Outline

- Worm propagation
  - Worm examples
  - Propagation models
- Detection methods
  - Traffic patterns: Autograph, EarlyBird, Polygraph
  - Watch attack: TaintCheck and Sting
  - Look at vulnerabilities: Generic Exploit Blocking

Worm

- A worm is self-replicating software designed to spread through the network
  - Typically exploit security flaws in widely used services
  - Often conscripts machine into bot network
  - May cause enormous collateral damage
    - Access sensitive information
    - Corrupt files
    - Cause malfunction, overload, etc.
- Worm vs Virus vs Trojan horse
  - A virus is code embedded in a file or program
  - Viruses and Trojan horses rely on human intervention
  - Worms are self-contained and may spread autonomously

Cost of worm attacks

- Morris worm, 1988
  - Infected approximately 6,000 machines
    - 10% of computers connected to the Internet
  - cost ~ $10 million in downtime and cleanup
- Code Red worm, July 16 2001
  - Direct descendant of Morris’ worm
  - Infected more than 500,000 servers
    - Programmed to go into infinite sleep mode July 28
  - Caused ~ $2.6 Billion in damages,
- Love Bug worm: $8.75 billion

Statistics: Computer Economics Inc., Carlsbad, California

Internet Worm (First major attack)

- Released November 1988
  - Program spread through Digital, Sun workstations
  - Exploited Unix security vulnerabilities
    - VAX computers and SUN-3 workstations running versions 4.2 and 4.3 Berkeley UNIX code
- Consequences
  - No immediate damage from program itself
  - Replication and threat of damage
    - Load on network, systems used in attack
    - Many systems shut down to prevent further attack

Internet Worm Description

- Two parts
  - Program to spread worm
    - look for other machines that could be infected
    - try to find ways of infiltrating these machines
  - Vector program (99 lines of C)
    - compiled and run on the infected machines
    - transferred main program to continue attack
- Security vulnerabilities
  - fingerd – Unix finger daemon
  - sendmail - mail distribution program
  - Trusted logins (.rhosts)
  - Weak passwords

Three ways the worm spread

- Sendmail
  - Exploit debug option in sendmail to allow shell access
- Fingerd
  - Exploit a buffer overflow in the fgets function
  - Apparently, this was the most successful attack
- Rsh
  - Exploit trusted hosts
  - Password cracking

sendmail

- Worm used debug feature
  - Opens TCP connection to machine's SMTP port
  - Invokes debug mode
  - Sends a RCPT TO that pipes data through shell
  - Shell script retrieves worm main program
    - places 40-line C program in temporary file called x$$,l1.c where $$ is current process ID
    - Compiles and executes this program
    - Opens socket to machine that sent script
    - Retrieves worm main program, compiles it and runs

fingerd

- Written in C and runs continuously
- Array bounds attack
  - Fingerd expects an input string
  - Worm writes long string to internal 512-byte buffer
- Attack string
  - Includes machine instructions
  - Overwrites return address
  - Invokes a remote shell
  - Executes privileged commands

Remote shell

- Unix trust information
  - /etc/host.equiv – system wide trusted hosts file
  - /.rhosts and ~/.rhosts – users' trusted hosts file
- Worm exploited trust information
  - Examining files that listed trusted machines
  - Assume reciprocal trust
    - If X trusts Y, then maybe Y trusts X
- Password cracking
    - Worm was running as daemon (not root) so needed to break into accounts to use .rhosts feature
    - Dictionary attack
    - Read /etc/passwd, used ~400 common password strings

The worm itself

- Program is called 'sh'
  - Clobbers argv array so a 'ps' will not show its name
  - Opens its files, then unlinks (deletes) them so can't be found
    - Since files are open, worm can still access their contents
- Tries to infect as many other hosts as possible
  - When worm successfully connects, forks a child to continue the infection while the parent keeps trying new hosts
- Worm did not:
  - Delete system's files, modify existing files, install trojan horses, record or transmit decrypted passwords, capture superuser privileges, propagate over UUCP, X.25, DECNET, or BITNET

Detecting Morris Internet Worm

- Files
  - Strange files appeared in infected systems
  - Strange log messages for certain programs
- System load
  - Infection generates a number of processes
  - Systems were reinfected => number of processes grew and systems became overloaded
    - Apparently not intended by worm’s creator

Thousands of systems were shut down

Stopping the worm

- System admins busy for several days
  - Devised, distributed, installed modifications
- Perpetrator
  - Student at Cornell; turned himself in
  - Sentence: community service and $10,000 fine
    - Program did not cause deliberate damage
    - Tried (failed) to control # of processes on host machines
- Lessons?
  - Security vulnerabilities come from system flaws
  - Diversity is useful for resisting attack
  - “Experiments” can be dangerous

Sources for more information

- Eugene H. Spafford, The Internet Worm: Crisis and Aftermath, CACM 32(6) 678-687, June 1989
- Page, Bob, "A Report on the Internet Worm", http://www.ee.ryerson.ca:8080/~elf/hack/iworm.html

Some historical worms of note

Worm	Date	Distinction
Morris	11/88	Used multiple vulnerabilities, propagate to “nearby” sys
ADM	5/98	Random scanning of IP address space
Ramen	1/01	Exploited three vulnerabilities
Lion	3/01	Stealthy, rootkit worm
Cheese	6/01	Vigilante worm that secured vulnerable systems
Code Red	7/01	First sig Windows worm; Completely memory resident
Walk	8/01	Recompiled source code locally
Nimda	9/01	Windows worm: client-to-server, c-to-c, s-to-s, …
Scalper	6/02	11 days after announcement of vulnerability; peer-to-peer network of compromised systems
Slammer	1/03	Used a single UDP packet for explosive growth

Kienzle and Elder

Increasing propagation speed

- Code Red, July 2001
  - Affects Microsoft Index Server 2.0,
    - Windows 2000 Indexing service on Windows NT 4.0.
    - Windows 2000 that run IIS 4.0 and 5.0 Web servers
  - Exploits known buffer overflow in Idq.dll
  - Vulnerable population (360,000 servers) infected in 14 hours
- SQL Slammer, January 2003
  - Affects in Microsoft SQL 2000
  - Exploits known buffer overflow vulnerability
    - Server Resolution service vulnerability reported June 2002
    - Patched released in July 2002 Bulletin MS02-39
  - Vulnerable population infected in less than 10 minutes

