CFI and Malware

With material from Michelle Mazurek,
Dave Levin, Vern Paxson, Dawn Song

Control-flow Integrity (CFI)

- *Define "expected behavior":*

  Control flow graph (CFG)

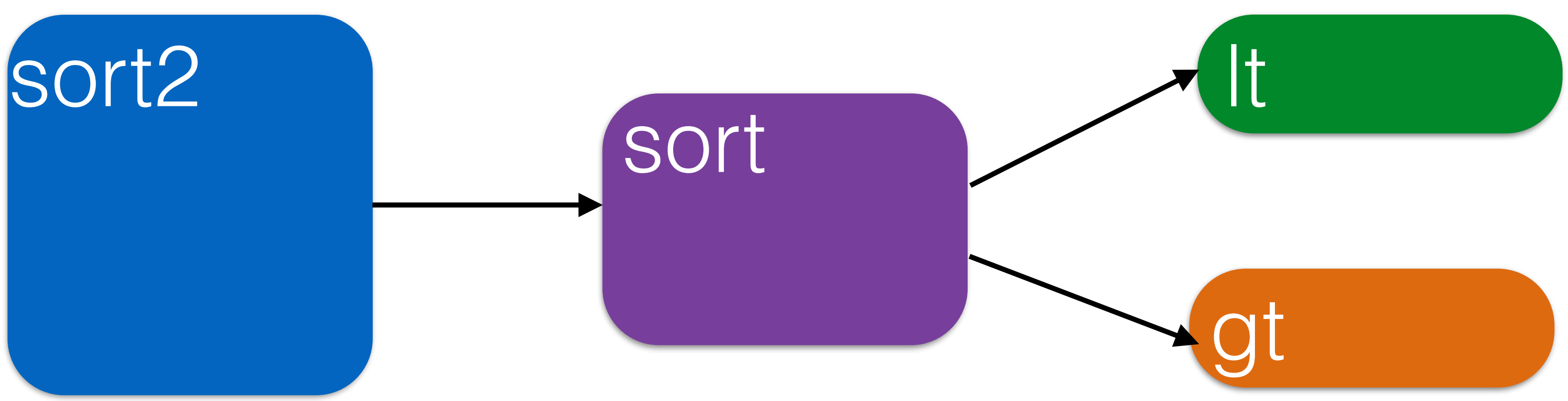
- *Detect deviations from expectation efficiently*

- *Avoid compromise of the detector*

Call Graph

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```

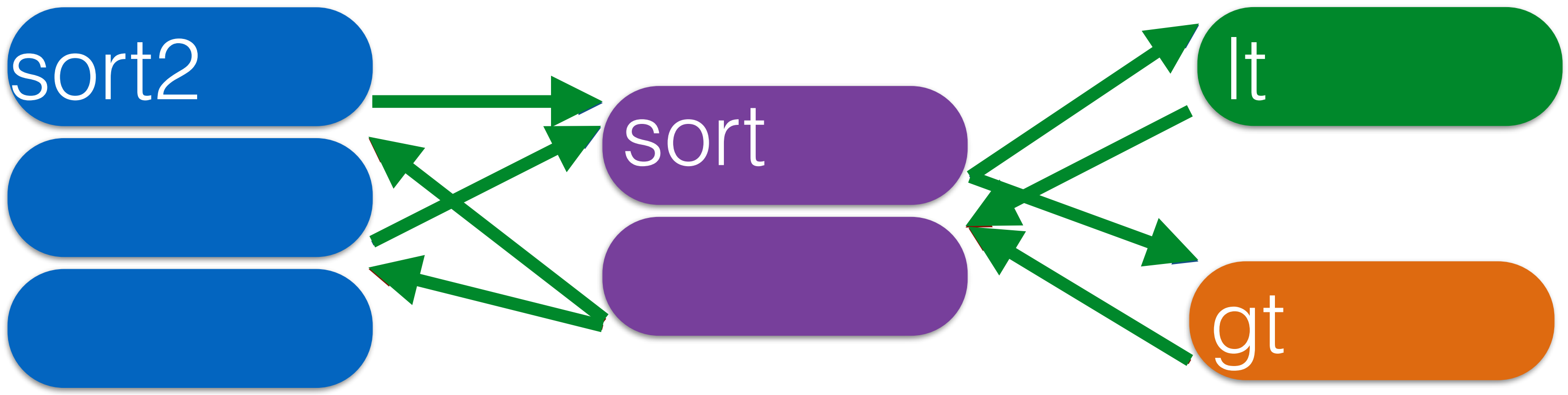


Which functions call other functions

Control Flow Graph

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



*Break into **basic blocks***
*Distinguish **calls** from **returns***

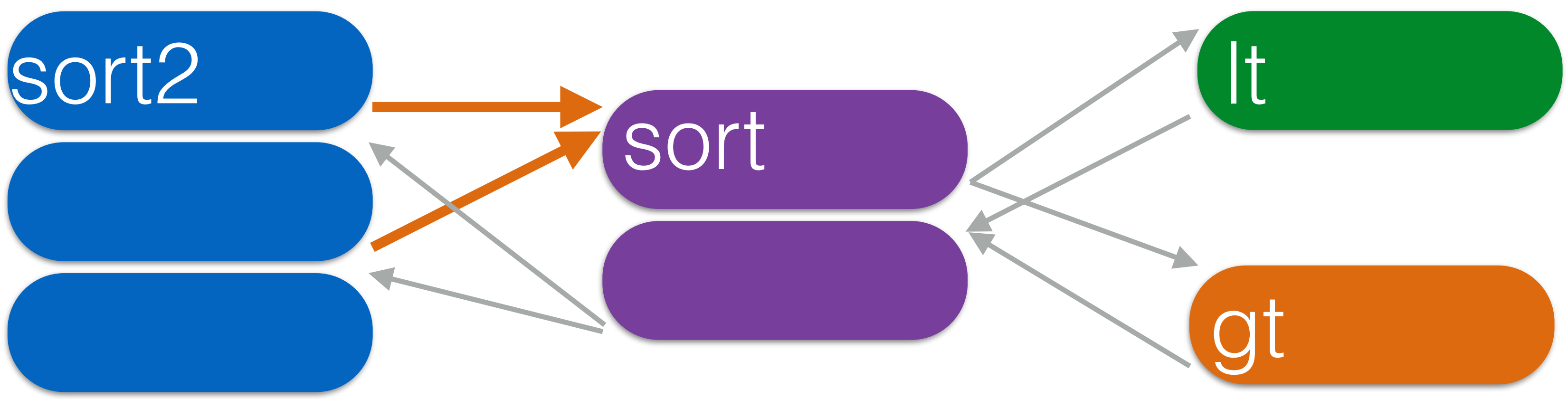
CFI: Compliance with CFG

- **Compute the call/return CFG** in advance
  - During compilation, or from the binary
- **Monitor the control flow** of the program and ensure that it only follows paths allowed by the CFG
- Observation: **Direct calls** need not be monitored
  - Assuming the code is immutable, the target address cannot be changed
- Therefore: **monitor only indirect calls**
  - `jmp`, `call`, `ret` with non-constant targets

Control Flow Graph

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```

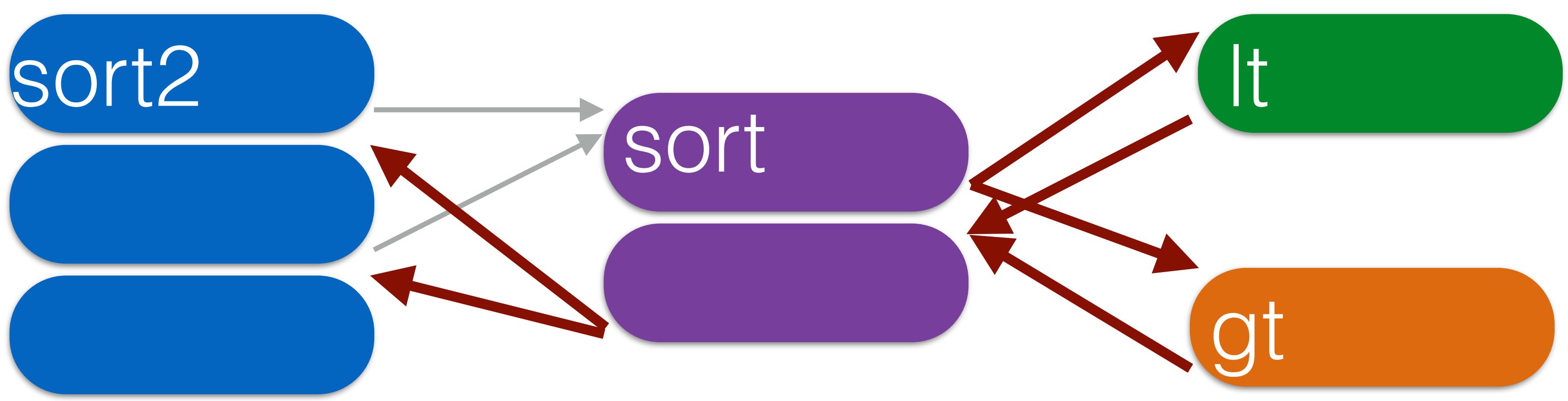


Direct calls *(always the same target)*

Control Flow Graph

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



Indirect transfer *(`call` via register, or `ret`)*

Control-flow Integrity (CFI)

- *Define "expected behavior":*

  Control flow graph (CFG)

- *Detect deviations from expectation efficiently*

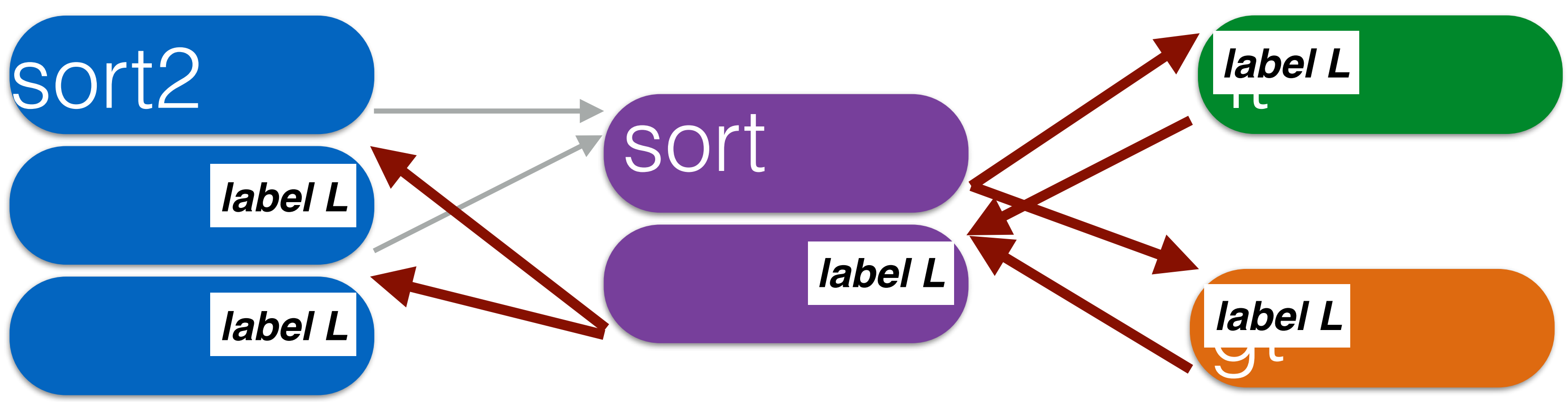
  In-line reference monitor (IRM)

- *Avoid compromise of the detector*

In-line Monitor

- Implement the monitor in-line, as a **program transformation**
- Insert a **label just before the target address** of an indirect transfer
- Insert **code to check the label of the target** at each indirect transfer
  - Abort if the label does not match
- The **labels are determined by the CFG**

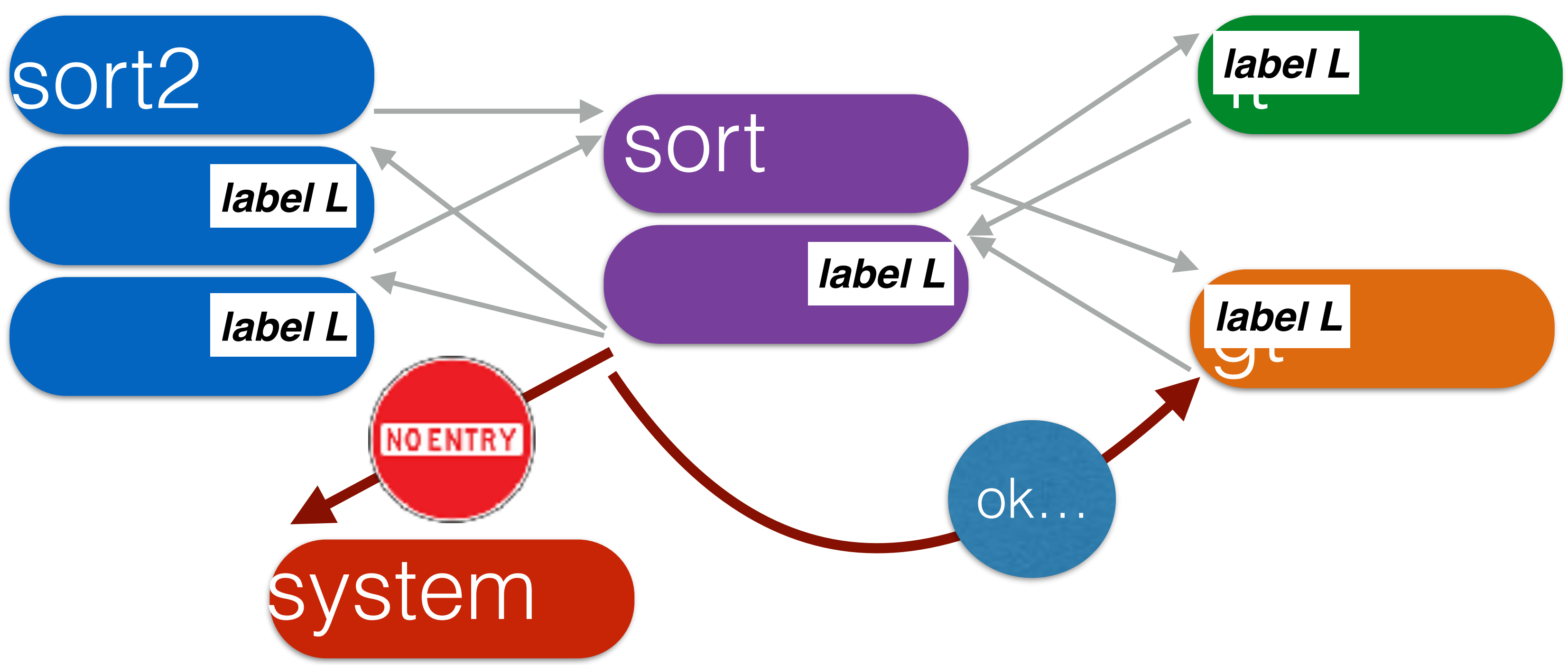
Simplest labeling



Use the same label at all targets:
label just means it's OK to jump here.

