

# Security Engineering (9)

Email: christian.urban at kcl.ac.uk

Office: N7.07 (North Wing, Bush House)

Slides: KEATS (also homework is there)



# Old-Fashioned Eng. vs. CS

## **bridges:**

engineers can “look” at a bridge and have a pretty good intuition about whether it will hold up or not (redundancy; predictive theory)

## **code:**

programmers have very little intuition about their code; often it is too expensive to have redundancy; not “continuous”

# Trusting Computing Base

When considering whether a system meets some security objectives, it is important to consider which parts of that system are trusted in order to meet that objective (TCB).

# Trusting Computing Base

When considering whether a system meets some security objectives, it is important to consider which parts of that system are trusted in order to meet that objective (TCB).

The smaller the TCB, the less effort it takes to get some confidence that it is trustworthy, by doing a code review or by performing some (penetration) testing.

CPU, compiler, libraries, OS, NP  $\neq$  P, random number generator, ...

# 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.”

unfortunately attackers exploit bugs (Satan's computer vs Murphy's)

# Proving Programs to be Correct

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

**Proof ...**

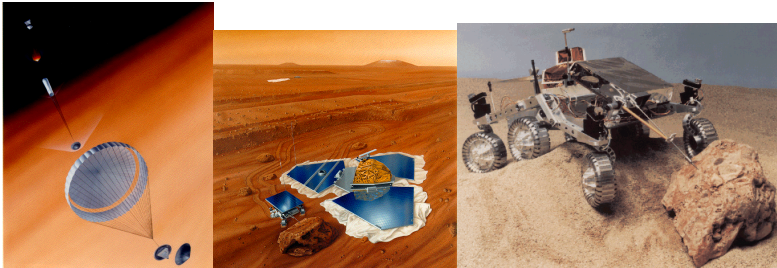
similarly

**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 (protocols, OS, ...).

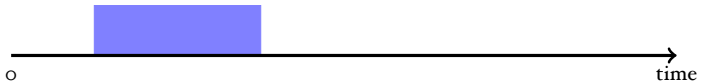
# Mars Pathfinder Mission 1997



- despite NASA's famous testing procedures, the lander crashed frequently on Mars
- a scheduling algorithm was not used in the OS



low priority



high priority



low priority



high priority



low priority



Scheduling: You want to avoid that a high priority process is starved indefinitely.

high priority



locked a resource

low priority



Scheduling: You want to avoid that a high priority process is starved indefinitely.

high priority



locked a resource

low priority



Scheduling: You want to avoid that a high priority process is starved indefinitely.

high priority



locked a resource

low priority

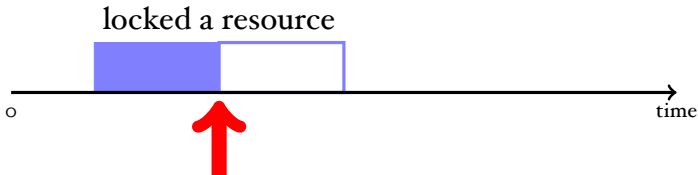


Scheduling: You want to avoid that a high priority process is starved indefinitely.

high priority



low priority



Scheduling: You want to avoid that a high priority process is starved indefinitely.

high priority



medium pr.



low priority

locked a resource



Scheduling: You want to avoid that a high priority process is starved indefinitely.



high priority



medium pr.



low priority

locked a resource



o

time



Scheduling: You want to avoid that a high priority process is starved indefinitely.

high priority



medium pr.



low priority

locked a resource



o

time



Scheduling: You want to avoid that a high priority process is starved indefinitely.

high priority



medium pr.



low priority

locked a resource



o

time



Scheduling: You want to avoid that a high priority process is starved indefinitely.

high priority

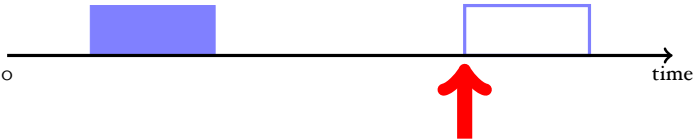


medium pr.



locked a resource

low priority



Scheduling: You want to avoid that a high priority process is starved indefinitely.

high priority



medium pr.

