Admin:

Pset #5 out, due 5/3

Project presentation start 5/8 (aim for 12 min total)
(10 min talk + 2 min Q&A & setup)

Attend days other than your own talk.

Today:

Quines

Decidability & A/V detection

↓ "Trusting Trust"

(no time) Certificates

Readings:

Ken Thompson's paper "Trusting Trust"

Katz: Chapter 12.7 (on certificates)

"Malware Theory"

6.857 Rivest
L14.3 3/31/08

- We are used to programs that work on other programs:
  E.g. interpreters, optimizers, compilers, byte-code verifier, virus detector,..

- We are now interested in programs that work on <u>themselves</u> ("self-referential") - they have access to their own source code.

- Can you write a program that <u>prints itself</u>? ("Quine")
  in C:

```
char *s="char *s=%c%s%c; main(){printf(s,34,s,34);}";
main(){printf(s,34,s,34);}
```

Note: 34 is decimal code for double-quote char.

- We can modify above pgm to save text in a variable, rather than printing it.
  (Use "sprintf" in C; or modify s by substitution,...)
  Thus, we can have programs P of form:

$$P \equiv \begin{cases} s = \langle \text{text for } P \rangle; & \leftarrow \text{modification of above} \\ \sim & \leftarrow \text{other code, can operate on s, as if it were input.} \end{cases}$$

- For example, we can write a program P that applies a routine A to text of P:

$$P \equiv \begin{cases} s = \langle \text{text for } P \rangle \\ \text{define } A(x): \equiv \\ \\ A(s) \end{cases}$$

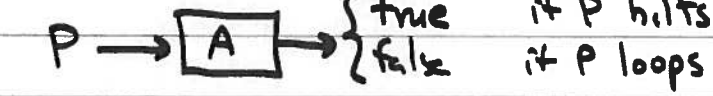
or (in high-level notation):

$P \equiv A(P)$ [running P is same as applying A to source code for P.]

(All this is called "Recursion Theorem" in theory of computation...)

- **Def:** The Halting Problem is: Given a program P that takes no inputs, decide if P halts, or loops forever when run.

- **Thm:** The Halting Problem is undecidable.
(I.e. there is no program A that takes as input a description of a program P, and always halts & outputs correctly whether P halts or loops.)



Proof: Assume such an A exists (& we have its code):

Let P ≡ [if A(P) then loop else halt

What does A do on P?

if A(P) true then A says P halts ⇒ A is wrong

if A(P) false then A says P loops ⇒ A is wrong

∴ A doesn't exist. ▨

- More generally, determining any nontrivial property of (output) behavior of a program is undecidable.

(known as "Rice's Theorem")

Nontrivial means: ∃ program X that exhibits behavior
& ∃ program X' that doesn't exhibit behavior.

Behavior = output behavior; not things that depend on source code or number of steps taken

e.g. "uses printer" or "halts" or "prints two zeros in a row"

Pf: Same as before.

Assume A can decide if program has property.

Consider P ≡ [if A(P) then X'() else X()]

A is wrong about P. ☒

- Thm: Virus detection is undecidable. (Cohen '87)

(Define virus to be a progam that "spreads" (infects other programs).)

Pf: Same as for Rice's Theorem, etc.

Assume A can decide if input program is a virus.

Consider P ≡ [if A(P) then halt else spread()]

A is wrong on P.

(Contradiction, so A doesn't exist.)

- Every virus scanner must make mistakes on some inputs, and/or fail to halt on some inputs.

- Even worse: (Chess & White)

What we have shown is

$(\forall A)(\exists P)$ A is wrong about whether P is a virus.

But it is true that

$(\exists P)(\forall A)$ A is wrong about whether P is a virus

P is polymorphic
P uses randomization
P spreads as a different form P'

P = if A(P) then halt else spread( )
↖ randomized

P' = if A'(P) then halt else spread.
↖ random A/V Scanner

∃ one member of family that defeats each A_{true}

- More generally, determining any property of behavior of a program is undecidable (as long as it is nontrivial.) ("Rice's theorem")
  halt ⇒ exhibit behavior
  false ⇒ don't exhibit behavior

- Ken Thompson's "Reflections on Trusting Trust" (1984)
  example of nasty malware: can't even find it by looking at source code (!)

let L = login program L(pw): [if check(pw) then allow_login() else reject()]

evil login program L'(pw): [if pw = "3YNQ74B" or check(pw) then allow_login() else reject()]

but: someone may notice source has been modified

so: attacker can also modify compiler (!)
Let C = standard compiler

evil compiler C'(x) = if x = L then output C(L') else output C(x)

now source for L is left alone, but source for compiler changed; it may be noticed.

- so doubly-evil compiler:

$C''(x) =$ if $x = L$ then output $C(L')$
else if $x = C$ then output $C(C'')$
else output $C(x)$

← note self-reference!

Attacks leaves sources as $L, C, \ldots, x$
but binaries as $C(L'), C(C''), \ldots, C'(x)$

all sources look clean!
situation is stable: recompiling any program yields same binary!

Ouch!!!

Moral (Thompson): "You can't trust any code you did not totally create yourself!"