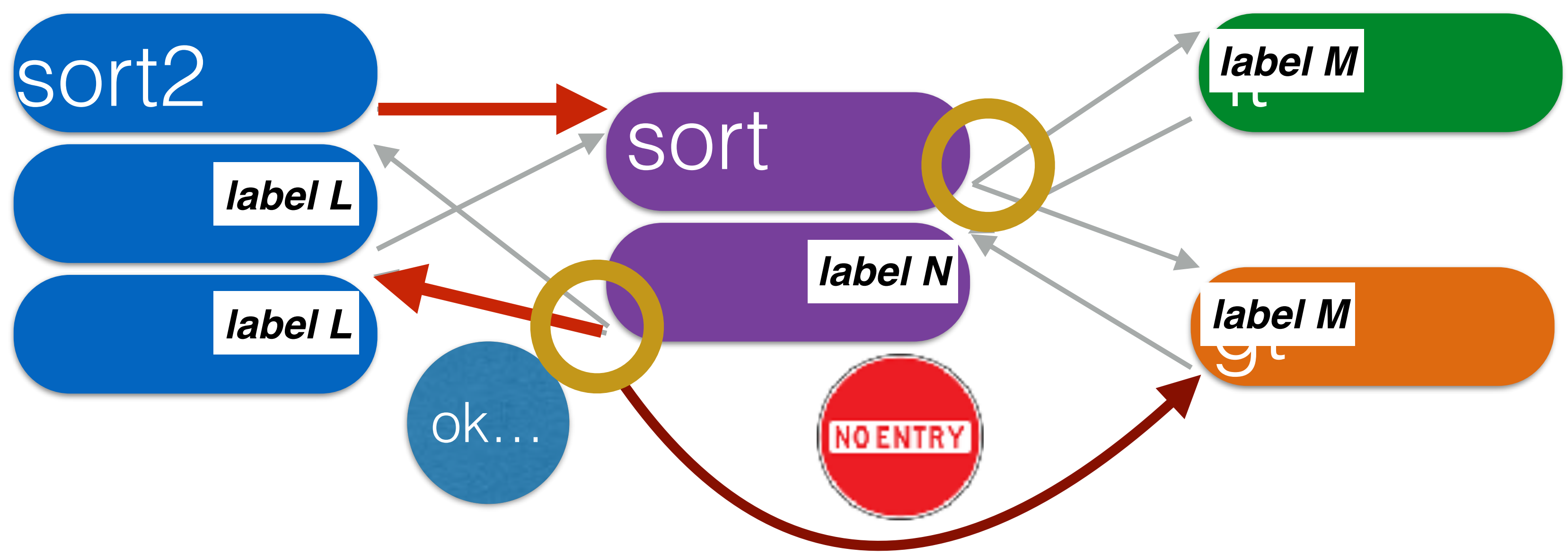
What could go wrong?

Simplest labeling



- Can’t return to functions that aren’t in the graph
- **Can** return to the right function in the wrong order

Detailed labeling



- All potential destinations of **same source** must match
  - Return sites from calls to `sort` must share a label (*L*)
  - Call targets `gt` and `lt` must share a label (*M*)
  - Remaining label unconstrained (*N*)

Prevents more abuse than simple labels,
but still permits call from site A to return to site B

Classic CFI instrumentation

Before CFI

```
FF 53 08                    call   [ebx+8]              ; call a function pointer
```

is instrumented using `prefetchnta` destination IDs, to become:

After CFI

```
8B 43 08                    mov   eax, [ebx+8]          ; load pointer into register
3E 81 78 04 78 56 34 12     cmp   [eax+4], 12345678h    ; compare opcodes at destination
75 13                       jne   error_label           ; if not ID value, then fail
FF D0                       call  eax                   ; call function pointer
3E 0F 18 05 DD CC BB AA     prefetchnta [AABBCCDDh]     ; label ID, used upon the return
```

Fig. 4. Our CFI implementation of a call through a function pointer.

Bytes (opcodes)	x86 assembly code	Comment
C2 10 00	ret 10h	; return, and pop 16 extra bytes

is instrumented using `prefetchnta` destination IDs, to become:

```
8B 0C 24                    mov   ecx, [esp]            ; load address into register
83 C4 14                    add   esp, 14h              ; pop 20 bytes off the stack
3E 81 79 04 DD CC BB AA     cmp   [ecx+4], AABBCCDDh    ; compare opcodes at destination
75 13                       jne   error_label           ; if not ID value, then fail
FF E1                       jmp   ecx                   ; jump to return address
```

Classic CFI instrumentation

```
FF 53 08                     call   [ebx+8]          ; call a function pointer
```

is instrumented using `prefetchnta` destination IDs, to become:

```
8B 43 08                     mov   eax, [ebx+8]      ; load pointer into register
3E 81 78 04 78 56 34 12      cmp   [eax+4], 12345678h; compare opcodes at destination
75 13                        jne   error_label       ; if not ID value, then fail
FF D0                        call  eax               ; call function pointer
3E 0F 18 05 DD CC BB AA      prefetchnta [AABBCCDDh] ; label ID, used upon the return
```

Fig. 4.   Our CFI implementation of a call through a function pointer.

```
Bytes (opcodes)              x86 assembly code       Comment
----------------------------------------------------------------------------------
C2 10 00                     ret   10h               ; return, and pop 16 extra bytes
```

is instrumented using `prefetchnta` destination IDs, to become:

```
8B 0C 24                     mov   ecx, [esp]        ; load address into register
83 C4 14                     add   esp, 14h          ; pop 20 bytes off the stack
3E 81 79 04 DD CC BB AA      cmp   [ecx+4], AABBCCDDh; compare opcodes at destination
75 13                        jne   error_label       ; if not ID value, then fail
FF E1                        jmp   ecx               ; jump to return address
```

Efficient?

- **Classic CFI** (2005) imposes **16% overhead** on average, **45%** in the **worst case**
  - Works on arbitrary executables
  - Not modular (no dynamically linked libraries)
- **Modular CFI** (2014) imposes **5% overhead** on average, **12%** in the **worst case**
  - C only (part of LLVM)
  - Modular, with separate compilation
  - http://www.cse.lehigh.edu/~gtan/projects/upro/

Control-flow Integrity (CFI)

- *Define "expected behavior":*

  Control flow graph (CFG)

- *Detect deviations from expectation efficiently*

  In-line reference monitor (IRM)

- *Avoid compromise of the detector*

  Sufficient randomness, immutability

Can we defeat CFI?

- **Inject code** that has a **legal label**
  - *Won't work* because we assume **non-executable data**
- **Modify code labels** to allow the desired control flow
  - *Won't work* because the **code is immutable**
- **Modify stack during a check**, to make it seem to succeed
  - *Won't work* because **adversary cannot change registers** into which we load relevant data
    - No time-of-check, time-of-use bug (TOCTOU)

CFI Assurances

- CFI defeats **control flow-modifying** attacks
  - Remote code injection, ROP/return-to-libc, etc.
- But **not manipulation of control-flow** that is **allowed by the labels**/graph
  - Called **mimicry attacks**
  - The simple, single-label CFG is susceptible to these
- **Nor data leaks or corruptions**
  - Heartbleed would not be prevented
  - Nor the `authenticated` overflow
    - Which is allowed by the graph

```
void func(char *arg1)
{
  int authenticated = 0;
  char buffer[4];
  strcpy(buffer, str);
  if(authenticated) { …
}
```

Secure?

- MCFI can **eliminate 95.75% of ROP gadgets** on x86-64 versions of SPEC2006 benchmark suite
  - By ruling their use non-compliant with the CFG
- Average Indirect-target Reduction (AIR) **> 99%**
  - Essentially, the percentage of **possible targets of indirect jumps** that CFI rules out

Malware: Malicious code that runs on the victim’s system

How does malware run?

- Attacks a user- or network-facing vulnerable service
  - e.g., using techniques from prior lectures
- Backdoor: Added by a malicious developer
- Social engineering: Trick user into running/clicking
- Trojan horse: Offer a good service, add in the bad
- Attacker with physical access installs & runs it

What does malware do?

- Potentially nearly anything (subject to permissions)
- Brag: “APRIL 1st HA HA HA HA YOU HAVE A VIRUS!”
- Destroy: files, hardware
- Crash the machine, e.g., by over-consuming resource
    - Fork bombing or “rabbits”: while(1) { fork();
- Steal information (“exfiltrate”)
- Launch external attacks: spam, click fraud, DoS
- Ransomware: e.g., by encrypting files
- Rootkits: Hide from user or software-based detection
    - Often by modifying the kernel
- Man-in-the-middle attacks to sit between UI and reality

Viruses vs. worms

- Virus: Run when user initiates something
  - Run program open attachment, boot machine
  - Typically infects by altering *stored* code
  - Self-propagating: Create new instance elsewhere

- Worm: Runs while another program is running
  - No user intervention required
  - Typically infects by altering *running* code
  - Self-propagating: infect running code elsewhere

The line between these is thin and blurry; some are both

Technical challenges

- Viruses: Detection
  - Antivirus software wants to detect
  - Virus writers want to avoid detection as long as possible
  - ***Evade*** human response

- Worms: Spreading
  - The goal is to hit as many machines and as quickly as possible
  - ***Outpace*** human response

Viruses

Viruses

- They are **opportunistic**: they will *eventually* be run due to user action

- Two *orthogonal* aspects define a virus:
  1. How does it **propagate**?
  2. What else does it do (what is the "**payload**")?

- General infection strategy:
  - Alter some existing code to include the virus
  - Share it, expect users to (unwittingly, possibly automatically) re-share

- Viruses have been around since at least the 70s

Classified by what they infect

- Document viruses
    - Implemented within a formatted document (Word, PDF, etc.)
    - Enabled by macros, javascript
    - (Why you shouldn't open random attachments)
- Boot sector viruses
    - Boot sector: small disk partition at fixed location; loaded by firmware at boot
    - What's *supposed* to happen: this code loads the OS
    - Similar: AutoRun on music/video disks
    - (Why you shouldn't plug random USB drives into your computer)
- Etc.