locked a resource

low priority



Scheduling: You want to avoid that a high priority process is starved indefinitely.

locked a resource

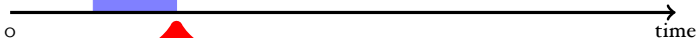
high priority



medium pr.



low priority



Scheduling: You want to avoid that a high priority process is starved indefinitely.

locked a resource

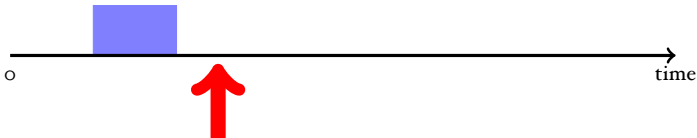
high priority



medium pr.



low priority



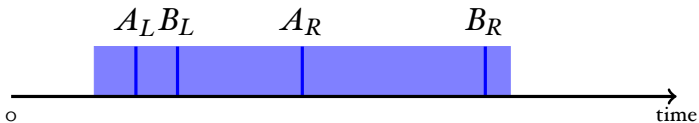
Scheduling: You want to avoid that a high priority process is starved indefinitely.

# Priority Inheritance Scheduling

- Let a low priority process  $L$  temporarily inherit the high priority of  $H$  until  $L$  leaves the critical section unlocking the resource.
- Once the resource is unlocked  $L$  returns to its original priority level.