Code Red

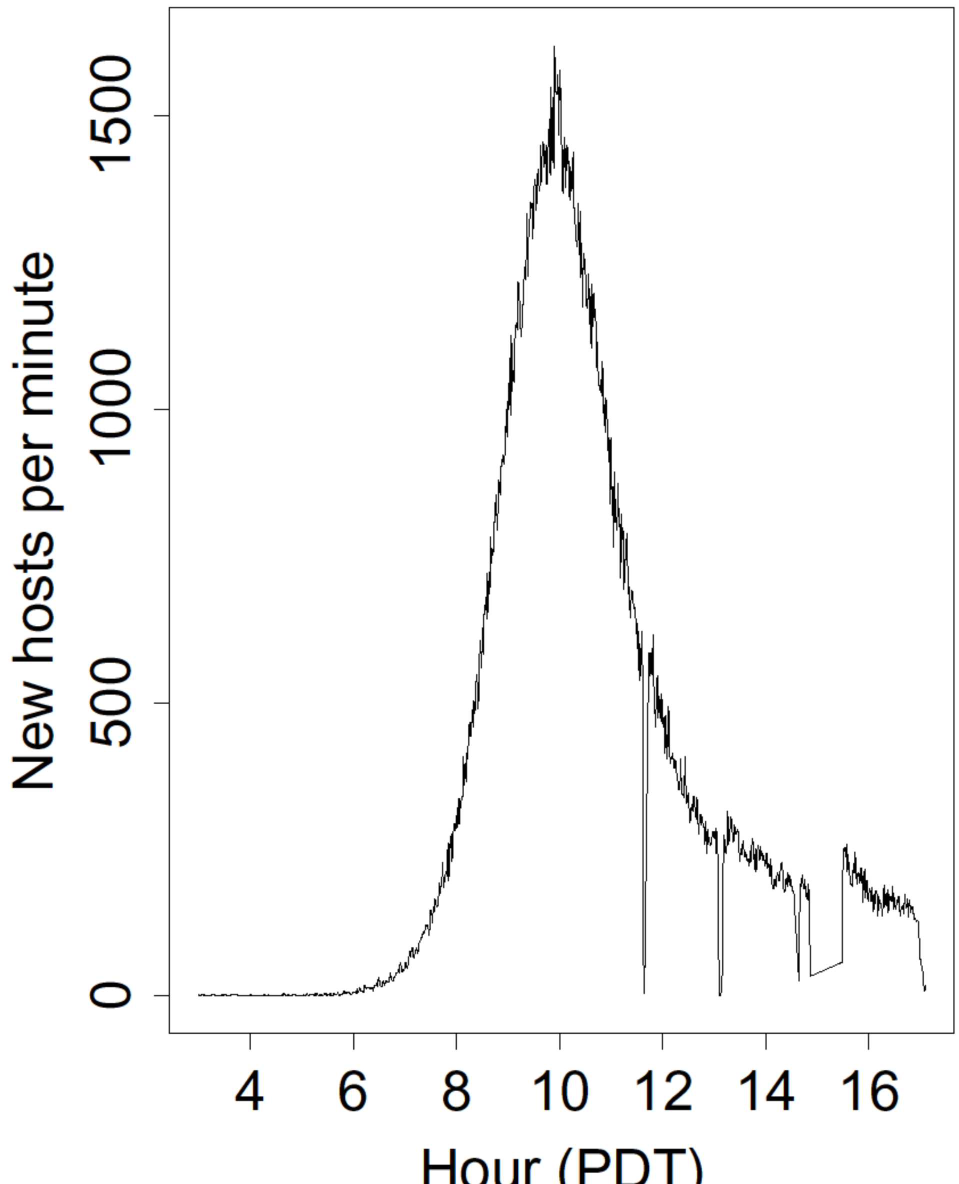
- Code Red I released July 12, 2001
  - If before 20th of month, scans IP addresses in fixed, pseudo-random order to find other targets
  - After 20th of month, mount DDOS attack
  - Send code as an HTTP request exploiting overflow
  - Just memory resident (rebooting clears infection)
- When executed,
  - Just sleep if C:\Notworm exists
  - Creates new threads to propagate infection

Code Red of July 12 and July 19

- Code Red I
  - 1st through 20th month: Spread
    - via pseudo-random scan of 32-bit IP addr space
  - 20th through end of each month: attack.
    - Flooding attack against 198.137.240.91 (*www.whitehouse.gov)*
  - Failure to seed random number generator ⇒ *linear growth*
- July 19th: Code Red I v2
  - White House responds to threat of flooding attack by changing the address of *www.whitehouse.gov*
  - Causes Code Red to die for date ≥ 20th of the month.
  - But: this time random number generator correctly seeded