Viruses have resulted in a technological arms race

The key is ***evasion***

Mechanisms for evasive **propagation**

Mechanisms for **detection** and prevention

Want to be able to claim wide coverage for a long time

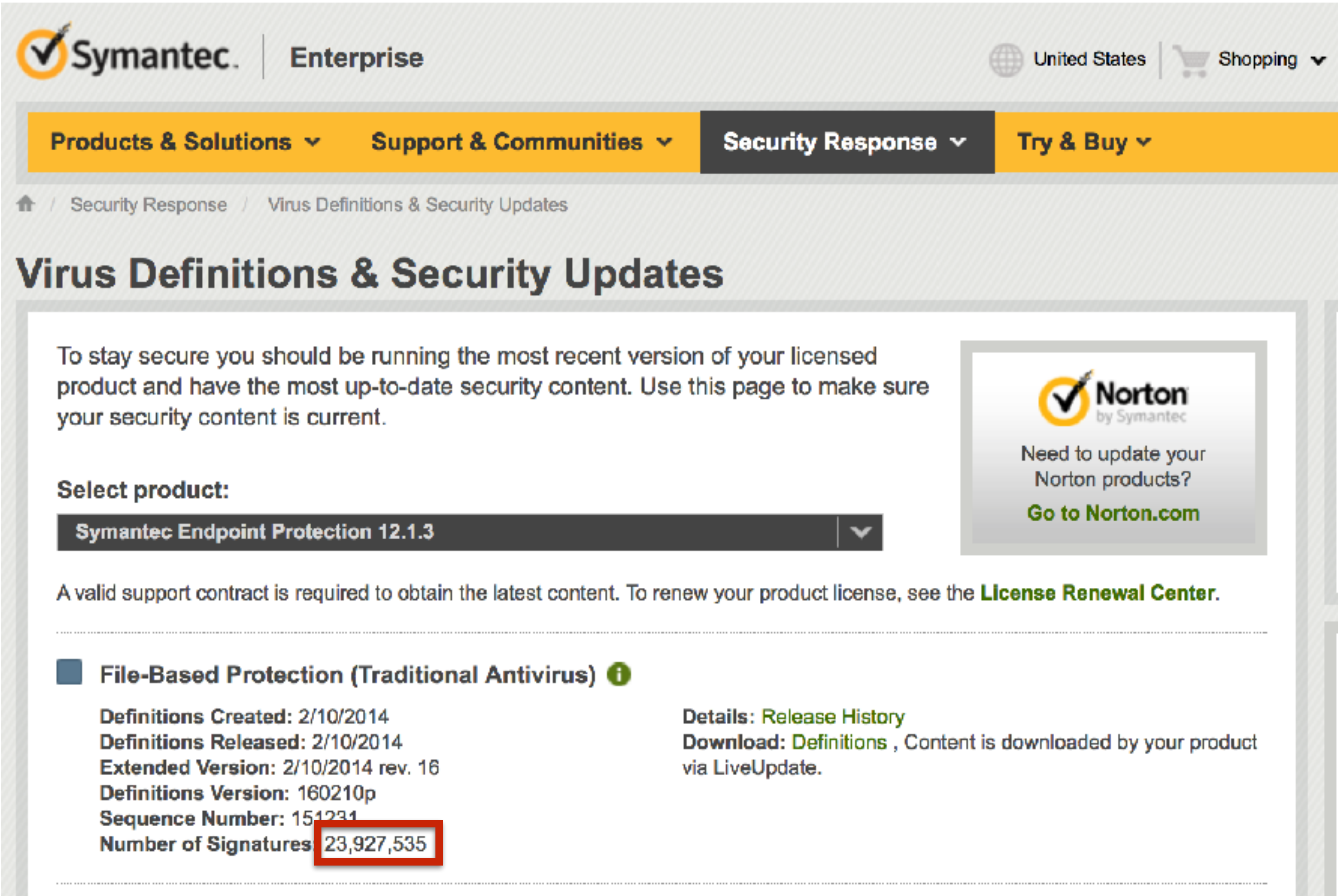
Want to be able to claim the ability to detect *many* viruses

How viruses propagate

- **Opportunity to run:** attach to something likely
  - autorun.exe on storage devices
  - Email attachments
- **Opportunity to infect:**
  - See a USB drive: overwrite autorun.exe
  - User is sending an email: alter the attachment
  - Proactively create emails (“I Love You”)

Detecting viruses: Signatures

- Identify bytes corresponding to known virus
- Install recognizer to check all files
  - In practice, requires fast scanning
- Drives multi-million$ antivirus market
  - Marketing via # signatures recognized
  - Is this a useful metric?


Symantec.
Enterprise
United States
Shopping
Products & Solutions
Support & Communities
Security Response
Try & Buy
Security Response / Virus Definitions & Security Updates
Virus Definitions & Security Updates
To stay secure you should be running the most recent version of your licensed product and have the most up-to-date security content. Use this page to make sure your security content is current.
Norton by Symantec
Need to update your Norton products?
Go to Norton.com
Select product:
Symantec Endpoint Protection 12.1.3
A valid support contract is required to obtain the latest content. To renew your product license, see the License Renewal Center.
File-Based Protection (Traditional Antivirus)
Definitions Created: 2/10/2014
Definitions Released: 2/10/2014
Extended Version: 2/10/2014 rev. 16
Definitions Version: 160210p
Sequence Number: 151231
Number of Signatures: 23,927,535
Details: Release History
Download: Definitions , Content is downloaded by your product via LiveUpdate.

Um.. thanks?

FEATURE

Antivirus vendors go beyond signature-based antivirus

Robert Westervelt, News Director

Share

This article can also be found in the Premium Editorial Download **"Information Security magazine: Successful cloud migrations require careful planning."**

Download it now to read this article plus other related content.

Security experts and executives at security vendors are in agreement that signature-based antivirus isn't able to keep up with the explosion of malware. For example, in 2009, Symantec says it wrote about 15,000 antivirus signatures a day; that number has increased to 25,000 antivirus signatures every day.

"Signatures have been dying for quite a while," says Mikko H. Hypponen, chief research officer of Finnish-based antivirus vendor, F-Secure. "The sheer number of malware samples we see every day completely overwhelms our ability to keep up with them."

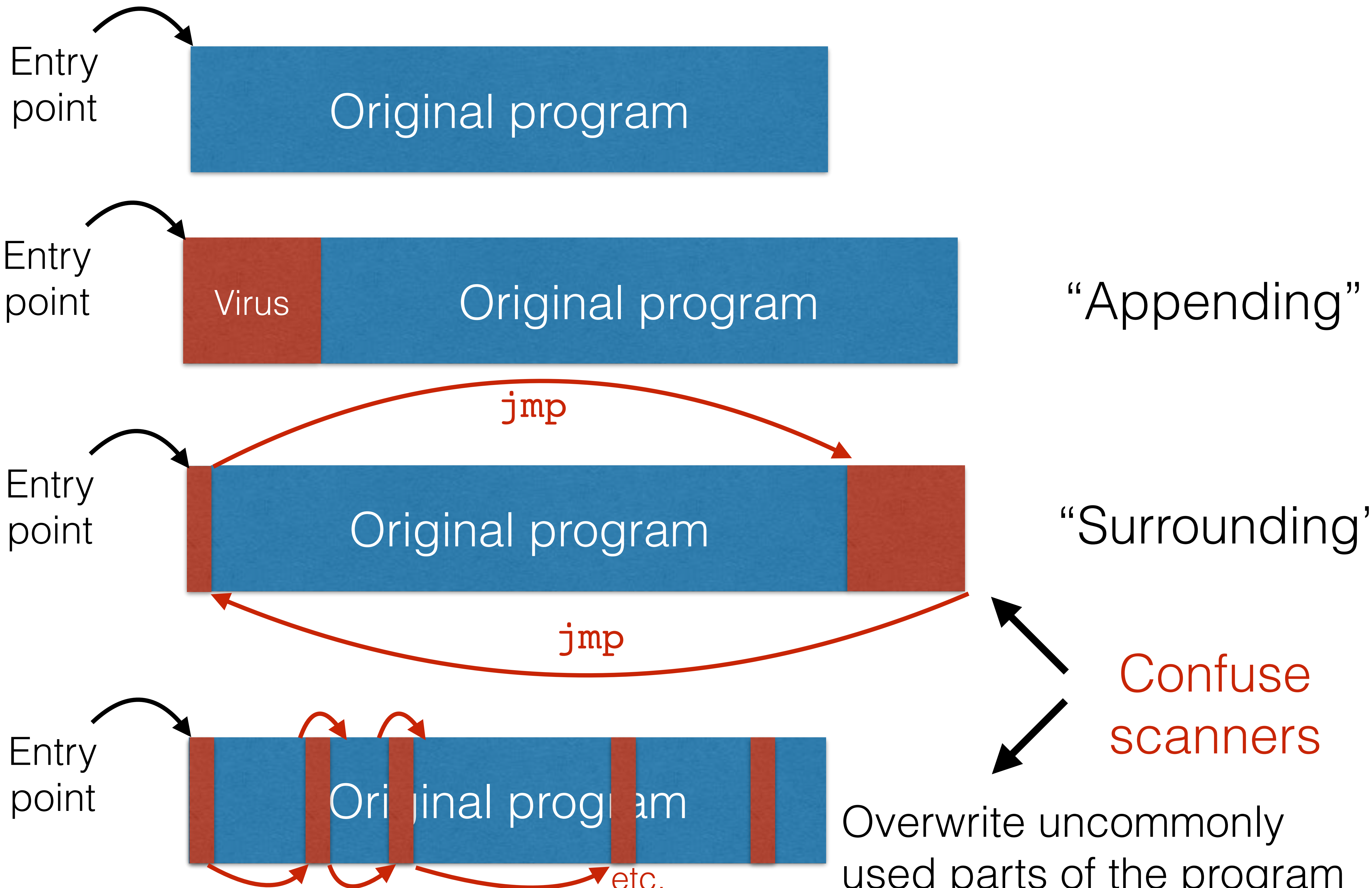
Security vendors have responded by updating their products with additional capabilities, such as file reputation and heuristics-based engines. They're also making upgrades to keep up with the latest technology trends, such as virtualization and cloud computing.

You are a virus writer

- Your goal is for your virus to spread far and wide
- How do you avoid detection by antivirus software that uses signatures?

1. Make signature **harder to find**

How viruses infect other programs



You are a virus writer

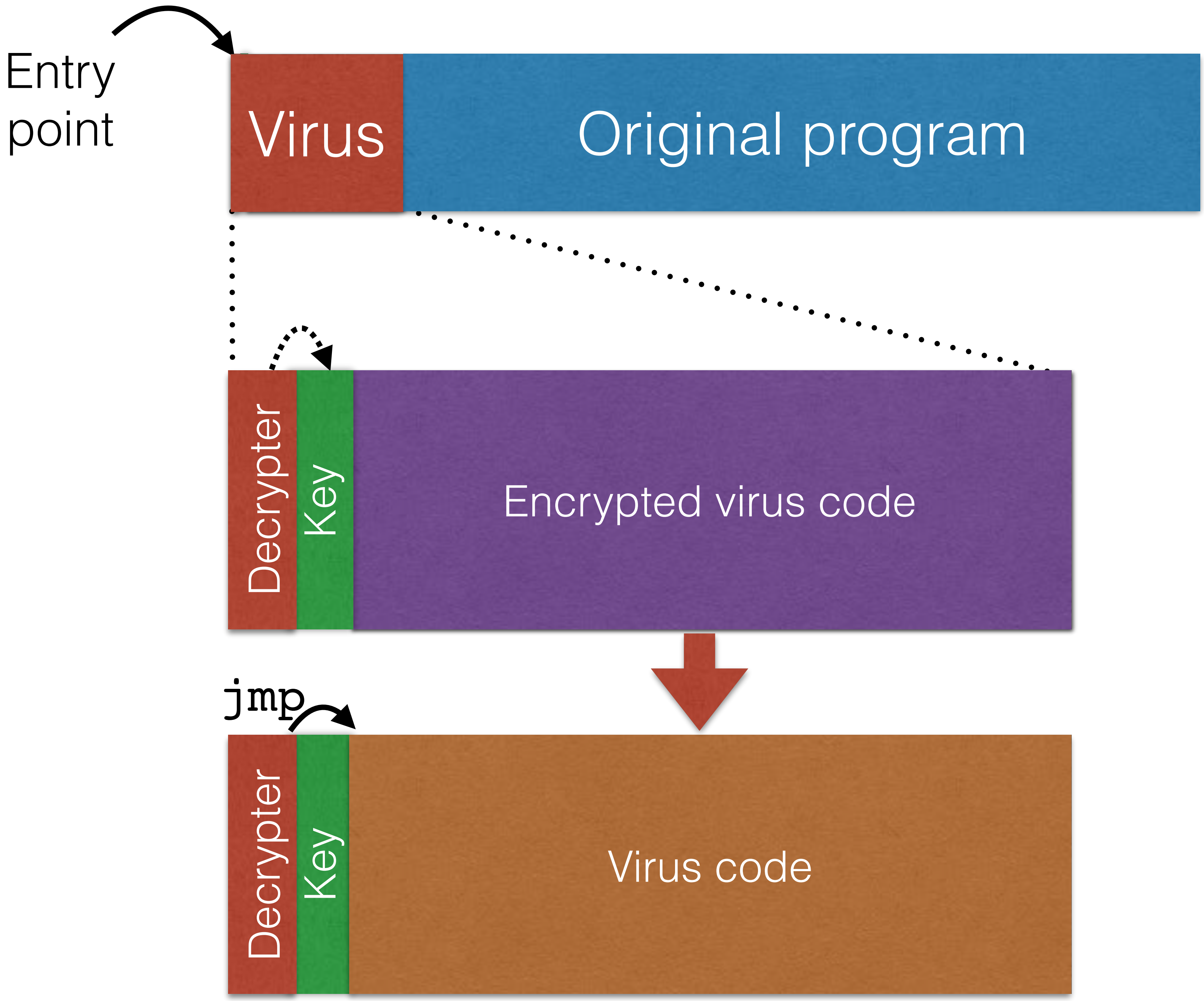
- Your goal is for your virus to spread far and wide
- How do you avoid detection by antivirus software that uses signatures?