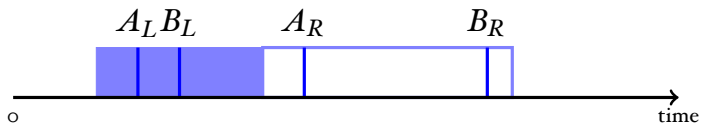


low priority



high priority

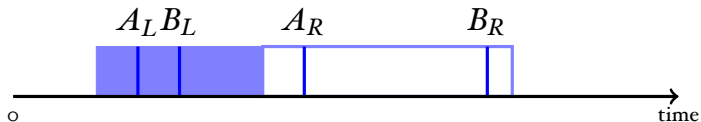
low priority



high priority



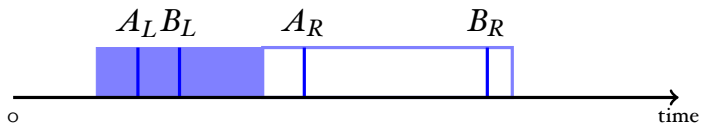
low priority



high priority



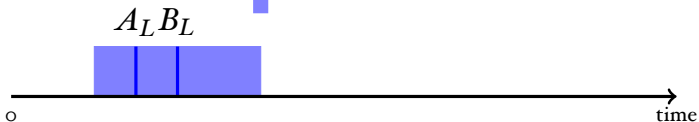
low priority



high priority



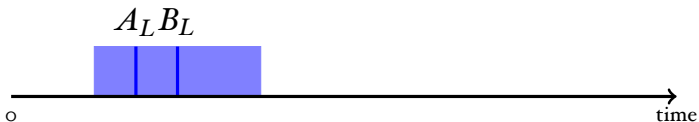
low priority



high priority



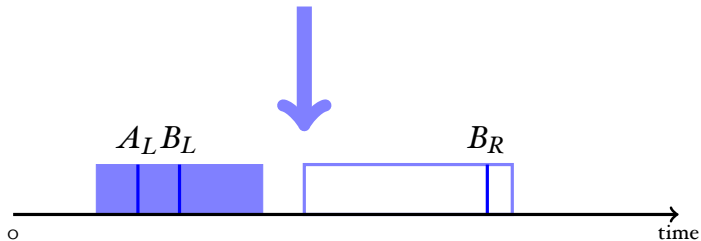
low priority



high priority



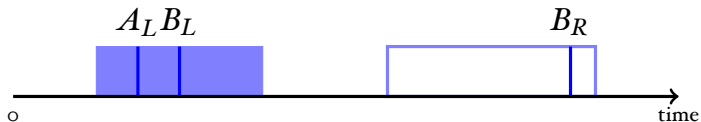
low priority



high priority



low priority

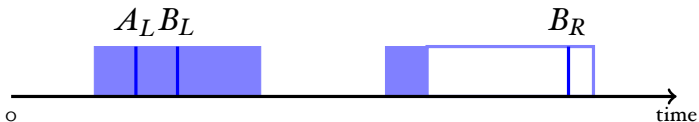


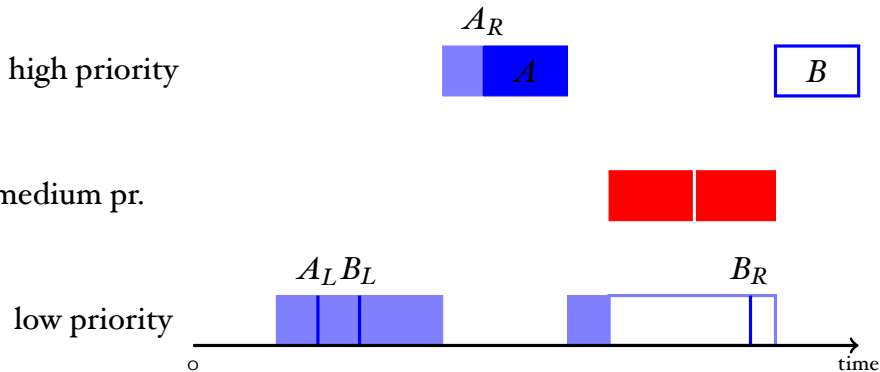


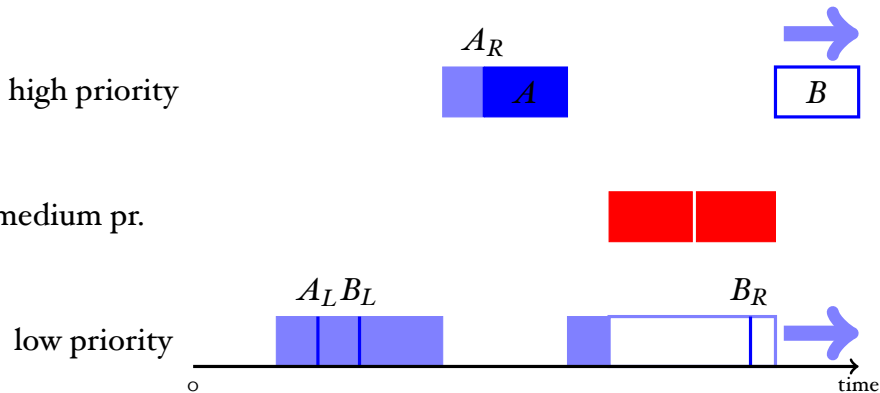
high priority



low priority







Scheduling: You want to avoid that a high priority process is staved indefinitely.

# Priority Inheritance Scheduling

- Let a low priority process  $L$  temporarily inherit the high priority of  $H$  until  $L$  leaves the critical section unlocking the resource.
- Once the resource is unlocked  $L$  returns to its original priority level. **BOGUS**

# Priority Inheritance Scheduling

- Let a low priority process  $L$  temporarily inherit the high priority of  $H$  until  $L$  leaves the critical section unlocking the resource.
- Once the resource is unlocked  $L$  returns to its original priority level. **BOGUS**
- ... $L$  needs to switch to the highest **remaining** priority of the threads that it blocks.

this error is already known since around 1999

- by Rajkumar, 1991
- *“it resumes the priority it had at the point of entry into the critical section”*

- by Jane Liu, 2000
- *“The job  $J_1$  executes at its inherited priority until it releases  $R$ ; at that time, the priority of  $J_1$  returns to its priority at the time when it acquires the resource  $R$ .”*
- gives pseudo code and totally bogus data structures
- interesting part *“left as an exercise”*

- by Laplante and Ovaska, 2011 (§113.76)
- “*when [the task] exits the critical section that caused the block, it reverts to the priority it had when it entered that section*”



- by Silberschatz, Galvin, and Gagne, 2013 (OS-bible)
- *“Upon releasing the lock, the [low-priority] thread will revert to its original priority.”*

# Priority Scheduling

- a scheduling algorithm that is widely used in real-time operating systems
- has been “proved” correct by hand in a paper in 1983
- but this algorithm turned out to be incorrect, despite its “proof”