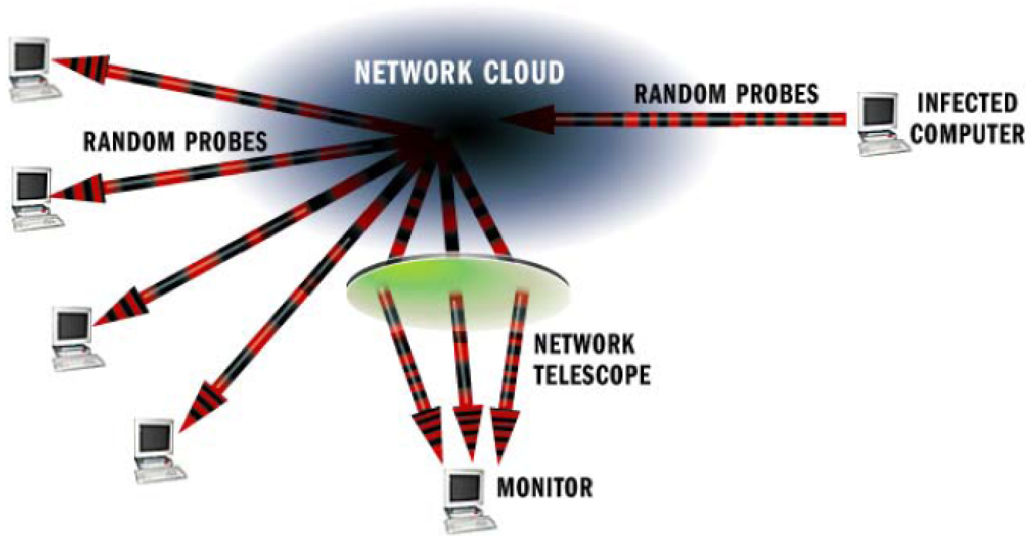
Slides: Vern Paxson

Growth of Code Red Worm



Slide: Vern Paxson

Measuring activity: network telescope



- Monitor cross-section of Internet address space, measure traffic
  - "Backscatter" from DOS floods
  - Attackers probing blindly
  - Random scanning from worms
- LBNL's cross-section: 1/32,768 of Internet
- UCSD, UWisc's cross-section: 1/256.

Spread of Code Red

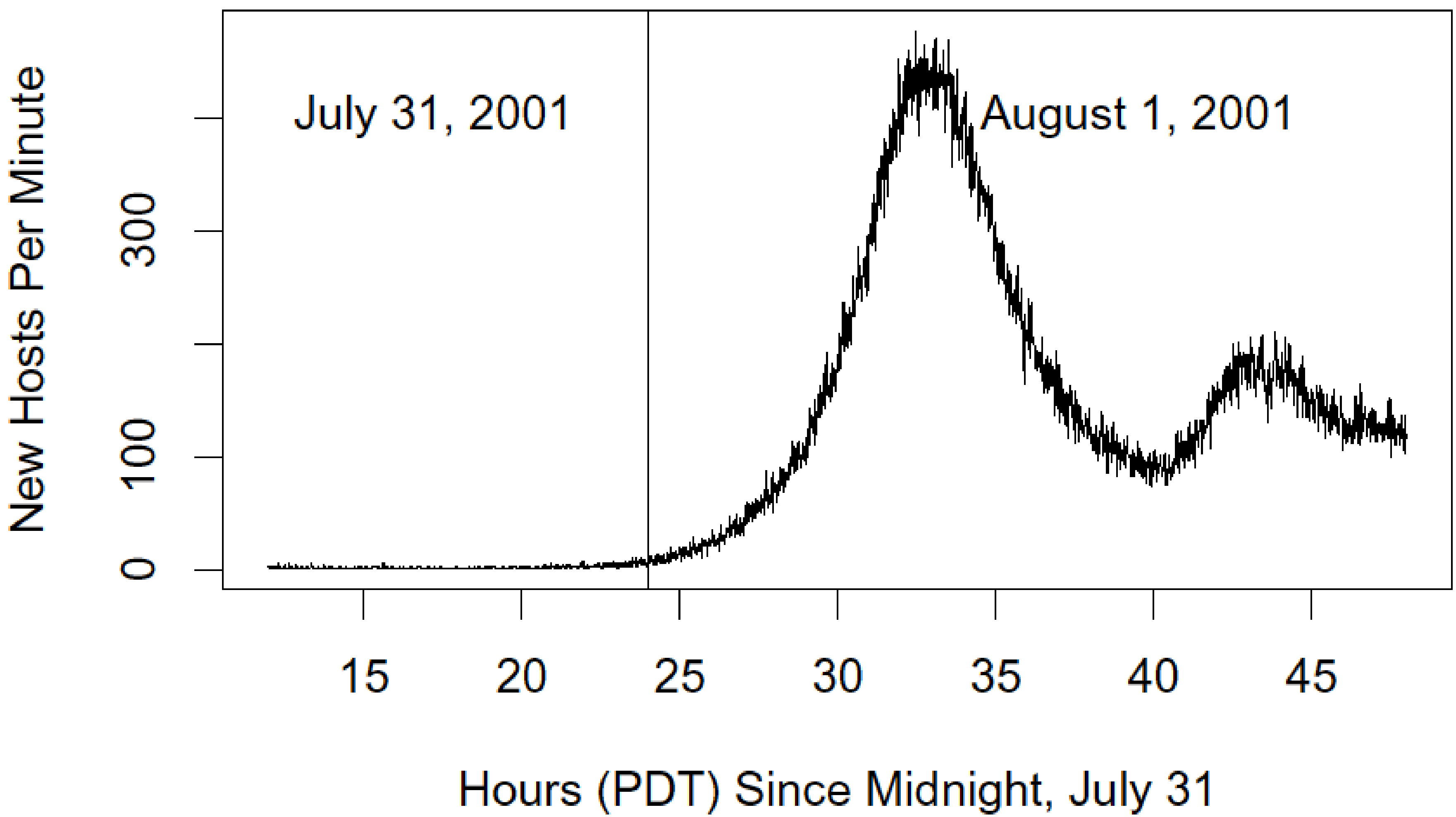
- Network telescopes estimate of # infected hosts: 360K. (Beware DHCP & NAT)
- Course of infection fits classic *logistic*.
- Note: larger the vulnerable population, *faster* the worm spreads.

- That night ($\Rightarrow$ 20th), worm dies …

  … except for hosts with inaccurate clocks!
- It just takes one of these to restart the worm on August 1st …

Slides: Vern Paxson

Return of Code Red Worm



Slides: Vern Paxson

Code Red 2

- Released August 4, 2001.
- Comment in code: “Code Red 2.”
  - But in fact completely different code base.
- Payload: a root backdoor, resilient to reboots.
- Bug: crashes NT, only works on Windows 2000.
- Kills Code Red 1.
- Safety valve: programmed to die Oct 1, 2001.