1. Make signature harder to find
2. **Change code** to prevent defining a signature

***Mechanize* code changes:**
Goal: every time you inject your code, it looks different

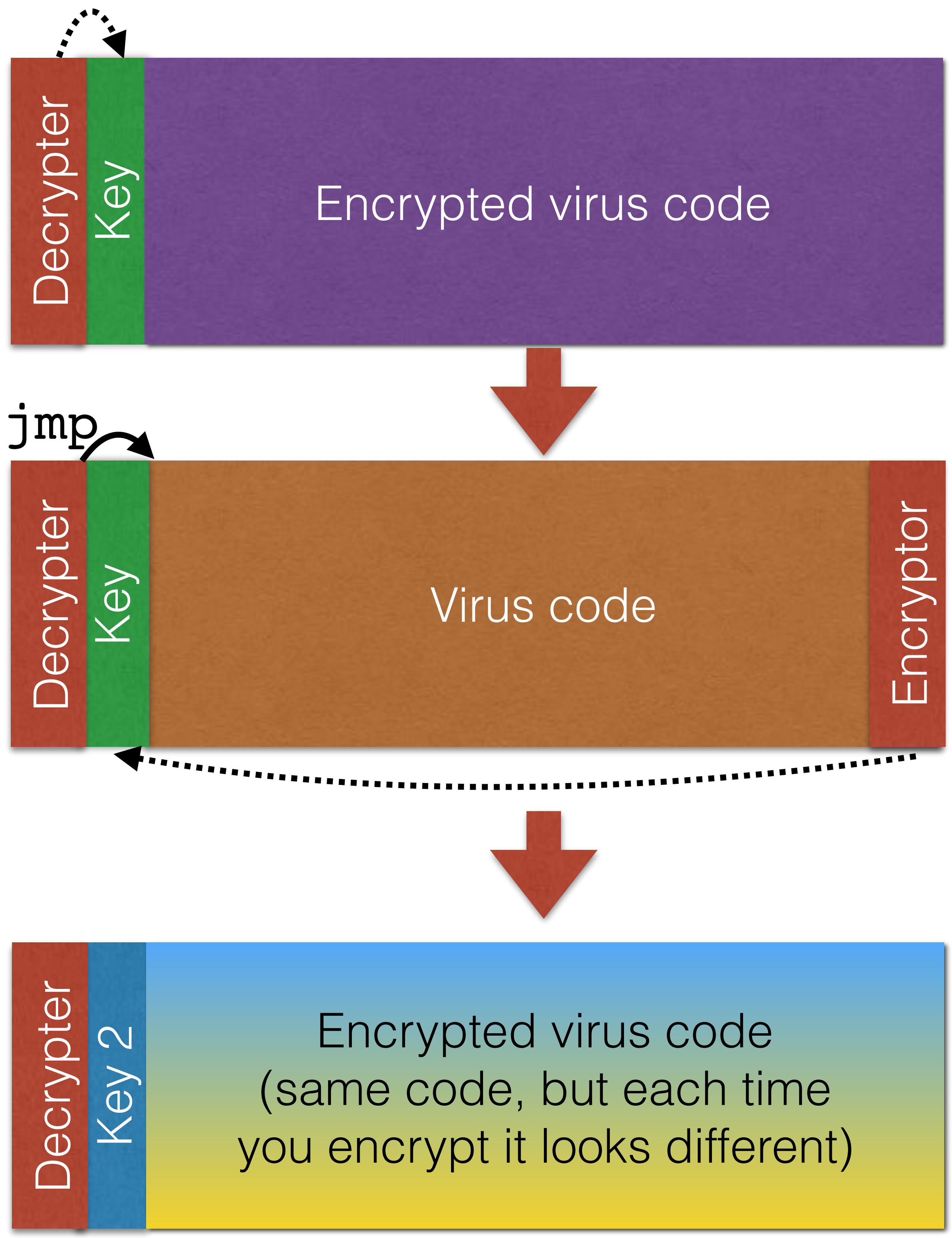
Polymorphic and metamorphic viruses

Polymorphic using encryption



Take over the entry point

Making it automatic



When used properly, encryption will yield a different, random output upon each invocation

Polymorphic viruses: Arms race

Now you are the antivirus writer: how do you detect?

- Idea #1: Narrow signature to catch the decrypter
  - Often very small: can result in many false positives
  - Attacker can spread this small code around and `jmp`
- Idea #2: Execute or statically analyze the suspect code to see if it decrypts.
  - How do you distinguish from common “packers” which do something similar (decompression)?
  - How long do you execute the code??

Now you are the *virus* writer again: how do you *evade*?

Polymorphic countermeasures

- Change the decrypter
  - Oligomorphic viruses: assemble decrypter from several interchangeable alternative pieces
  - True polymorphic viruses: can generate an endless number of decrypters
    - Different encryption methods
    - Random generation of confounds
    - Downside: inefficient

Metamorphic viruses

- Every time the virus propagates, generate a *semantically different* version of the code
  - Higher-level semantics remain the same
  - But the way it does it differs
    - Different machine code instructions
    - Different algorithms to achieve the same thing
    - Different use of registers
    - Different constants….

- How would you do this?
  - Include a code rewriter with your virus
  - Add a bunch of complex code to throw others off (then just never run it)

```
5A                pop  edx
BF04000000        mov  edi,0004h
8BF5              mov  esi,ebp
B80C000000        mov  eax,000Ch
81C288000000      add  edx,0088h
8B1A              mov  ebx,[edx]
899C8618110000mov  [esi+eax*4+00001118],ebx

58                pop  eax
BB04000000        mov  ebx,0004h
8BD5              mov  edx,ebp
BF0C000000        mov  edi,000Ch
81C088000000      add  eax,0088h
8B30              mov  esi,[eax]
89B4BA18110000mov  [edx+edi*4+00001118],esi
```

Figure 4: Win95/Regswap using different registers in new generations

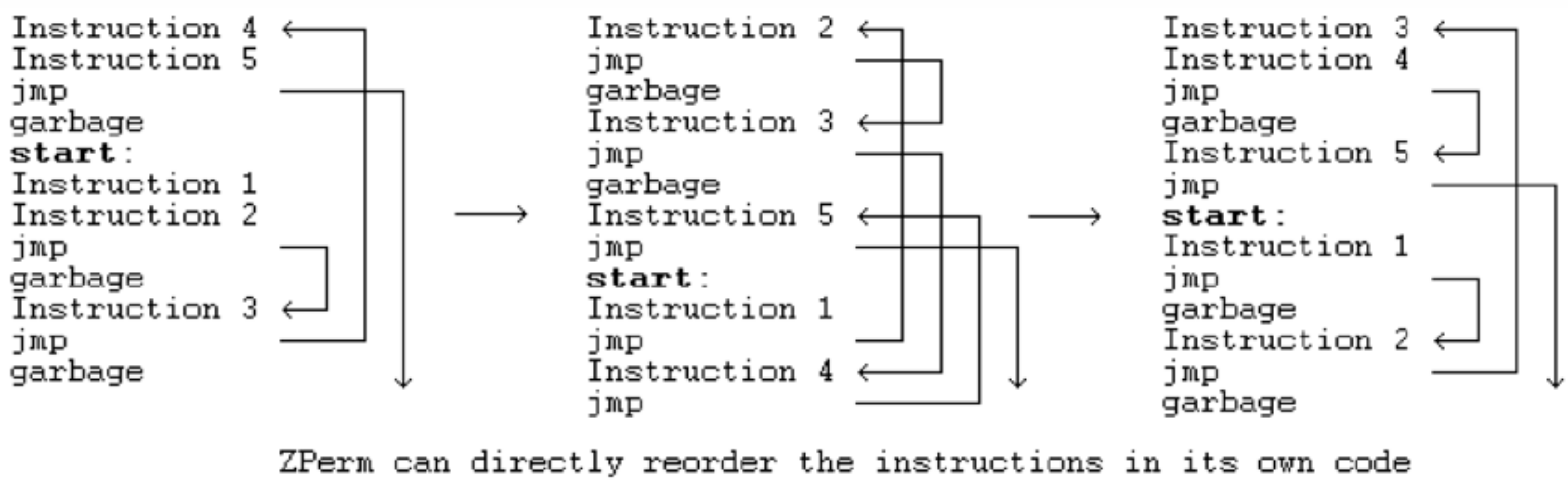
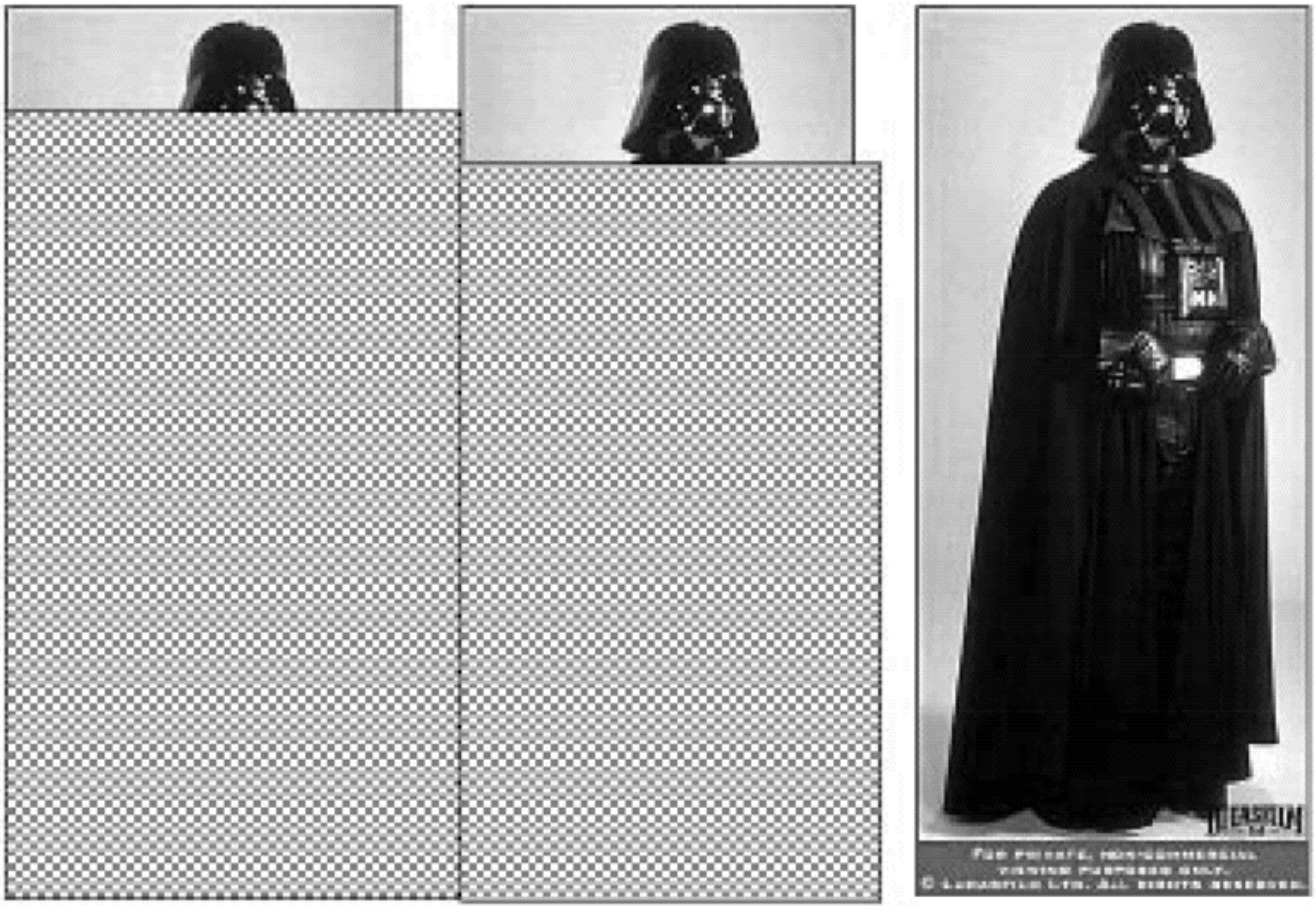


Figure 7 Zperm.A inserts JMP instruction into its code

```
a. An early generation:

C7060F000055   mov      dword ptr [esi],5500000Fh
C746048BEC5151 mov      dword ptr [esi+0004],5151EC8Bh

b. And one of its later generations:

BF0F000055     mov      edi,5500000Fh
893E           mov      [esi],edi
5F             pop      edi
52             push     edx
B640           mov      dh,40
BA8BEC5151     mov      edx,5151EC8Bh
53             push     ebx
8BDA           mov      ebx,edx
895E04         mov      [esi+0004],ebx

c. And yet another generation with recalculated ("encrypted") "con-
stant" data.

BB0F000055     mov      ebx,5500000Fh
891E           mov      [esi],ebx
5B             pop      ebx
51             push     ecx
B9CB00C05F     mov      ecx,5FC000CBh
81C1C0EB91F1   add      ecx,F191EBC0h ; ecx=5151EC8Bh
894E04         mov      [esi+0004],ecx
```

Figure 6: Example of code metamorphosis of Win32/Evol

Polymorphic



When can AV software successfully scan?

Figure 8: A partial or complete snapshot of polymorphic virus during execution cycle

Metamorphic

When can AV software successfully scan?

Figure 10: T-1000 of Terminator 2

Detecting metamorphic viruses?

