & Progs: KEATS (also homework is there)

1 Introduction, Languages	6 While-Language
2 Regular Expressions, Derivatives	7 Compilation, JVM
3 Automata, Regular Languages	8 Compiling Functional Languages
4 Lexing, Tokenising	9 Optimisations
5 Grammars, Parsing	10 LLVM

The Fun Language

```
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);
```

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 6
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
```

```
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(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(args)` can be compiled as

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

Tail Recursive Call

```scala
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) => i"istore $y" }

    args.map(a => compile_expT(a, env, "")).mkString ++
    stores.reverse.mkString ++
    i"goto ${n}_Start"
  } else {
    val is = "I" * args.length
    args.map(a => compile_expT(a, env, "")).mkString ++
    i"invokestatic XXX/XXX/${n}(${is})I"
  }
```

Factorial Funct. on the JVM

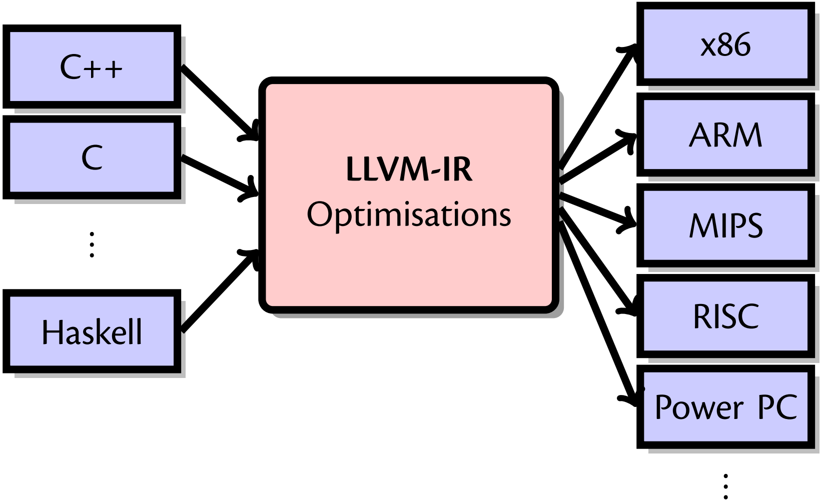
```
.method public static facT(II)I
.limit locals 2
.limit stack 6
  iload 0
  ldc 0
  if_icmpne If_else_2
  iload 1
  goto If_end_3
If_else_2:
  iload 0
  ldc 1
  isub
  iload 0
  iload 1
  imul
  invokestatic fact/fact/facT(II)I
If_end_3:
  ireturn
.end method
```

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

LLVM

- Chris Lattner, Vikram Adve (started in 2000)
- Apple hired Lattner in 2006
- modular architecture, LLVM-IR
- `lli` and `llc`

LLVM: Overview



LLVM-IR

```
define i32 @fact (i32 %n) {
   %tmp_19 = icmp eq i32  %n, 0
   br i1 %tmp_19, label %if_br_23, label %else_br_24

if_br_23:
   ret i32 1

else_br_24:
   %tmp_21 = sub i32  %n, 1
   %tmp_22 = call i32 @fact (i32 %tmp_21)
   %tmp_20 = mul i32  %n, %tmp_22
   ret i32 %tmp_20
}
```

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

LLVM Types

`boolean`	`i1`
`byte`	`i8`
`short`	`i16`
`char`	`i16`
`integer`	`i32`
`long`	`i64`
`float`	`float`
`double`	`double`
`*_`	`pointer to`
`**_`	`pointer to a pointer to`
`[_]`	`arrays of`

LLVM-IR Instructions

```
br i1 %var, label %if_br, label %else_br

icmp eq i32  %x, %y     ; for equal
icmp sle i32 %x, %y     ;   signed less or equal
icmp slt i32 %x, %y     ;   signed less than
icmp ult i32 %x, %y     ;   unsigned less than

%var = call i32 @foo(...args...)
```

SSA Format

$$(1 + a) + (3 + (b * 5))$$

```
tmp0 = add 1 a
tmp1 = mul b 5
tmp2 = add 3 tmp1
tmp3 = add tmp0 tmp2
```