Slides: Vern Paxson

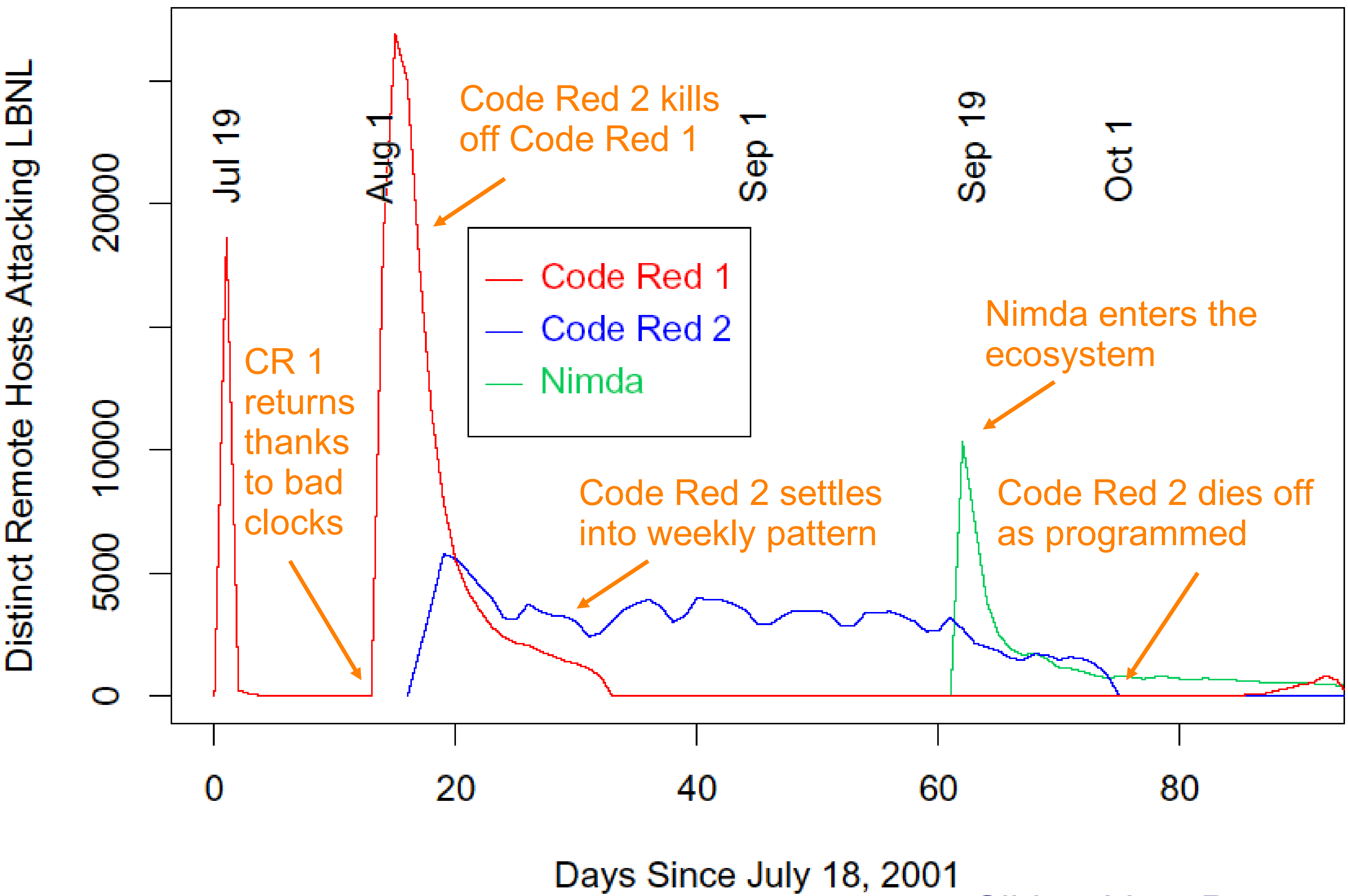
Code Red 2 (continued)

- Slept for 24 hours after infection
  - Couldn't correlate outgoing flows w. new infection
  - Then reboots machine and starts spreading
- Localized scanning: prefers nearby addresses.
  - w. prob. 1/2 try machines in same /8 network
  - w. prob. 3/8 try machines in same /16 network
  - w. prob. 1/8 try random non-class-D non-loopback
- Sets up back door w. administrative access to machine
- Not just memory resident--Resilient to reboot

Striving for Greater Virulence: Nimda

- Released September 18, 2001.
- Multi-mode spreading:
  - attack IIS servers via infected clients
  - email itself to address book as a virus
  - copy itself across open network shares
  - modifying Web pages on infected servers w/ client exploit
  - scanning for Code Red II backdoors (!)
- Worms form an ecosystem!
- Leaped across firewalls.