Scanning isn't enough

- Need to analyze execution behavior
- Two broad stages in practice (both take place in a safe environment, like gdb or a virtual machine)
  1. AV company analyzes new virus to find behavioral signature
  2. AV system at end host analyzes suspect code to see if it matches the signature

Detecting metamorphic viruses

- Countermeasures
  - Change slowly (hard to observe pattern)
  - Detect if you are in a safe execution environment (e.g., gdb) and act differently

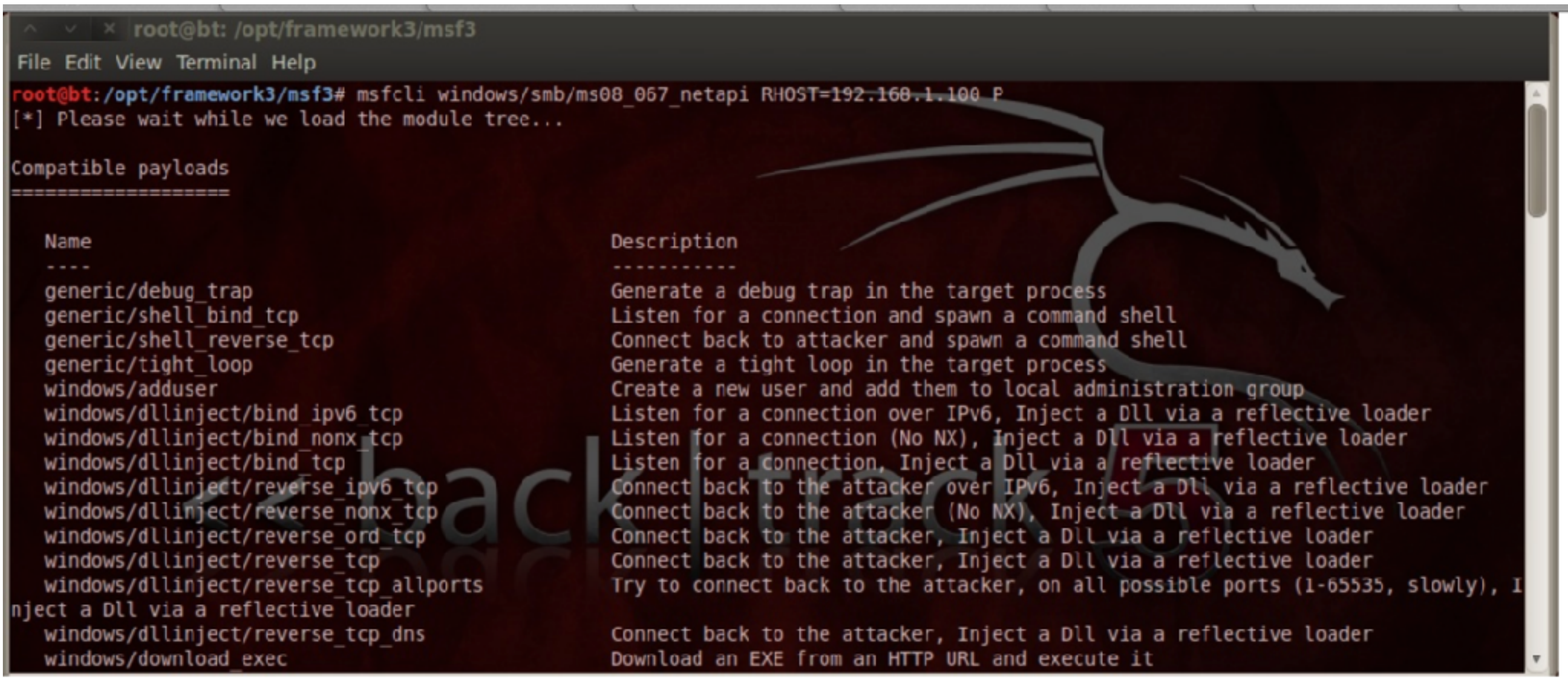
- Counter-countermeasures
  - **Detect detection** and skip those parts

- Counter-counter-counter…. Arms race

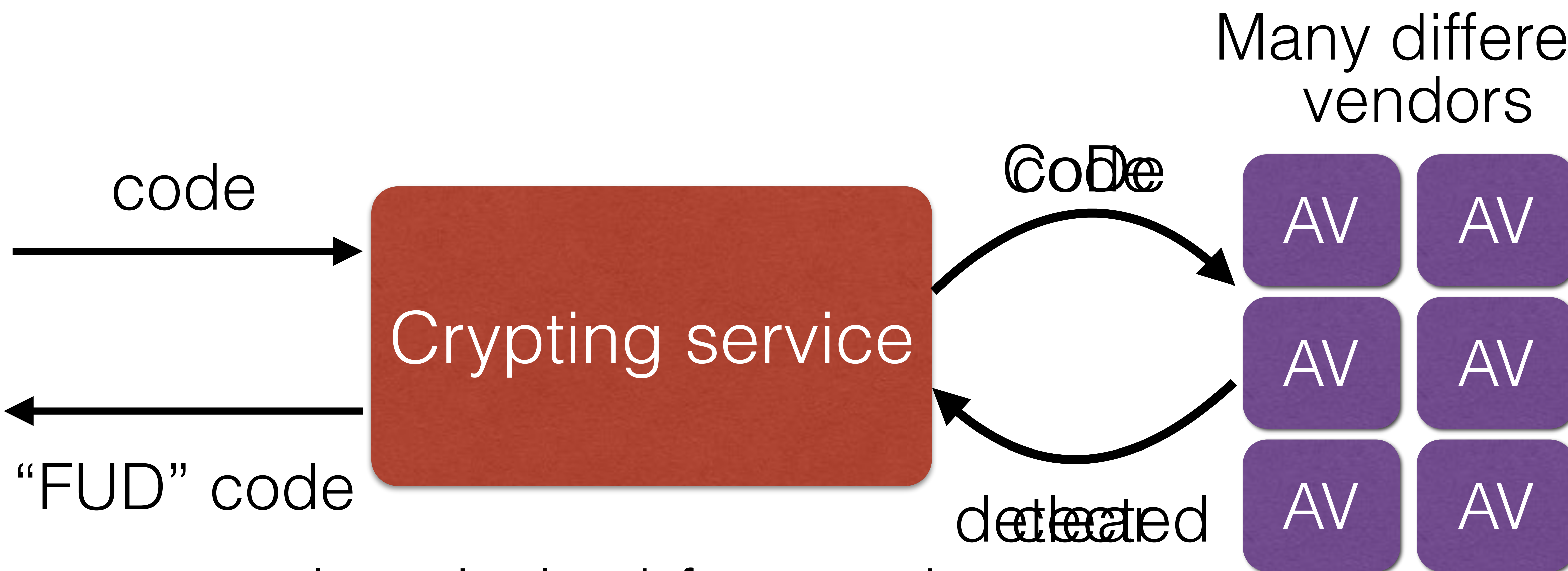
Attackers have the upper hand:
AV systems hand out signature *oracles*

Putting it all together sounds hard

- **Creating** a virus can be really difficult
  - Historically error prone
- But **using** them is easy: any scriptkiddy can use metasploit
  - Good news: so can any white hat pen tester



Crypting services

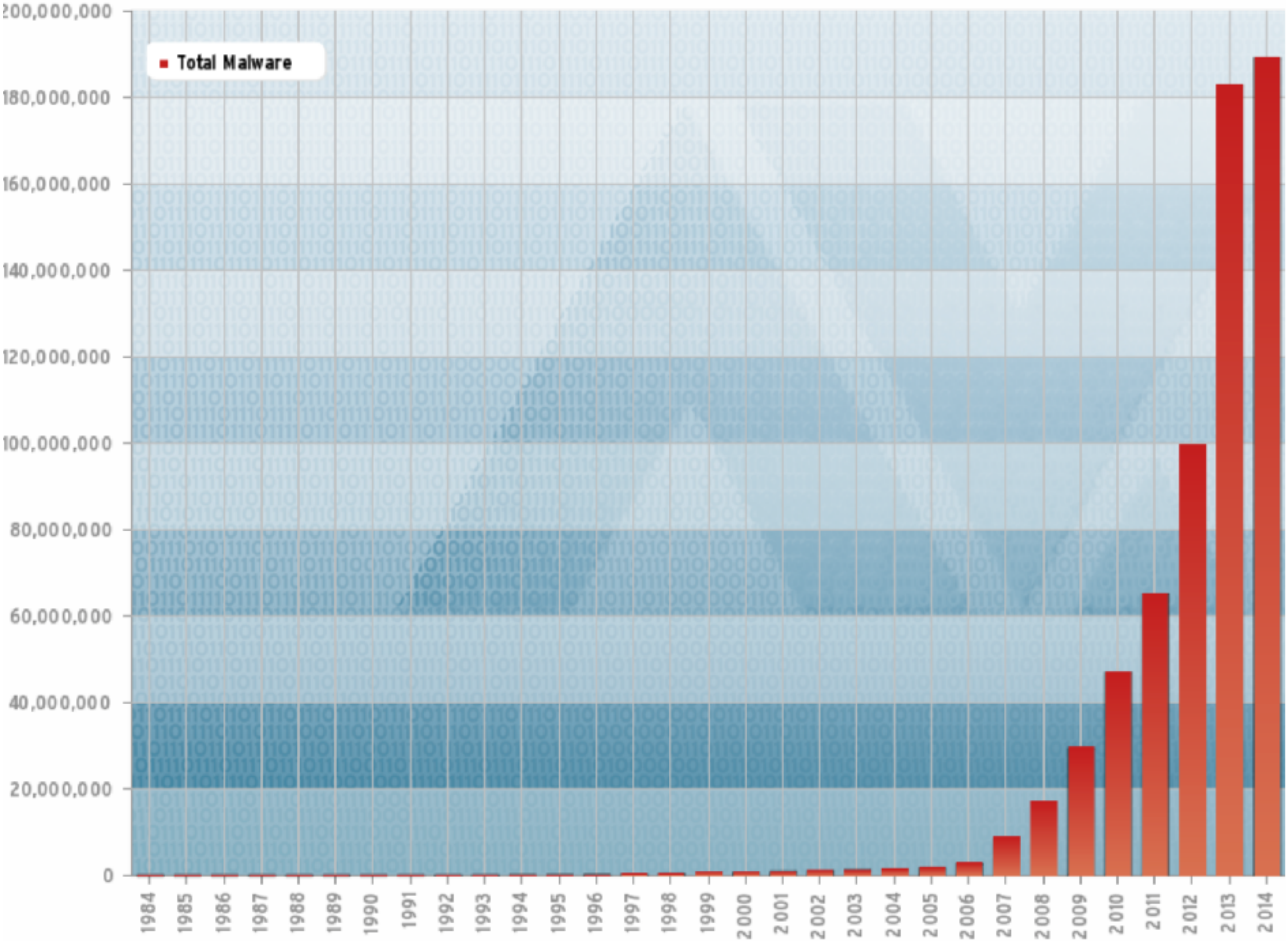


Iteratively obfuscate the code (encrypt + jmp + …)

Until the obfuscated code is “fully undetectable”

So how much malware is out there?

- Polymorphic and metamorphic viruses can make it easy to *miscount* viruses
- Take numbers with a grain of salt
  - Large numbers are in the AV vendors' best interest
- Previously, most malware was showy
  - Now primary goal is frequently to not get noticed



Last update: 01-21-2014 17:53

Copyright © AV-TEST GmbH, www.av-test.org

How do we clean up an infection?