# Priority Scheduling

- a scheduling algorithm that is widely used in real-time operating systems
- has been “proved” correct by hand in a paper in 1983
- but this algorithm turned out to be incorrect, despite its “proof”
- we corrected the algorithm and then **really** proved that it is correct
- we implemented this algorithm in a small OS called PINTOS (used for teaching at Stanford)
- our implementation was much more efficient than their reference implementation

# Design of AC-Policies

Imagine you set up an access policy (root, lpd, users, staff, etc)

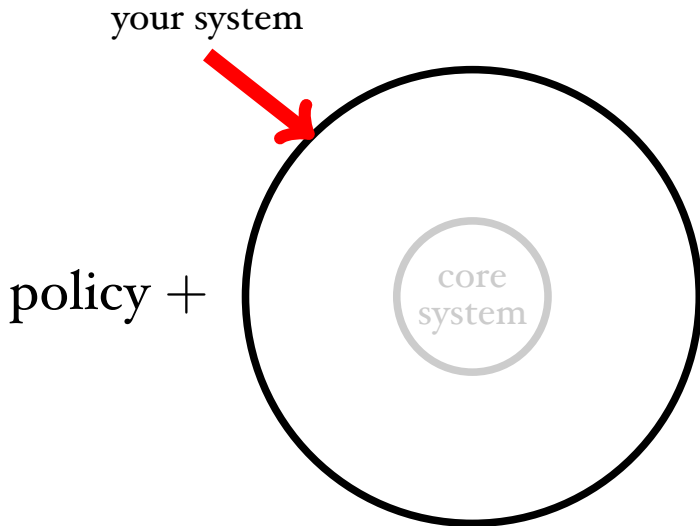
# Design of AC-Policies

Imagine you set up an access policy (root, lpd, users, staff, etc)

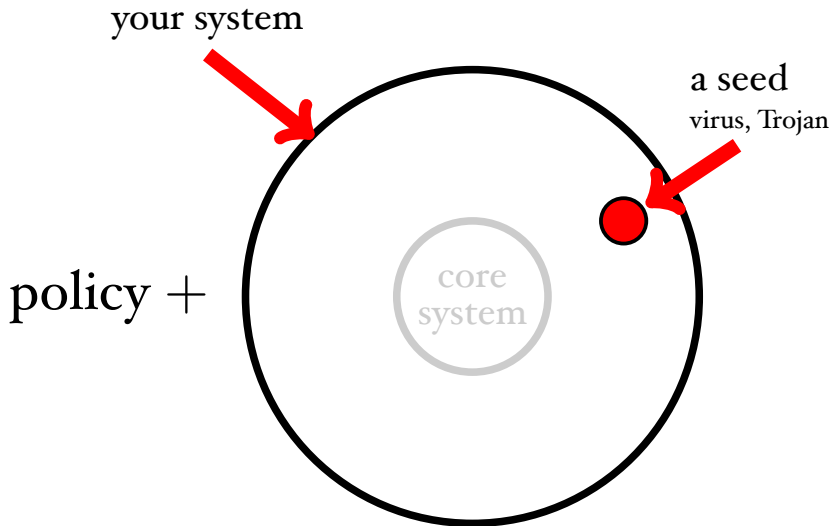
*“what you specify is what you get but not necessarily what you want...”*

main work by Chunhan Wu (a PhD-student in Nanjing)