Slides: Vern Paxson



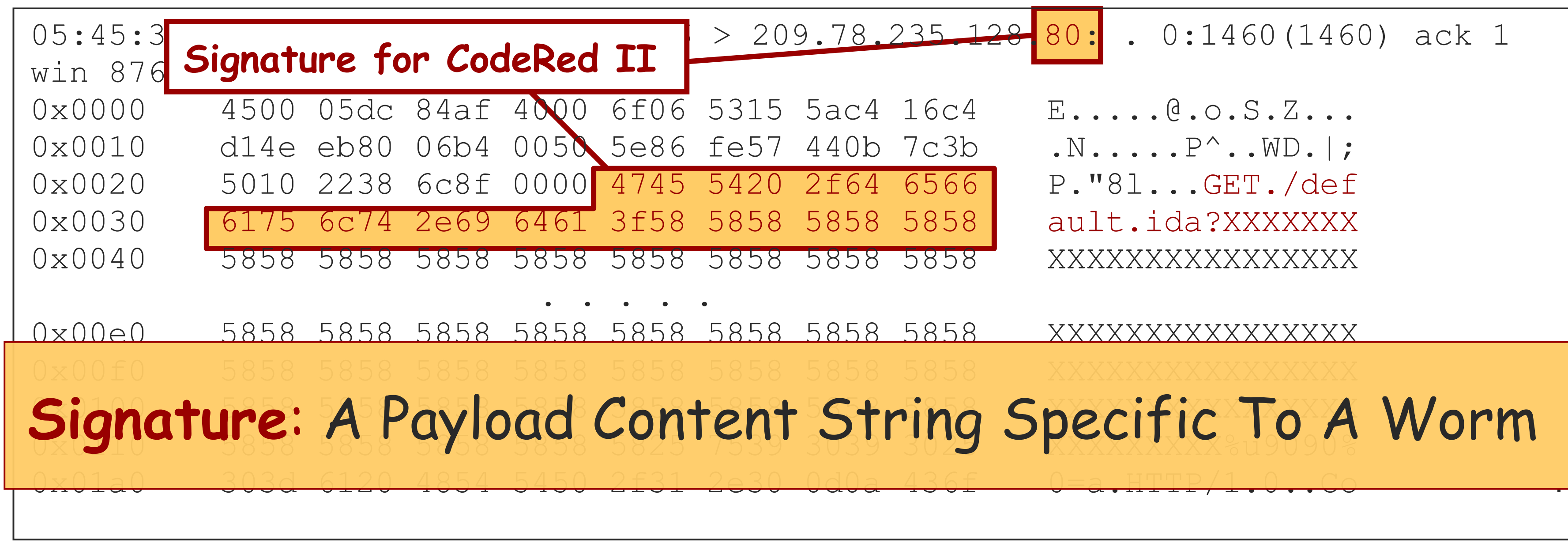
Slides: Vern Paxson

How do worms propagate?

- Scanning worms (This is currently the most common)
  - Worm chooses “random” address
- Coordinated scanning
  - Different worm instances scan different addresses
- Flash worms
  - Assemble tree of vulnerable hosts in advance, propagate along tree
    - Not observed in the wild, yet
    - Potential for 10^6 hosts in < 2 sec ! [Staniford]
- Meta-server worm
  - Ask server for hosts to infect (e.g., Google for “powered by phpbb”)
- Topological worm:
  - Use information from infected hosts (web server logs, email address books, config files, .rhosts, SSH “known hosts”)
- Contagion worm
  - Propagate parasitically along with normally initiated communication

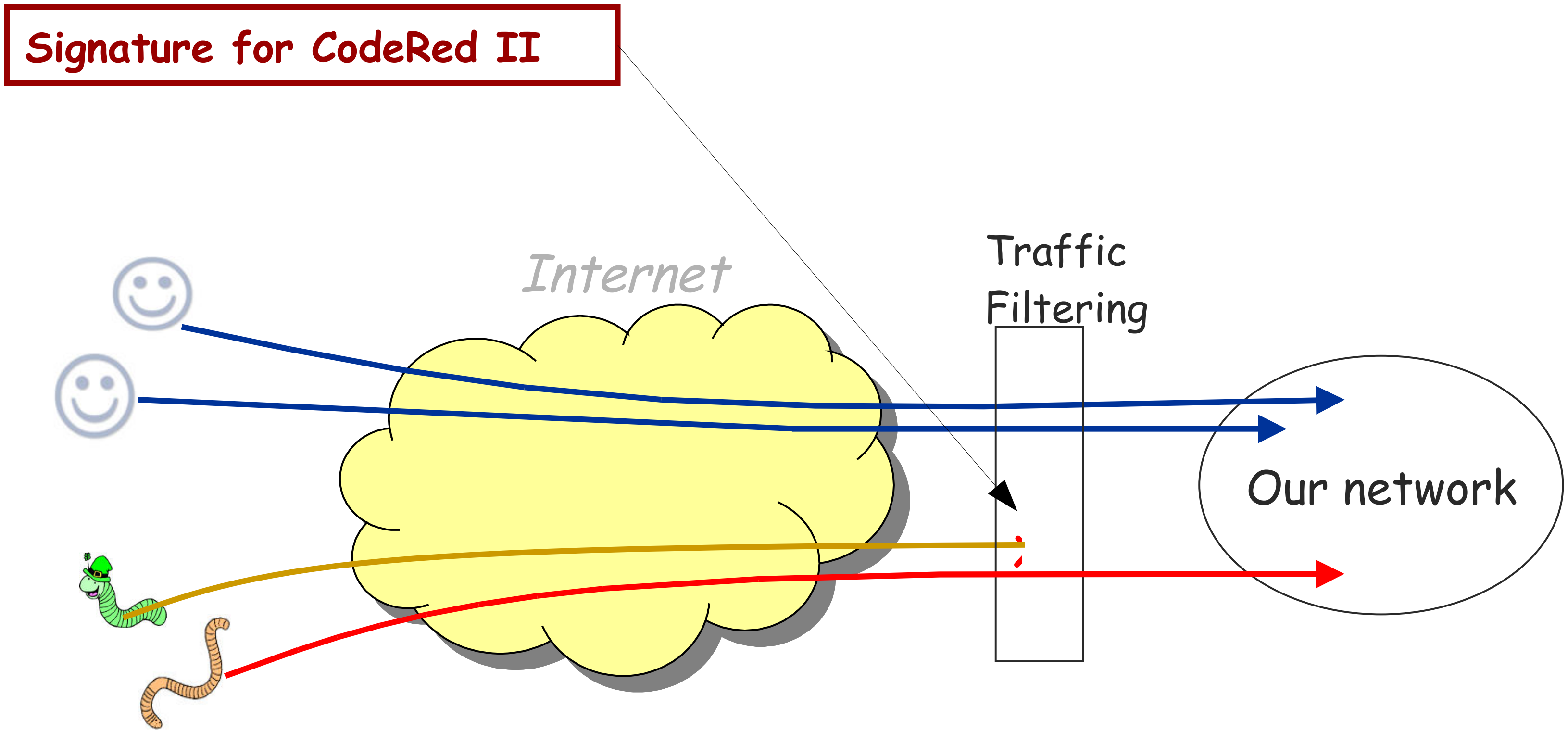
Internet Worm Quarantine

- Internet Worm Quarantine Techniques
  - Destination port blocking
  - Infected source host IP blocking
  - Content-based blocking [Moore et al.]
- Worm Signature



Next slides: Brad Karp

Content-based Blocking



Can be used by Bro, Snort, Cisco's NBAR, ...