- Depends what the virus did, but..
- May require restoring / repairing files
  - A service that antivirus companies sell
- What if the virus ran as root?
  - May need to rebuild the entire system
- So what, just recompile it?
  - What if the malware left a backdoor in your compiler?
    - Compile the malware back into the compiler
  - May need to use original media and data backups

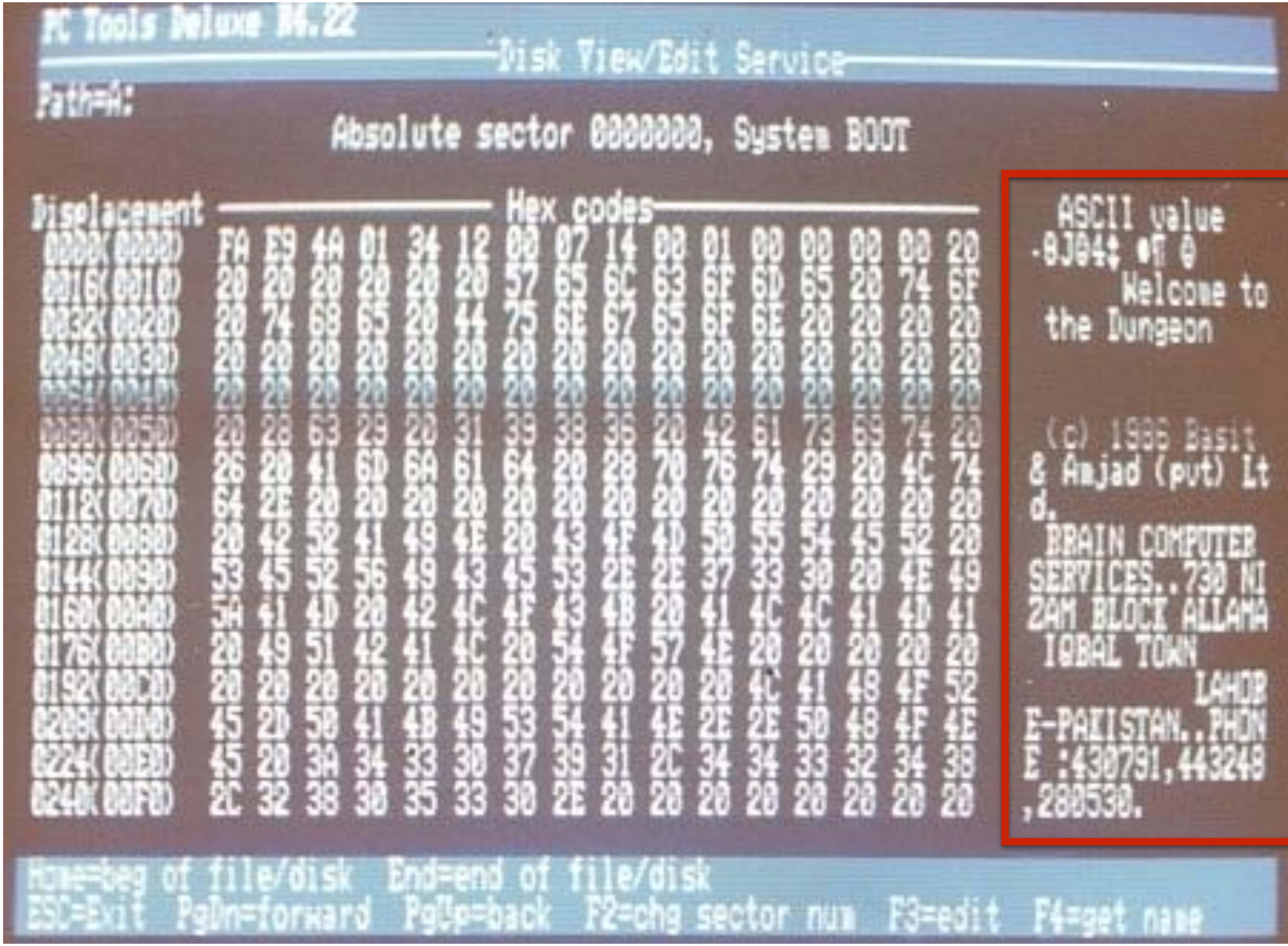
Virus case studies

Brain

First IBM PC virus (1987)

- Propagation method
  - Copies itself into the boot sector
  - Tells the OS that all of the boot sector is “faulty” (so that it won’t list contents to the user)
    - Thus also one of the first examples of a **stealth** virus
  - Intercepts disk read requests for 5.25” floppy drives
    - Sees if the 5th and 6th bytes of the boot sector are 0x1234
    - If so, then it’s already infected, otherwise, infect it

- Payload:
  - Nothing really; goal was just to spread (to show off?)
  - However, it served as the template for future viruses



Downloaded from wikipedia.org

Rootkits

- Recall: a rootkit is malicious code that takes steps to go undiscovered
  - By intercepting system calls, patching the kernel, etc.
  - Often effectively done by a man in the middle attack
- Rootkit revealer: analyzes the disk offline and through the online system calls, and compares
- Mark Russinovich ran a rootkit revealer and found a rootkit in 2005… **installed by a CD he had bought.**

Sony XCP rootkit

Detected 2005

- Goal: keep users from copying copyrighted material
- How it worked:
  - Loaded thanks to autorun.exe on the CD
  - Intercepted read requests for its music files
  - If anyone but Sony's music player is accessing them, then garble the data
  - Hid itself from the user (to avoid deletion)
- How it messed up
  - Morally: violated trust
  - Technically: Hid ***all files*** that started with "sys"
  - Seriously?: Uninstaller did not actually uninstall; introduced additional vulnerability instead

Worms

Controlling millions of hosts: ***Why?***

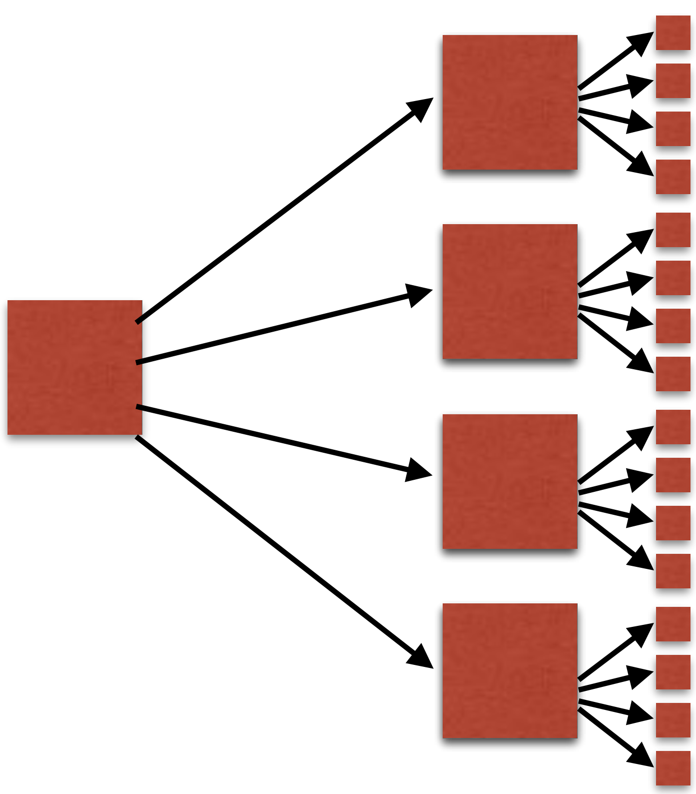
- Distributed Denial of Service (DDoS)
  - Generate network traffic from many sources..
  - .. to a single destination
  - .. with the intention of overloading their network
    - Consume too many resources for legitimate users to also use
- Steal sensitive information from millions of others
  - Even a small fraction of unprotected people ⇒ $
- Confuse and disrupt

Controlling millions of hosts: ***How?***

- Worm: *self*-propagates by arranging to have itself *immediately* executed
  - At which point it creates a new, additional instance of itself
- Typically infects by altering *running* code
  - No user intervention required
- Like viruses, propagation and payload are orthogonal

Self-propagation

- Goal: spread as quickly as possible
- The key is ***parallelization***
  - Without being triggered by human interaction!



Propagation

(1) Targeting: how does the worm **find new prospective victims**?

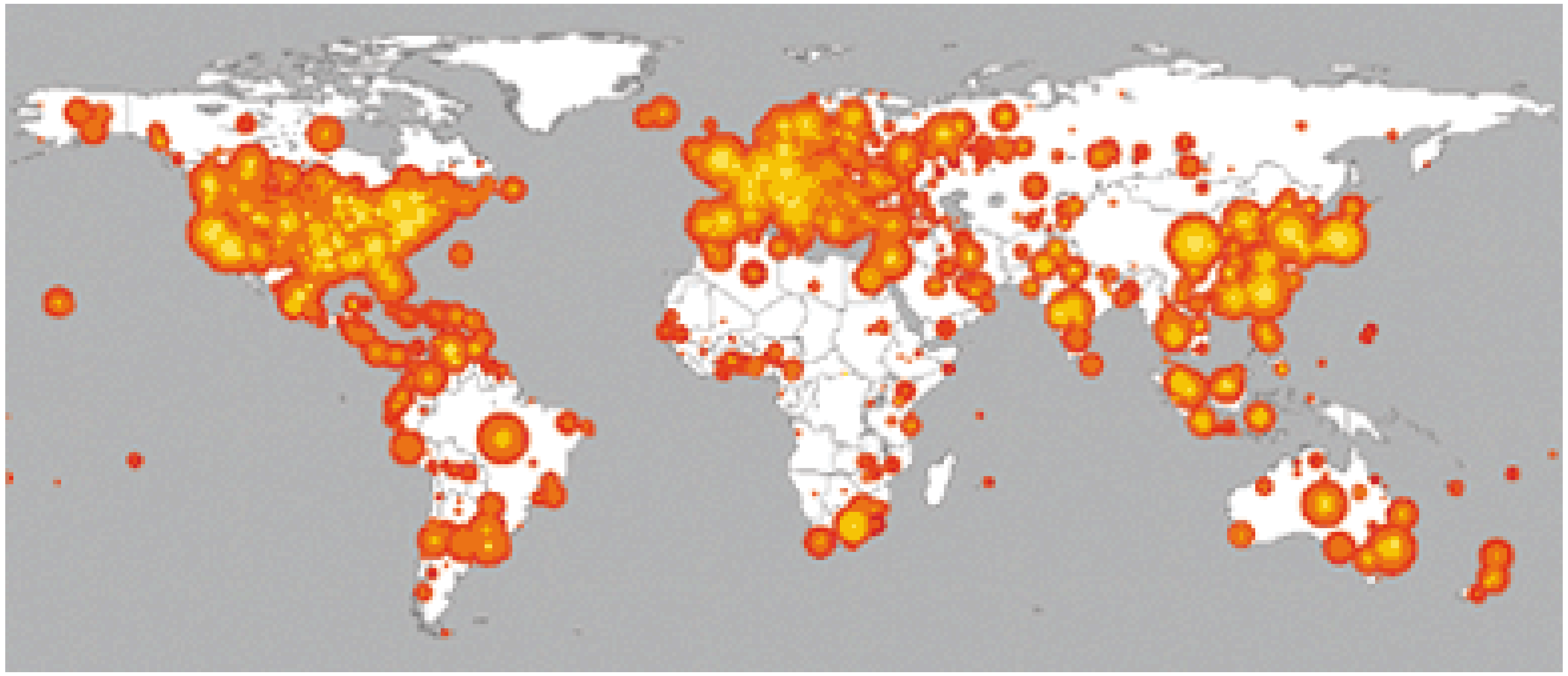
(2) Exploit: how does the worm get code to **automatically run**?

Morris worm — 1988

- Accidentally more agressive than intended
  - 6-10% of all internet hosts infected
- Scan local subnet; exploit fingerd overflow
  - Crack passwords
  - Phone home

Code Red — 2001

- Exploited overflow in MS-IIS server
  - At peak, more than 2000 new infections/minute
- Before 20th of month, propagate
  - After 20th, attack whitehouse.gov



Copyright UC Regents, Jeff Brown for CAIDA, UCSD.

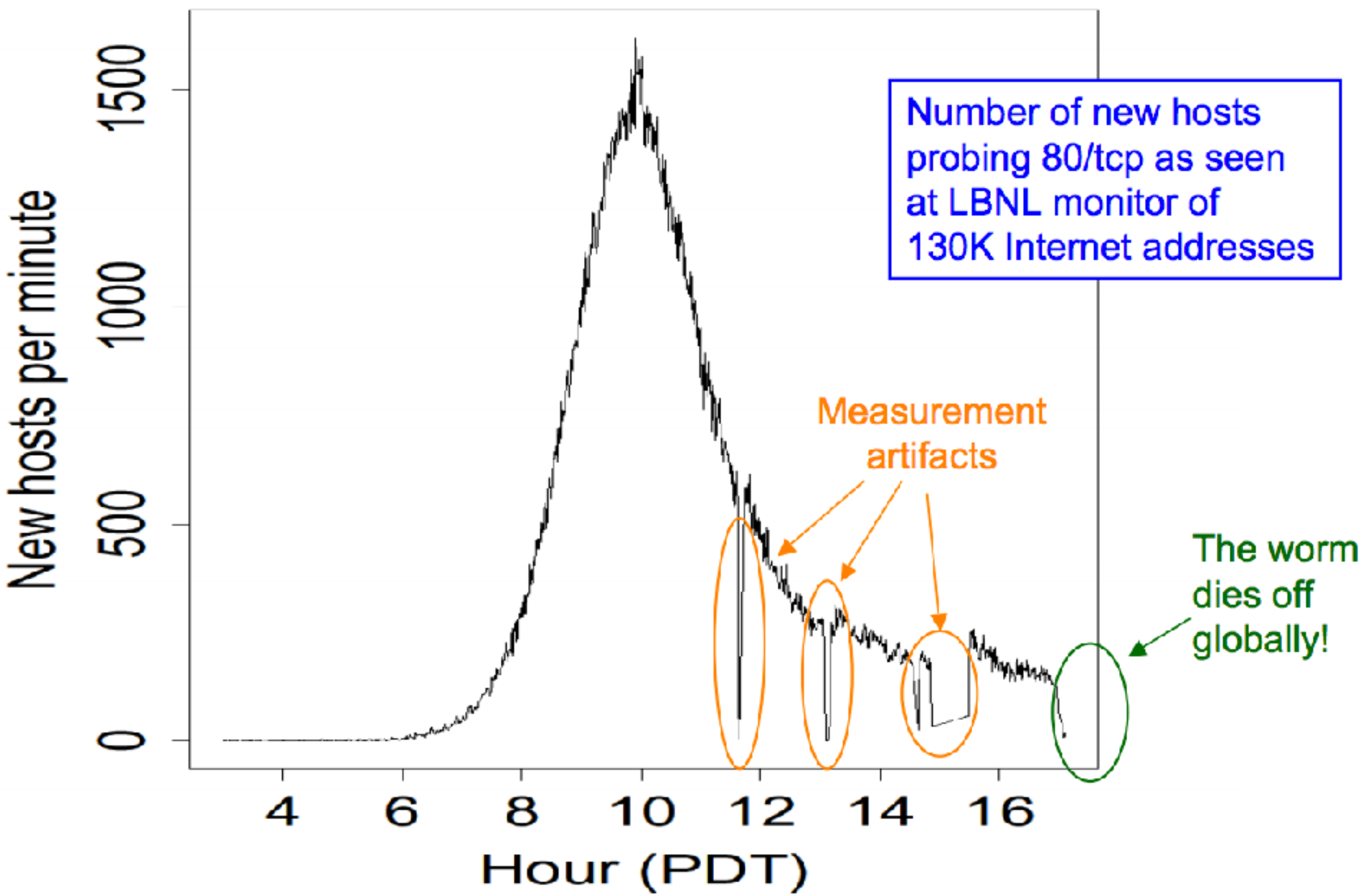
CodeRed Propagation

- Spread by randomly scanning the entire 32-bit IP address space
  - Pick a pseudorandom 32-bit number = IP addr
  - Send exploit packet to that address
  - Repeat
- This is a very common worm technique
- Each instance used the ***same random seed***
  - What does this mean in practice?