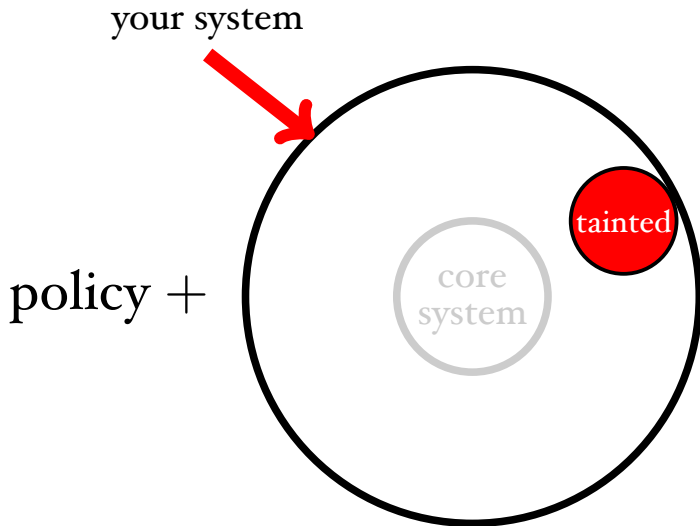
# Testing Policies



# Testing Policies

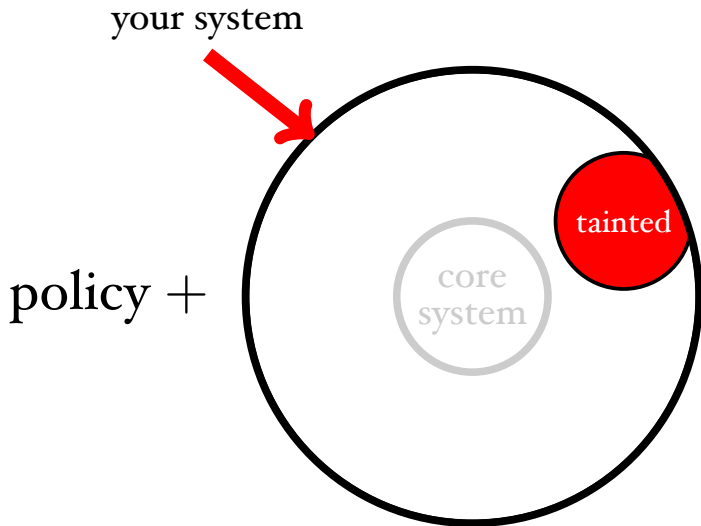


# Testing Policies

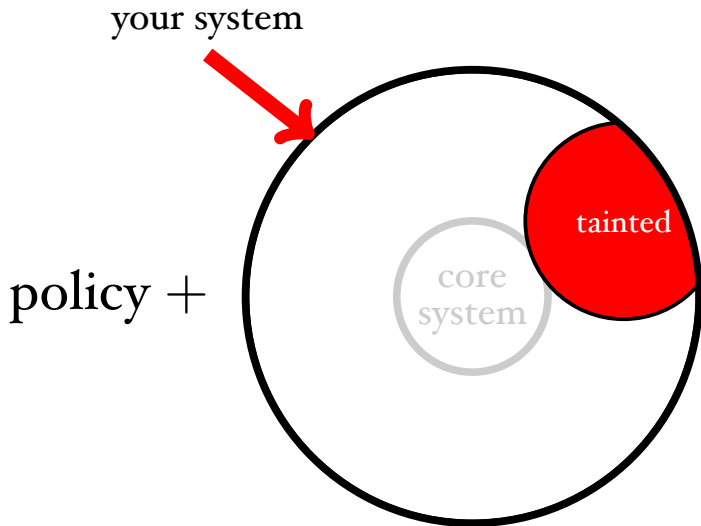




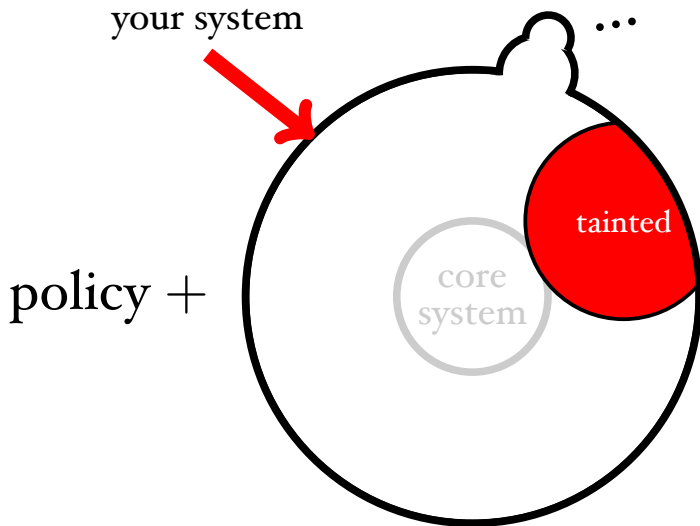
# Testing Policies



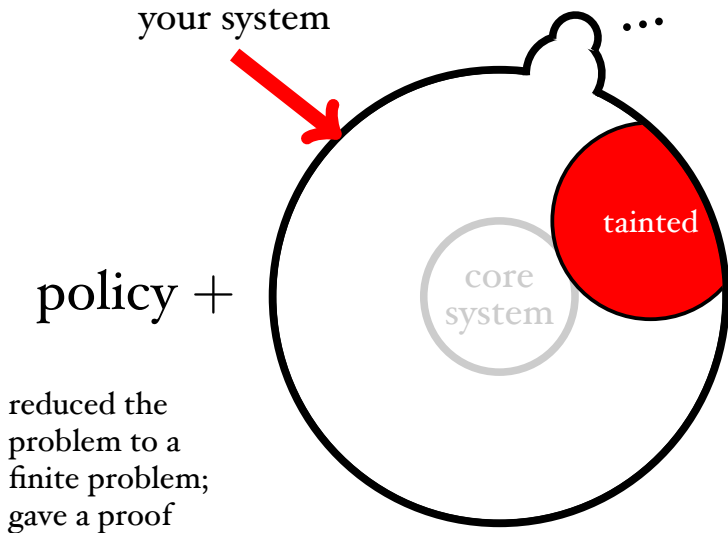
# Testing Policies



# Testing Policies



# Testing Policies



# Big Proofs in CS (I)

Formal proofs in CS sound like science fiction?  
Completely irrelevant! Lecturer gone mad?

# Big Proofs in CS (I)

Formal proofs in CS sound like science fiction?  
Completely irrelevant! Lecturer gone mad?

- 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 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.”

# Big Proofs in CS (2)

- in 2010, verification of a micro-kernel operating system (approximately 8700 loc)
  - used in helicopters and mobile phones
  - 200k loc of proof
  - 25 - 30 person years
  - found 160 bugs in the C code (144 by the proof)

“Real-world operating-system kernel with an end-to-end proof of implementation correctness and security enforcement”



# Big Proofs in CS (2)

- in 2010, verification of a micro-kernel operating system (approximately 8700 loc)
  - used in helicopters and mobile phones
  - 200k loc of proof
  - 25 - 30 person years
  - found 160 bugs in the C code (144 by the proof)

“Real-world operating-system kernel with an end-to-end proof of implementation correctness and security enforcement”

unhackable kernel (New Scientists, September 2015)

# Big Proofs in CS (3)

- verified TLS implementation
- verified compilers (CompCert, CakeML)
- verified distributed systems (Verdi)
- verified Oses or OS components  
(seL4, CertiKOS, Ironclad Apps, ...)