Signature derivation is too slow

- Current Signature Derivation Process
  - New worm outbreak
  - Report of anomalies from people via phone/email/newsgroup
  - Worm trace is captured
  - Manual analysis by security experts
  - Signature generation

⇒ Labor-intensive, Human-mediated

Autograph [Kim & Karp]

Goal: Automatically generate signatures of previously unknown Internet worms

- as accurately as possible

  ⇒ Content-Based Analysis

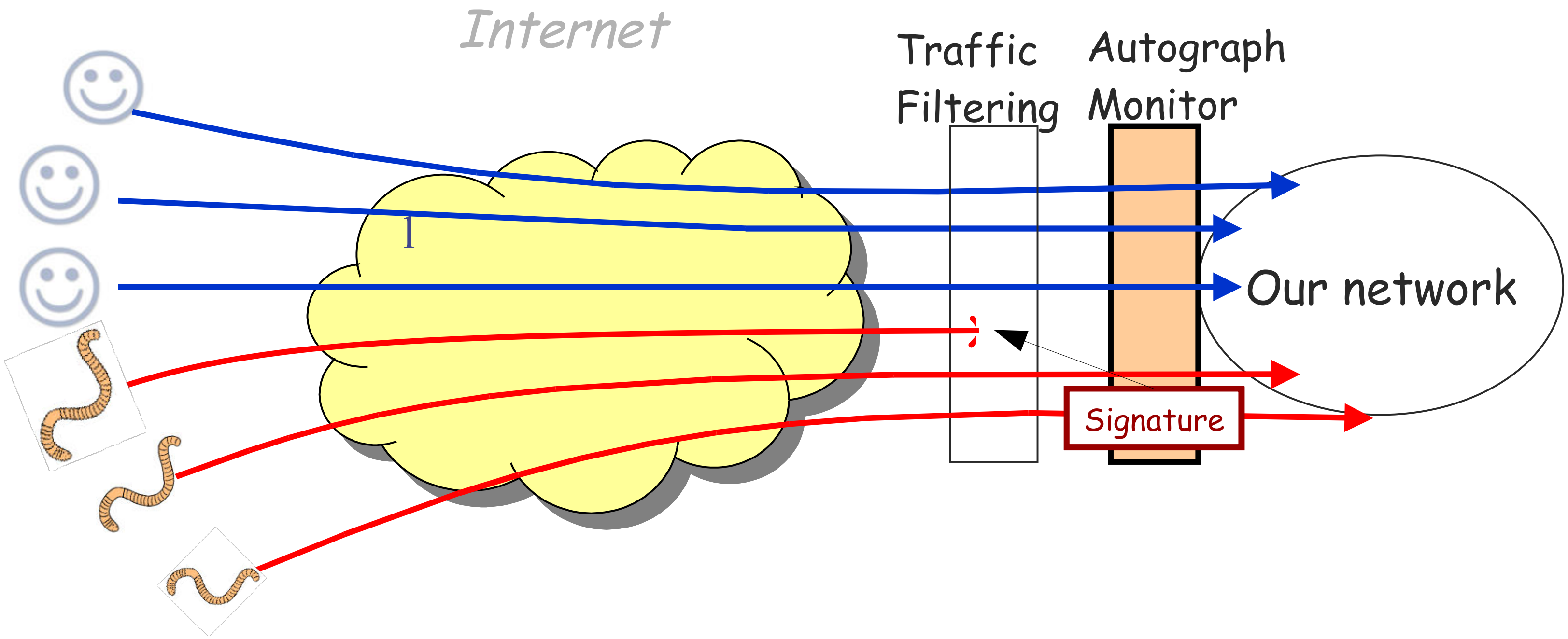
- as quickly as possible

  ⇒ Automation, Distributed Monitoring

Autograph: Assumptions

- Propagation is via scanning
- Source address can't be asily spoofed
- Can easily monitor/decode communications
- Worm’s payloads share a common substring
  - Definitely holds for non-polymorphic worms
  - May hold anyway because vulnerability exploit part is not easily mutable
  - In 2004, Singh et al. claim all common worms have had at least 400 bytes of constant payload

Automated Signature Generation



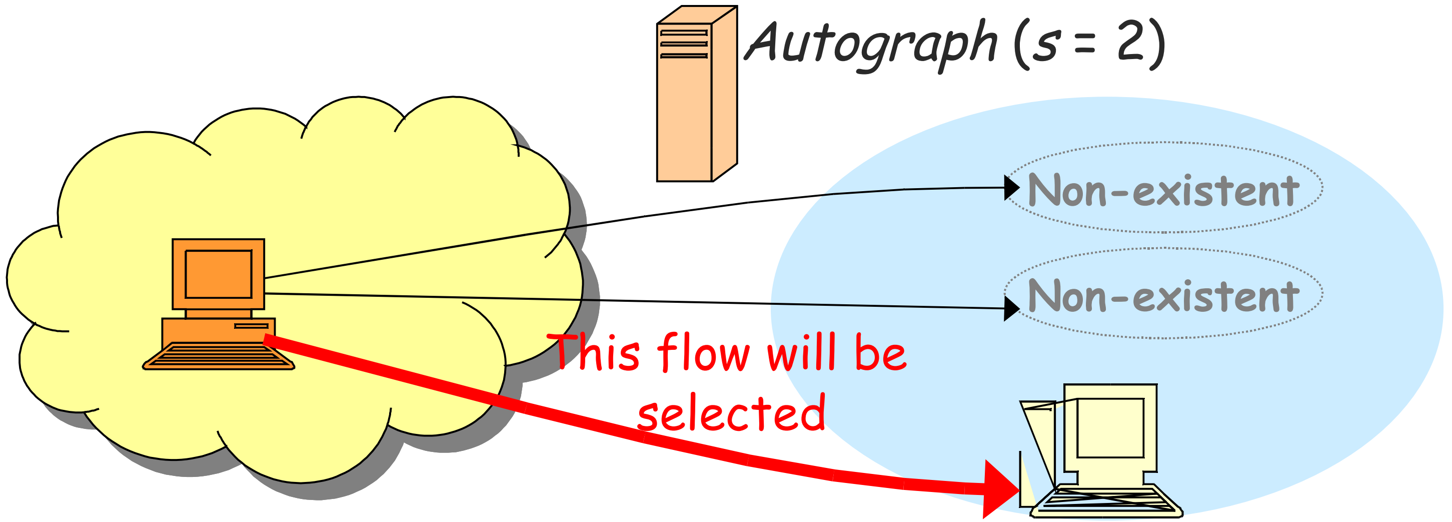
- Step 1: Select suspicious flows using heuristics
- Step 2: Generate signature using content-prevalence analysis