More CodeRed

- If found c:\notworm then do nothing
- Whitehouse.gov **changed** its IP address
  - Made the attack portion useless
- Revision one week later: random number generator was **seeded** properly
  - No attack function, installs backdoor instead
  - By then many but not all hosts patched

Growth of Code Red Worm



Credit: Vern Paxson's CS 161 at Berkeley

Modeling worm spread

- Classic epidemic model: ***Susceptible-Infectable***

$$\frac{dI}{dt} = \beta \cdot I \cdot \frac{S}{N}$$

Change in #infected over time

Total attempted contacts per unit time

Proportion of contacts expected to succeed

S(t) = Susc. hosts at time t
I(t) = Infected hosts at time t
N = size of vuln. population = S(t) + I(t)
β = contact rate

Modeling worm spread

- Classic epidemic model: ***Susceptible-Infectable***

$$\frac{dI}{dt} = \beta \cdot I \cdot \frac{S}{N}$$

Change in #infected over time

Total attempted contacts per unit time

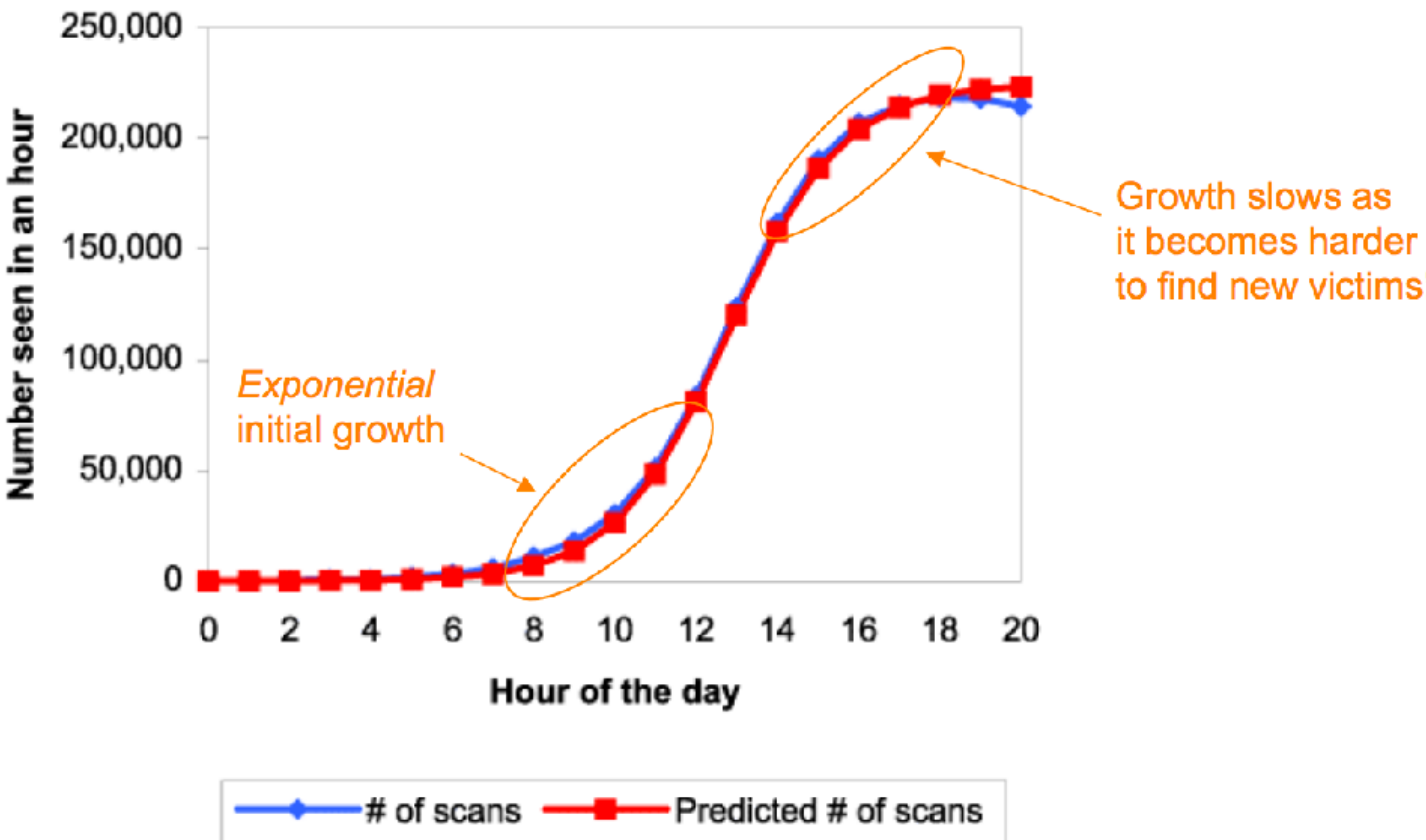
Proportion of contacts expected to succeed

Rewriting using i(t) = $\mathbf{I}$(t) / N and S = N - $\mathbf{I}$:

$$\frac{di}{dt} = \beta \cdot i \cdot (1-i) \quad \Rightarrow \quad i(t) = \frac{e^{\beta t}}{1+e^{\beta t}}$$

Fraction infected grows as a *logistic*

Fitting the model to Code Red

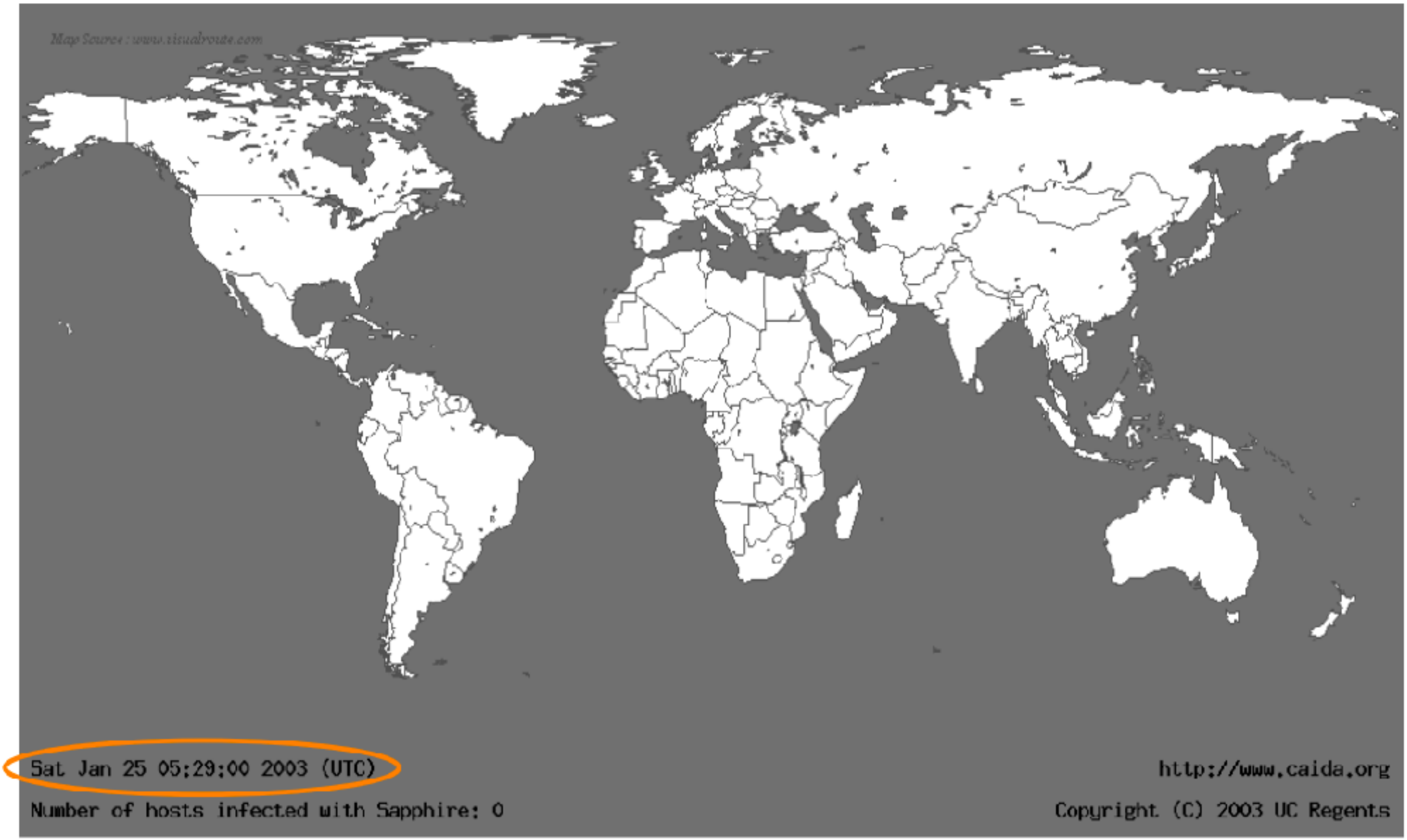


Credit: Vern Paxson's CS 161 at Berkeley

SQL Slammer (2003)

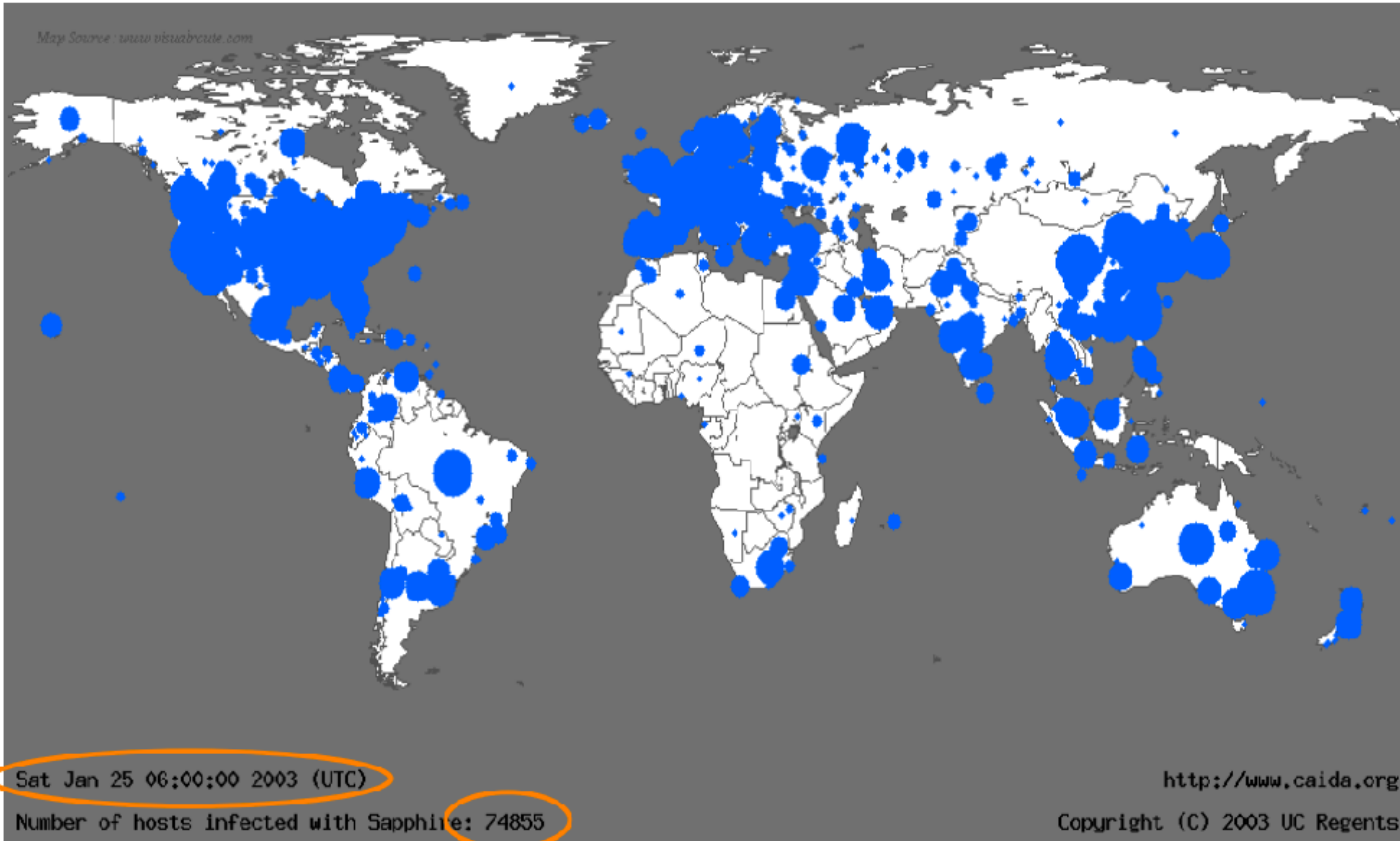
- Exploited overflow in MS SQL Server
  - Patch had been available for > 6 months
- Connectionless UDP rather than TCP
  - Entire worm fit in a single packet!
- When scanning, the worm could “fire and forget”
  - Stateless!
- Infected 75k machines in 10 minutes
  - At its peak, doubled every <u>8.5 seconds</u>