Static Single Assignment

Abstract Syntax Trees

```scala
// Fun language (expressions)
abstract class Exp
abstract class BExp

case class Call(name: String, args: List[Exp]) extends Exp
case class If(a: BExp, e1: Exp, e2: Exp) extends Exp
case class Write(e: Exp) extends Exp
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 Sequence(e1: Exp, e2: Exp) extends Exp
case class Bop(o: String, a1: Exp, a2: Exp) extends BExp
```

K-(Intermediate)Language

```
abstract class KExp
abstract class KVal

// K-Values
case class KVar(s: String) extends KVal
case class KNum(i: Int) extends KVal
case class Kop(o: String, v1: KVal, v2: KVal) extends KVal
case class KCall(o: String, vrs: List[KVal]) extends KVal
case class KWrite(v: KVal) extends KVal

// K-Expressions
case class KIf(x1: String, e1: KExp, e2: KExp) extends KExp
case class KLet(x: String, v: KVal, e: KExp) extends KExp
case class KReturn(v: KVal) extends KExp
```

KLet

```
tmp0 = add 1 a
tmp1 = mul b 5
tmp2 = add 3 tmp1
tmp3 = add tmp0 tmp2
```

```
  KLet tmp0 , add 1 a in
   KLet tmp1 , mul b 5 in
    KLet tmp2 , add 3 tmp1 in
     KLet tmp3 , add tmp0 tmp2 in
      ...
```

```
case class KLet(x: String, e1: KVal, e2: KExp)
```

KLet

```
tmp0 = add 1 a
tmp1 = mul b 5
tmp2 = add 3 tmp1
tmp3 = add tmp0 tmp2
```

```
  let tmp0 = add 1 a in
   let tmp1 = mul b 5 in
    let tmp2 = add 3 tmp1 in
     let tmp3 = add tmp0 tmp2 in
      ...
```

```
case class KLet(x: String, e1: KVal, e2: KExp)
```

CPS-Translation

```
def CPS(e: Exp)(k: KVal => KExp) : KExp =
  e match { ... }
```

the continuation k can be thought of:

```
let tmp0 = add 1 a in
let tmp1 = mul □ 5 in
let tmp2 = add 3 tmp1 in
let tmp3 = add tmp0 tmp2 in
  KReturn tmp3
```

CPS-Translation

```
def CPS(e: Exp)(k: KVal => KExp) : KExp =
  e match {
     case Var(s) => k(KVar(s))
     case Num(i) => k(KNum(i))
     ...
  }
```

```
let tmp0 = add 1 a in
let tmp1 = mul □ 5 in
let tmp2 = add 3 tmp1 in
let tmp3 = add tmp0 tmp2 in
  KReturn tmp3
```

CPS-Translation

```
def CPS(e: Exp)(k: KVal => KExp) : KExp = e match {
  case Aop(o, e1, e2) => {
    val z = Fresh("tmp")
    CPS(e1)(y1 =>
      CPS(e2)(y2 =>
                KLet(z, Kop(o, y1, y2), k(KVar(z)))))
  } ...
}
```

```
...
let z = op □y1 □y2
let tmp0 = add 1 a in
let tmp1 = mul z 5 in
let tmp2 = add 3 tmp1 in
let tmp3 = add tmp0 tmp2 in
  KReturn tmp3
```

CPS-Translation

```
def CPS(e: Exp)(k: KVal => KExp) : KExp =
  e match {
    case Sequence(e1, e2) =>
      CPS(e1)(_ => CPS(e2)(y2 => k(y2)))
     ...
  }
```

```
let tmp0 = add 1 a in
let tmp1 = mul □ 5 in
let tmp2 = add 3 tmp1 in
let tmp3 = add tmp0 tmp2 in
  KReturn tmp3
```

CPS-Translation

```
def CPS(e: Exp)(k: KVal => KExp) : KExp =
  e match {
    ...
    case If(Bop(o, b1, b2), e1, e2) => {
      val z = Fresh("tmp")
      CPS(b1)(y1 =>
        CPS(b2)(y2 =>
          KLet(z, Kop(o, y1, y2),
                KIf(z, CPS(e1)(k), CPS(e2)(k)))))
     }
    ...
  }
```

The Basic Language, 1980+

```
 5 LET S = 0
10 INPUT V
20 PRINT "Input number"
30 IF N = 0 THEN GOTO 99
40 FOR I = 1 TO N
45 LET S = S + V(I)
50 NEXT I
60 PRINT S/N
70 GOTO 5
99 END
```

"Spaghetti Code"