Suspicious Flow Selection

Reduce the work by filtering out vast amount of innocuous flows

- Heuristic: Flows from scanners are suspicious
  - Focus on the successful flows from IPs who made unsuccessful connections to more than *s* destinations for last 24 hours

⇒ Suitable heuristic for TCP worm that scans network



Suspicious Flow Selection

Reduce the work by filtering out
vast amount of innocuous flows

- Heuristic: Flows from scanners are suspicious
  - Focus on the successful flows from IPs who made unsuccessful connections to more than *s* destinations for last 24 hours
  - ⇒ Suitable heuristic for TCP worm that scans network
- Suspicious Flow Pool
  - Holds reassembled, suspicious flows seen in last *t* time
  - Triggers signature generation if there are more than θ flows
- Note suspicion heuristic far from perfect
  - Must assume classifier will have false positives & negatives

Signature Generation

Use the most frequent byte sequences across suspicious flows as signatures

All instances of a worm have a common byte pattern specific to the worm

Rationale

- Worms propagate by duplicating themselves
- Worms propagate using vulnerability of a service

How to find the most frequent byte sequences?

Worm-specific Pattern Detection

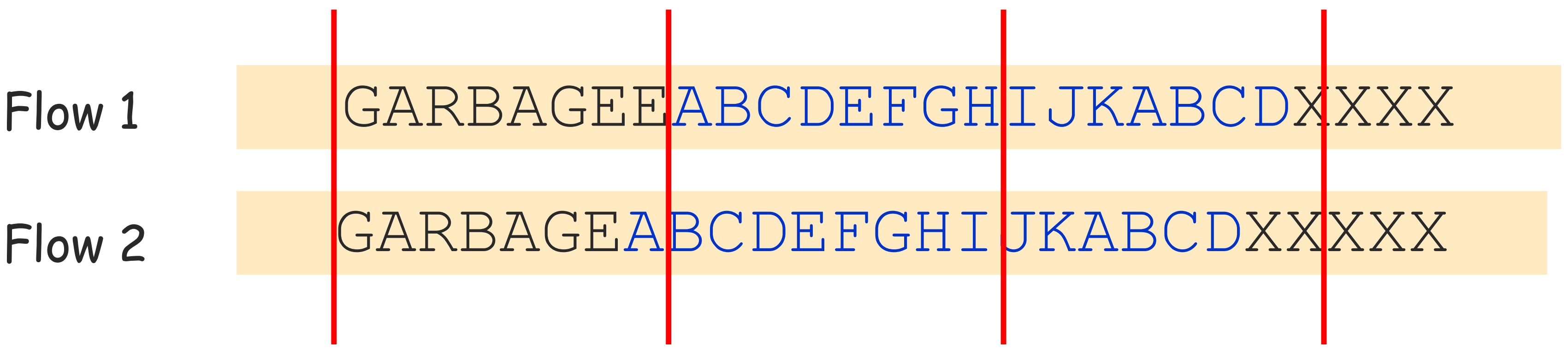
- Use the entire payload
  - Brittle to byte insertion, deletion, reordering

Flow 1	GARBAGEEABCDEFGHIJKABCDXXXX
Flow 2	GARBAGEABCDEFGHIJKABCDXXXXX

Worm-specific Pattern Detection

Partition flows into non-overlapping small blocks and count the number of occurrences

- Fixed-length Partition
  - Still brittle to byte insertion, deletion, reordering