Life just before Slammer



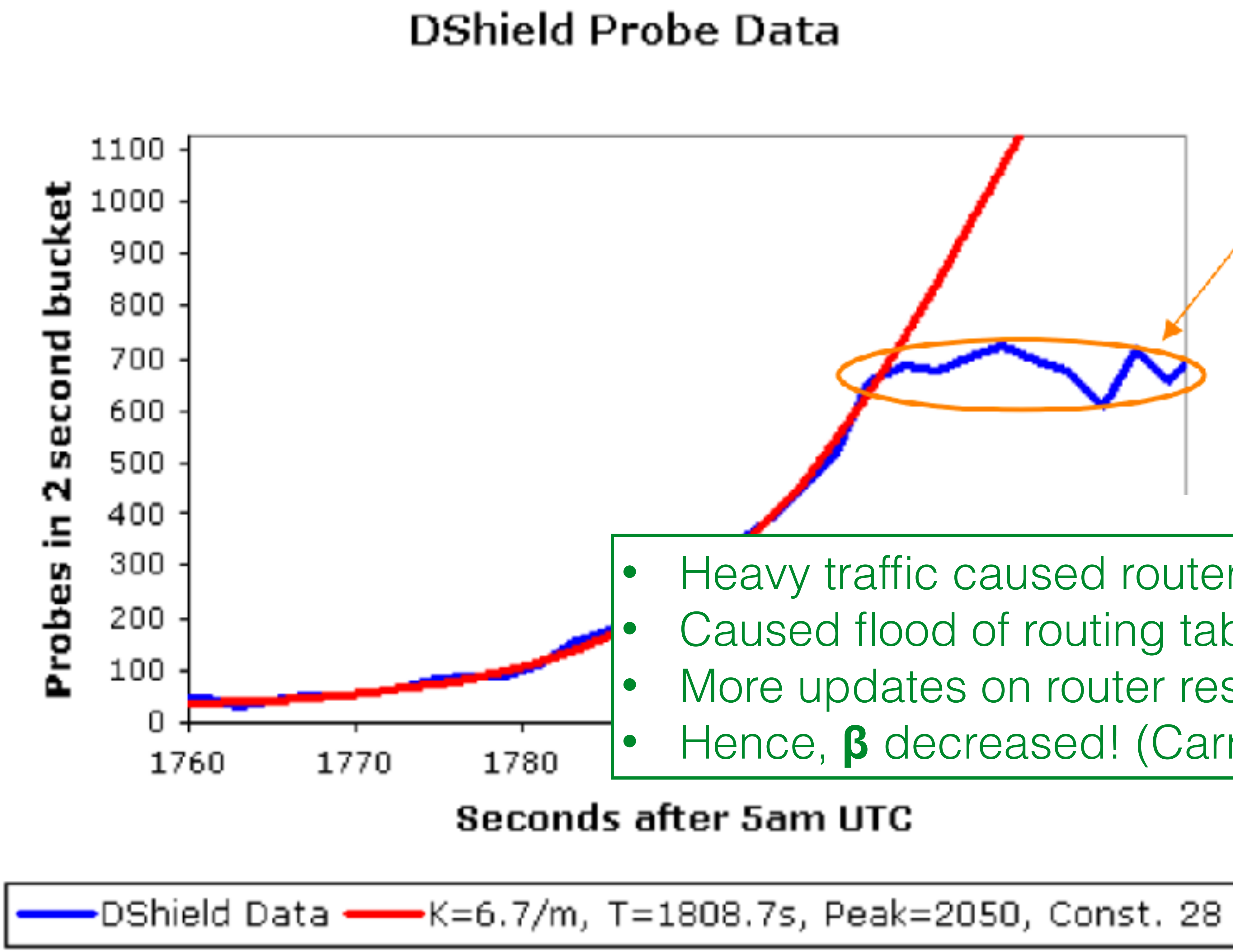
Credit: Vern Paxson's CS 161 at Berkeley

Life just after Slammer



Credit: Vern Paxson's CS 161 at Berkeley

Slammer's growth



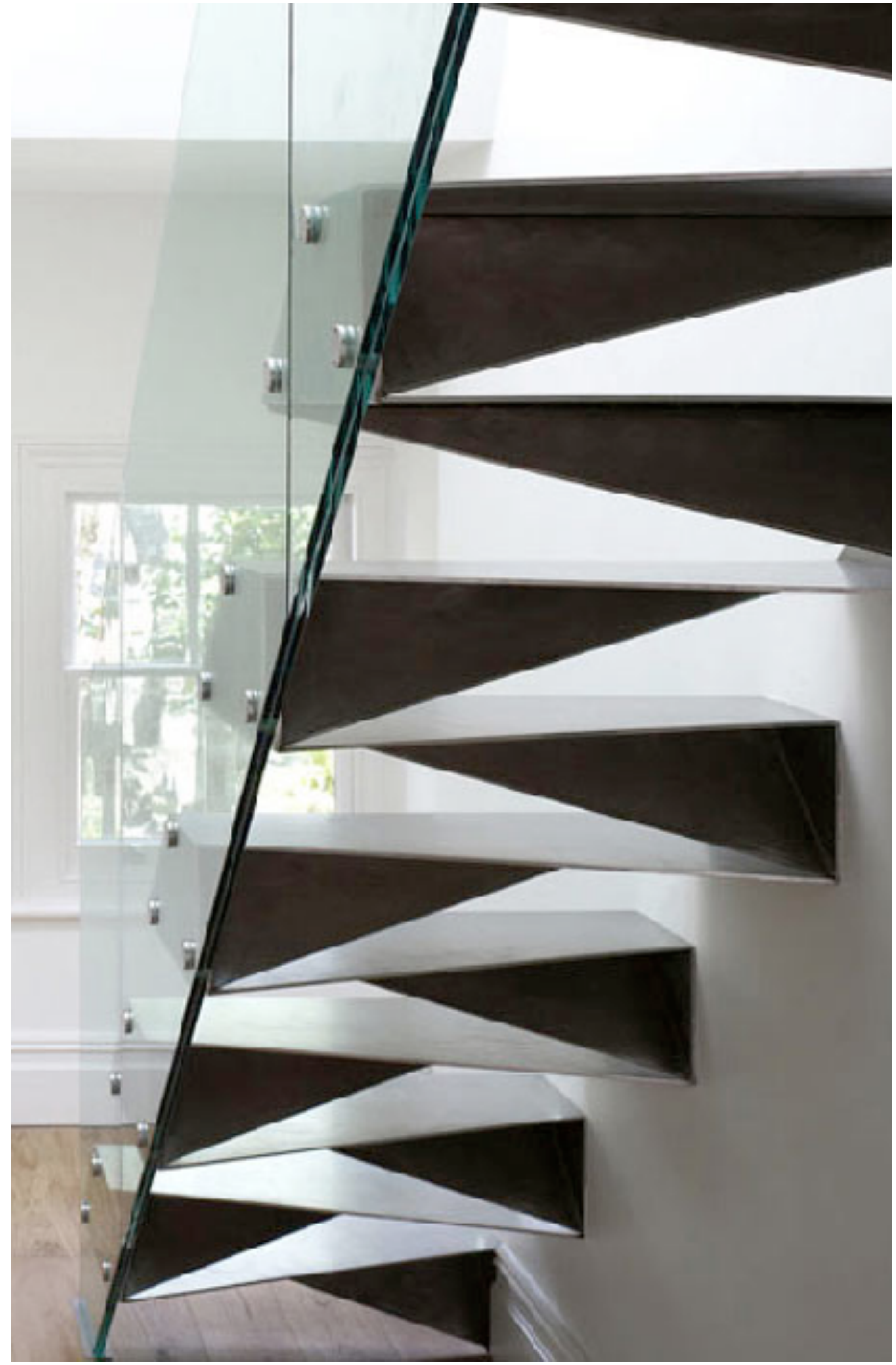
What could have caused growth to deviate from the model?

Hint: at this point the worm is generating *55,000,000 scans/sec*

- Heavy traffic caused routers to collapse
- Caused flood of routing table updates
- More updates on router restart
- Hence, **β** decreased! (Carrying capacity)

Credit: Vern Paxson's CS 161 at Berkeley

“Modern”
Malware



- Note that most of these examples are old, why?
    - Maybe the problem is solved? (Hint: no)
- Instead, new era of malware
    - Old: Pride, anger, destruction, low-level politics
    - New: Economics, governments, espionage
    - How does this change the game?

- Didn’t change: Spread fast, avoid detection
- New goals:
  - Avoid detection ***longer****;* persistence
  - Exfil key data
  - Maintain command-control (remote admin)
- New infection vectors
  - Web security, coming next week

Botnets (More later)

- Infect many hosts; maintain control
- Sell these hosts as resources
  - To send spam, mine bitcoin, turn on webcams, install keyloggers

Stuxnet: Propagation

June 2010

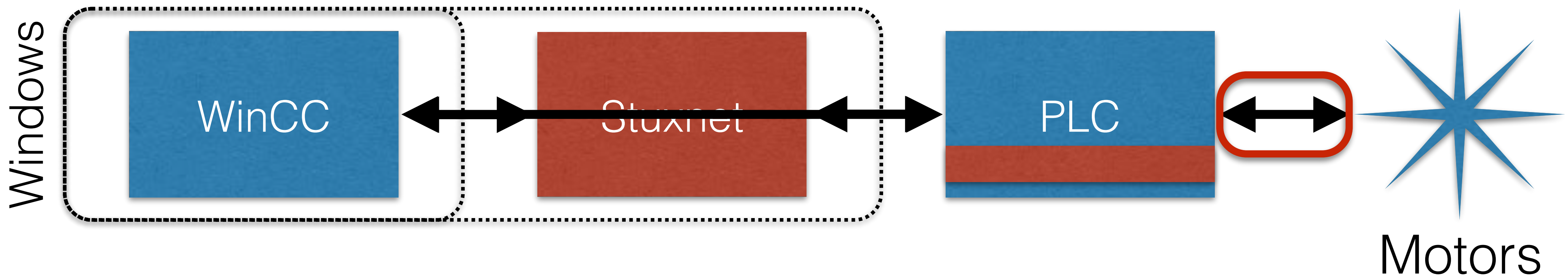
- Virus: initially spread by infected USB stick
  - Once inside network, acted as a worm, spreading quickly
- Exploited **four** zero-day exploits
  - Zero-day: Known to only the attacker until the attack
  - Typically, one zero-day is enough to profit
  - Four was unprecedented
    - Immense cost and sophistication on behalf of the attacker
- Rootkit: Installed *signed* device drivers
  - Thereby avoiding user alert when installing
  - Signed with certificates stolen from two Taiwanese CAs

Stuxnet: Payload

- Do nothing
- Unless attached to particular models of frequency converter drives that operate at 807-1210Hz
  - You know, like those in Iran and Finland
  - .. those ones that are used to operate centrifuges
  - .. for producing enriched uranium for nuclear weapons
- In which case, slowly increase the freq to 1410Hz
  - You know, enough to break the centrifuge
  - .. all the while sending “looks good to me” readings to the user
  - .. then drop back to normal range

Stuxnet: Payload

- Target industrial control systems: overwrite programmable logic boards
- Man-in-the-middle between Windows and Siemens control systems; looked like it was working properly to the operator



- In reality, it sped up and slowed down the motors
- Result: Destroy (or at least decrease the productivity of) nuclear centrifuges

Stuxnet: Fallout

- Iran denied they had been hit by Stuxnet
- Then claimed they were, but had contained it
- Now believed it took out 1k of Iran’s 5k centrifuges
- Security experts believe the U.S. did it (possibly along with Israel) due to its sophistication and cost
- **Legitimized cyber warfare**

Detecting modern malware

- Connection to known C&C server
  - Counter: Cycle domain and use dynamic DNS
  - Re-counter: Block connections to new domains
- “Custom” TCP and UDP
- Generating direct email (vs. traversing mail server)
- Anomaly detection

Detection, not prevention

All subject to arms race!

Malware summary

- Technological arms race between those who wish to detect and those who wish to evade detection
- Started off innocuously
- Became professional, commoditized
  - Economics, cyber warfare, corporate espionage
- Advanced detection: based on behavior, anomalies
  - Must react to attacker responses