- verified cryptography

# How Did This Happen?

Lots of ways!

- better programming languages
  - basic safety guarantees built in
  - powerful mechanisms for abstraction and modularity
- better software development methodology
- stable platforms and frameworks
- better use of specifications

If you want to build software that works or is secure, it is helpful to know what you mean by “works” and by “is secure”!

# Goal

Remember the Bridges example?

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

# Goal

Remember the Bridges example?

- Can we look at our programs and somehow ensure they are secure/bug free/correct?
- Very hard: Anything interesting about programs is equivalent to halting problem, which is undecidable.

# Goal

Remember the Bridges example?

- Can we look at our programs and somehow ensure they are secure/bug free/correct?
- Very hard: Anything interesting about programs is equivalent to 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.  $\Rightarrow$  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?

for example no key is leaked

# Concrete Example: Sign-Analysis

$\langle Exp \rangle ::= \langle Exp \rangle + \langle Exp \rangle$   
|  $\langle Exp \rangle * \langle Exp \rangle$   
|  $\langle Exp \rangle = \langle Exp \rangle$   
|  $\langle num \rangle$   
|  $\langle var \rangle$

$\langle Stmt \rangle ::= \langle label \rangle :$   
|  $\langle var \rangle := \langle Exp \rangle$   
|  $jmp? \langle Exp \rangle \langle label \rangle$   
|  $goto \langle label \rangle$

$\langle Prog \rangle ::= \langle Stmt \rangle \dots \langle Stmt \rangle$

```
a := 1
n := 5
top: jmp? n = 0 done
     a := a * n
     n := n + -1
     goto top
done:
```

