6.857 Rivest
L 18.1 4/11/11

Admin:

Outline: Viruses

- ☐ Brief history of viruses
- ☐ Self-referential programs
- ☐ Halting Problem is undecidable
- ☐ Virus detection is undecidable
- ☐ Some practicalities of virus detection

6.857 Rivest
L18.2 4/11/11

Virus History

1981: first virus: "Elk Cloner" Apple II; Floppy disk xmit.

1983: Term "virus" coined by Adleman & Cohen (1986 PhD thesis)

1988: Internet worm (≈60,000 hosts on Internet; 10% infected.)
- fingerd buffer overflow
- sendmail debug mode left on - left access to shell
- dictionary attacks on passwords
- "trusted hosts" vulnerability

1990's: 1M hosts on web in '92
First "polymorphic" viruses (Tequila & Amoeba)
Start of anti-virus industry (Symantec)
late 90's: 30M hosts on web ('98)

2001: Code Red infects 360,000 hosts in 14 hours
(out of 125M hosts on web in '01)
buffer overflow in M.S. indexing service

2003: Slammer: infected M.S. SQL servers
doubled # infected every 8.5 seconds!
infected ≥ 75,000 hosts
caused network congestion (severe)
376 byte UDP packet
random scanning of 32-bit IP addresses

Jan '09
625M

stuxnet

2006: 440M hosts on Internet (www.isc.org)

"Malware Theory"

6.857 Rivest
L 18.3 4/11/11

- 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 themselves ("self-referential") - they have access to their own source code.

- Can you write a program that prints itself? ("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 ≡ { s = <text for P>;   ← modification of above
    { ~~~                 ← other code, can operate on s, as if it were input.
```

- 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.)

P → [A] → { true if P halts; false if P 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 soure 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 program 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

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

But it is true that

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

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

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

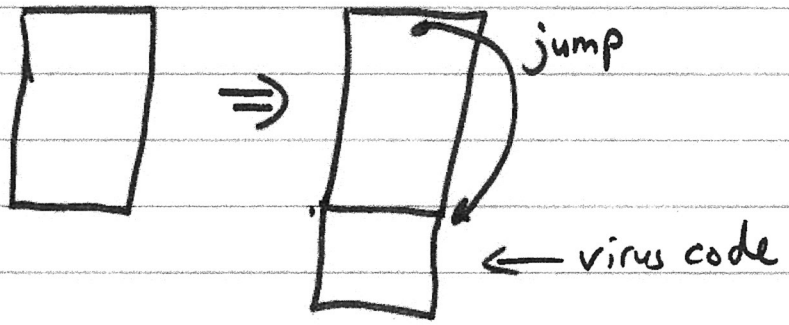
P' = if A'(P) then halt else spread.
↖ random A/V scanner

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

6.857 Rivest
L 18.7 4/11/11

(Material from Carey Nachenberg slides "Virus/Antivirus Co-evolution")

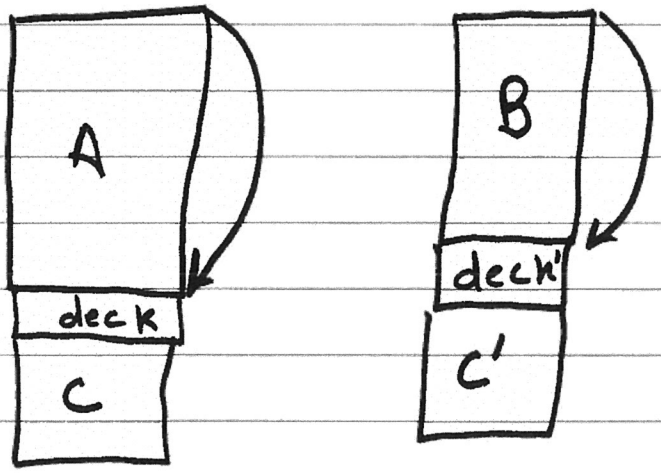
① simple virus: insert code at end for virus
insert jump to beginning of virus code



② A/V responses:
"signatures": for each virus, find static pattern that occurs in virus code, but not in any common good code (e.g. μsoft word)
build table of signatures
scan files for signatures

③ virus writer response: polymorphic code
"poly" = many
"morphic" = shape
virus code never looks the same

infected pgm



dec = decryption routine

Use encryption: when infected pgm A runs, it generates a new key k', so that ciphertext $C' = E(k', \text{virus})$ looks random. Only common part is small decryption routine.

(4) A/V response:
scan for small decryption routine

(5) Virus writer response:
change enc/dec algorithm when spreading;
generate at random a new enc/dec pair.
∴ no static decryption routine!

(6) A/V response:
emulate code in "sandbox" until it decrypts itself, <u>then</u> run static signature check.
(Hard to know exactly when to stop, though...)

(7) Virus writer's response:

- randomly decrypt, or not
- don't decrypt at beginning of execution, but later
- "recompile" virus code in a way that preserves semantic equivalence, but destroys all "signatures"

(8) A/V response: this is getting really hard!