Target Specific ASM

```
llc -march=x86-64 fact.ll
llc -march=arm fact.ll
```

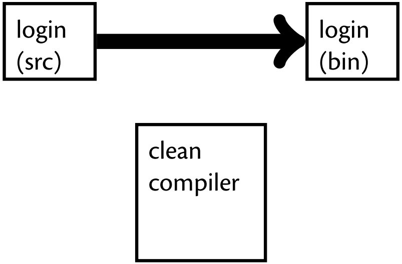
```
Intel:   xorl   %eax, %eax
ARM:     mov    pc, lr
```

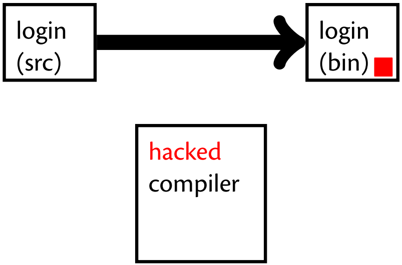
Using a compiler,
how can you mount the
perfect attack against a system?

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

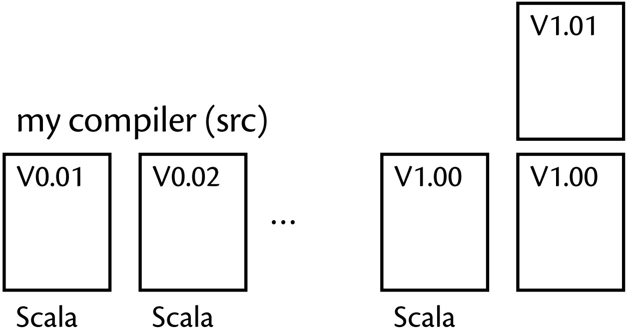
my compiler (src)

V0.01 V0.02 … V1.00

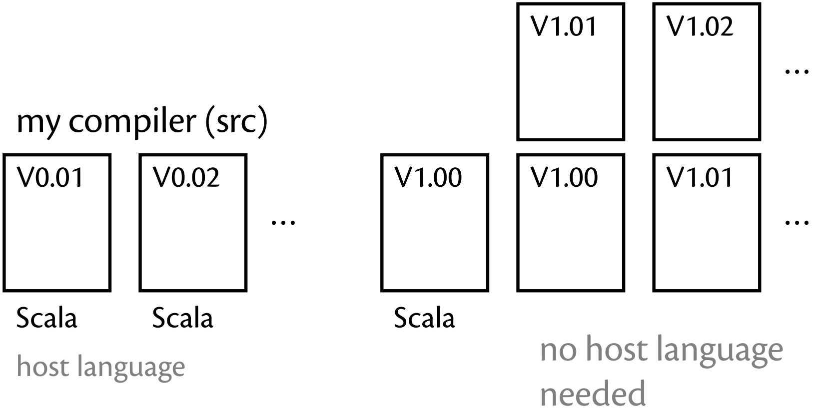
Scala Scala Scala

host language

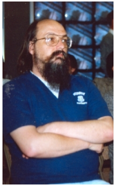
my compiler (src)



host language


my compiler (src)
V0.01
V0.02
...
V1.00
V1.01
V1.02
...
V1.00
V1.01
...
Scala
Scala
Scala
host language
no host language needed

Hacking Compilers

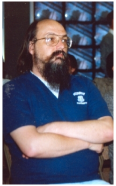


Ken Thompson
Turing Award, 1983

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

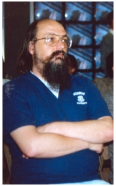


Ken Thompson
Turing Award, 198

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)*

… a Tro-
…ng any
…on will
…acks.

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.

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

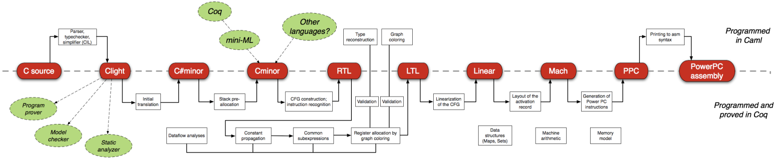
Theorem: The program is doing what it is supposed to be doing.

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."

Next Week

- Revision Lecture
- How many strings are in $L(a^*)$?

Next Week

- Revision Lecture
- How many strings are in $L(a^*)$?
- How many strings are in $L((a+b)^*)$?
  Are there more than in $L(a^*)$?