# Concrete Example: Sign-Analysis

$\langle Exp \rangle ::= \langle Exp \rangle + \langle Exp \rangle$

|  $\langle Exp \rangle * \langle Exp \rangle$

|  $\langle Exp \rangle = \langle Exp \rangle$

|  $\langle num \rangle$

|  $\langle var \rangle$

$\langle Stmt \rangle ::= \langle label \rangle :$

|  $\langle var \rangle := \langle Exp \rangle$

|  $\text{jmp? } \langle Exp \rangle \langle$

|  $\text{goto } \langle label \rangle$

$\langle Prog \rangle ::= \langle Stmt \rangle \dots \langle Stmt \rangle$

```
n := 6
m1 := 0
m2 := 1
top: jmp? n = 0 done
     tmp := m2
     m2 := m1 + m2
     m1 := tmp
     n := n + -1
     goto top
done:
```

# Concrete Example: Sign-Analysis

$$\begin{aligned} \langle \textit{Exp} \rangle & ::= \langle \textit{Exp} \rangle + \langle \textit{Exp} \rangle \\ & | \langle \textit{Exp} \rangle * \langle \textit{Exp} \rangle \\ & | \langle \textit{Exp} \rangle = \langle \textit{Exp} \rangle \\ & | \langle \textit{num} \rangle \\ & | \langle \textit{var} \rangle \\ \langle \textit{Stmt} \rangle & ::= \langle \textit{label} \rangle : \\ & | \langle \textit{var} \rangle := \langle \textit{Exp} \rangle \\ & | \textit{jmp?} \langle \textit{Exp} \rangle \langle \textit{label} \rangle \\ & | \textit{goto} \langle \textit{label} \rangle \\ \langle \textit{Prog} \rangle & ::= \langle \textit{Stmt} \rangle \dots \langle \textit{Stmt} \rangle \end{aligned}$$

# Eval: An Interpreter

$$\begin{aligned} [n]_{env} &\stackrel{\text{def}}{=} n \\ [x]_{env} &\stackrel{\text{def}}{=} env(x) \\ [e_1 + e_2]_{env} &\stackrel{\text{def}}{=} [e_1]_{env} + [e_2]_{env} \\ [e_1 * e_2]_{env} &\stackrel{\text{def}}{=} [e_1]_{env} * [e_2]_{env} \\ [e_1 = e_2]_{env} &\stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } [e_1]_{env} = [e_2]_{env} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

```
def eval_exp(e: Exp, env: Env) : Int = e match {
  case Num(n) => n
  case Var(x) => env(x)
  case Plus(e1, e2) => eval_exp(e1, env) + eval_exp(e2, env)
  case Times(e1, e2) => eval_exp(e1, env) * eval_exp(e2, env)
  case Equ(e1, e2) =>
    if (eval_exp(e1, env) == eval_exp(e2, env)) 1 else 0
}
```



## A program

```
    a := 1
    n := 5
top:  jmp? n = 0 done
      a := a * n
      n := n + -1
      goto top
done:
```

## The *snippets* of the program:

```
"""
    a := 1
    n := 5
top:  jmp? n = 0 done
      a := a * n
      n := n + -1
      goto top
done:
```

```
top:  jmp? n = 0 done
      a := a * n
      n := n + -1
      goto top
done:
```

```
done:
```

# Eval for Stmt

Let  $sn$  be the snippets of a program

$$[nil]_{env} \stackrel{\text{def}}{=} env$$

$$[Label(l) :: rest]_{env} \stackrel{\text{def}}{=} [rest]_{env}$$

$$[x := e :: rest]_{env} \stackrel{\text{def}}{=} [rest]_{(env[x:=e]_{env})}$$

$$[jmp? e l :: rest]_{env} \stackrel{\text{def}}{=} \begin{cases} [sn(l)]_{env} & \text{if } [e]_{env} = \mathbf{I} \\ [rest]_{env} & \text{otherwise} \end{cases}$$

$$[goto l :: rest]_{env} \stackrel{\text{def}}{=} [sn(l)]_{env}$$

Start with  $[sn(“”)]_{\emptyset}$

# Eval in Code

```
def eval(sn: Snips) : Env = {  
  def eval_stmts(sts: Stmts, env: Env) : Env = sts match {  
    case Nil => env  
    case Label(l)::rest => eval_stmts(rest, env)  
    case Assign(x, e)::rest =>  
      eval_stmts(rest, env + (x -> eval_exp(e, env)))  
    case Jmp(b, l)::rest =>  
      if (eval_exp(b, env) == 1) eval_stmts(sn(l), env)  
      else eval_stmts(rest, env)  
    case Goto(l)::rest => eval_stmts(sn(l), env)  
  }  
  
  eval_stmts(sn(""), Map())  
}
```

# The Idea of Static Analysis

```
a := 1
n := 5
top: jmp? n = 0 done
     a := a * n
     n := n + -1
     goto top
done:
```



```
a := '+'
n := '+'
top: jmp? n = '0' done
     a := a * n
     n := n + '- '
     goto top
done:
```

Replace all constants and variables by either +, - or 0. What we want to find out is what the sign of a and n is (they should always positive).

# Sign Analysis?

$e_1$	$e_2$	$e_1 + e_2$
-	-	-
-	0	-
-	+	-, 0, +
0	x	x
+	-	-, 0, +
+	0	+
+	+	+

$e_1$	$e_2$	$e_1 * e_2$
-	-	+
-	0	0
-	+	-
0	x	0
+	-	-
+	0	0
+	+	+

$$\begin{aligned}
 [n]_{aenv} &\stackrel{\text{def}}{=} \begin{cases} \{+\} & \text{if } n > 0 \\ \{-\} & \text{if } n < 0 \\ \{0\} & \text{if } n = 0 \end{cases} \\
 [x]_{aenv} &\stackrel{\text{def}}{=} aenv(x) \\
 [e_1 + e_2]_{aenv} &\stackrel{\text{def}}{=} [e_1]_{aenv} \oplus [e_2]_{aenv} \\
 [e_1 * e_2]_{aenv} &\stackrel{\text{def}}{=} [e_1]_{aenv} \otimes [e_2]_{aenv} \\
 [e_1 = e_2]_{aenv} &\stackrel{\text{def}}{=} \{0, +\}
 \end{aligned}$$

```

def aeval_exp(e: Exp, aenv: AEnv) : Set[Abst] = e match {
  case Num(0) => Set(Zero)
  case Num(n) if (n < 0) => Set(Neg)
  case Num(n) if (n > 0) => Set(Pos)
  case Var(x) => aenv(x)
  case Plus(e1, e2) =>
    aplus(aeval_exp(e1, aenv), aeval_exp(e2, aenv))
  case Times(e1, e2) =>
    atimes(aeval_exp(e1, aenv), aeval_exp(e2, aenv))
  case Equ(e1, e2) => Set(Zero, Pos)
}

```

# AEval for Stmts

Let  $sn$  be the snippets of a program

$$[\text{nil}]_{aenv} \rightarrow (\text{nil}, aenv)$$

$$[\text{Label}(l : ) :: \text{rest}]_{aenv} \rightarrow (\text{rest}, aenv)$$

$$[x := e :: \text{rest}]_{aenv} \rightarrow (\text{rest}, aenv[x := [e]_{aenv}])$$

$$[\text{jmp? } e \ l :: \text{rest}]_{aenv} \rightarrow (sn(l), aenv) \text{ and } (\text{rest}, aenv)$$

$$[\text{goto } l :: \text{rest}]_{aenv} \rightarrow (sn(l), aenv)$$

Start with  $[sn(\text{""})]_{\emptyset}$ , try to reach all *states* you can find (until a fix point is reached).

Check whether there are only  $aenv$  in the final states that have your property.

# Sign Analysis

- We want to find out whether  $a$  and  $n$  are always positive?
- After a few optimisations, we can indeed find this out.
  - equations return  $+$  or  $0$
  - branch into only one direction if you know
  - if  $x$  is  $+$ , then  $x + -1$  cannot be negative
- What is this good for? Well, you do not need underflow checks (in order to prevent buffer-overflow attacks). In general this technique is used to make sure keys stay secret.



# Take Home Points

- While testing is important, it does not show the absence of bugs/vulnerabilities.
- More and more we need (formal) proofs that show a program is bug free.