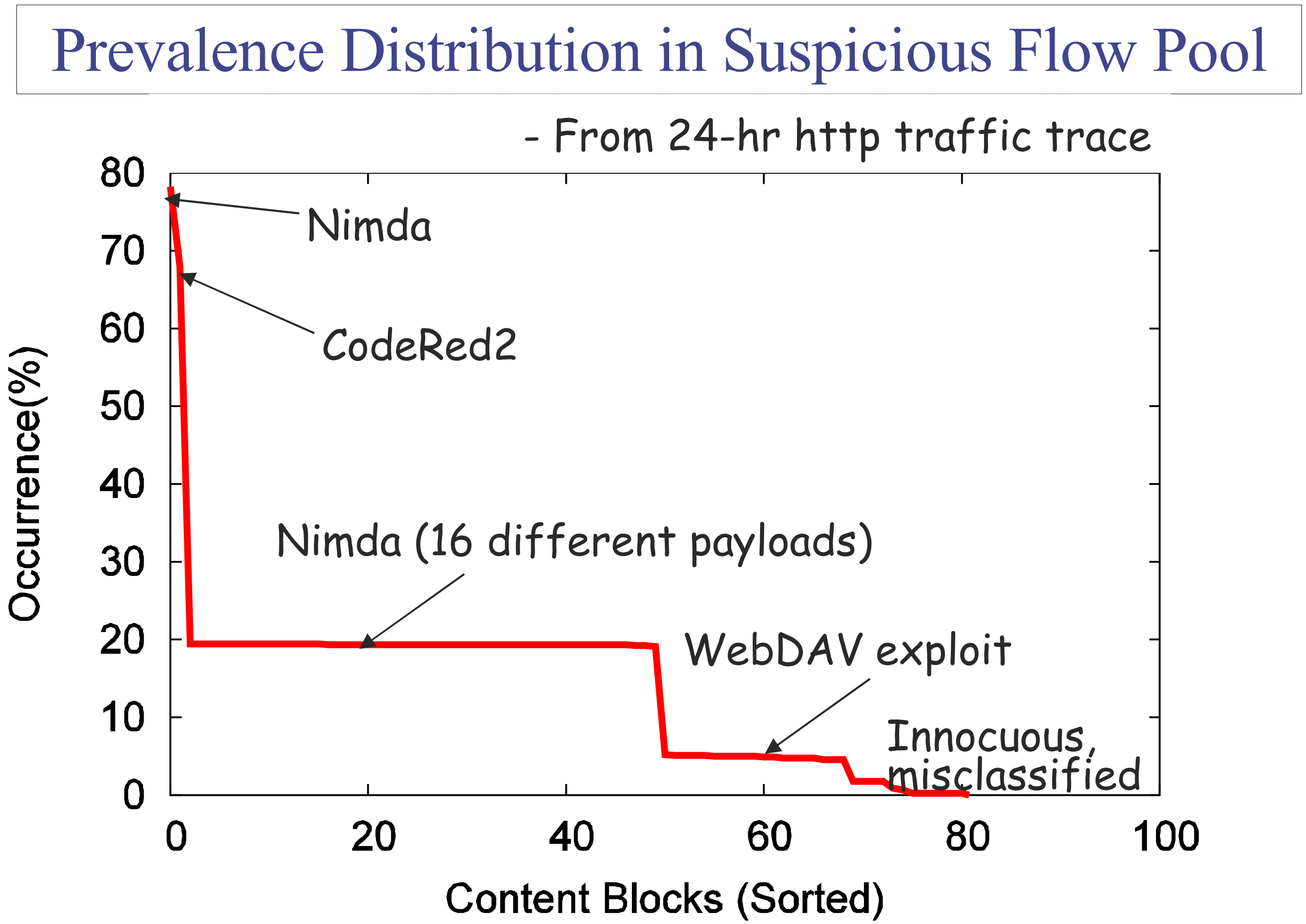


Worm-specific Pattern Detection

- Content-based Payload Partitioning (COPP)
  - Partition if Rabin fingerprint of a sliding window matches Breakmark
  - Configurable parameters: content block size (minimum, average, maximum), breakmark, sliding window

Breakmark = last 8 bits of fingerprint (ABCD)

Why Prevalence?

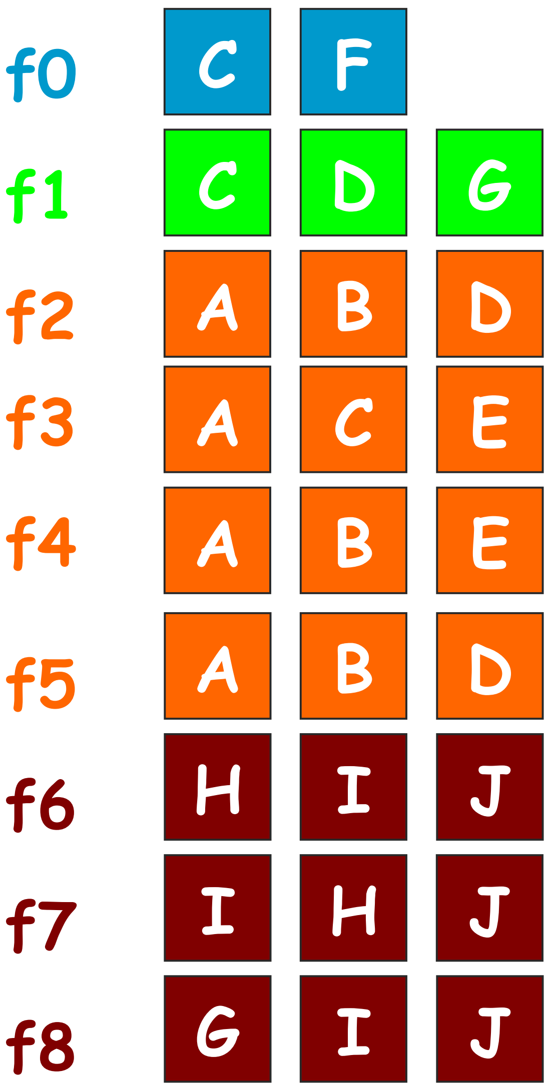


- Worm flows dominate in the suspicious flow pool
- Content-blocks from worms are highly ranked

Select Most Frequent Content Block

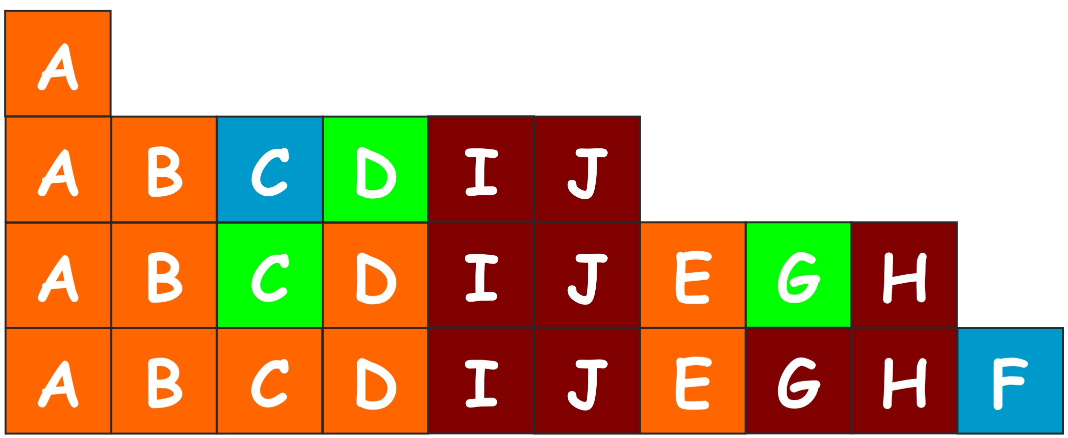
f0	C	F	
f1	C	D	G
f2	A	B	D
f3	A	C	E
f4	A	B	E
f5	A	B	D
f6	H	I	J
f7	I	H	J
f8	G	I	J

Select Most Frequent Content Block



f0	C F
f1	C D G
f2	A B D
f3	A C E
f4	A B E
f5	A B D
f6	H I J
f7	I H J
f8	G I J

Select Most Frequent Content Block



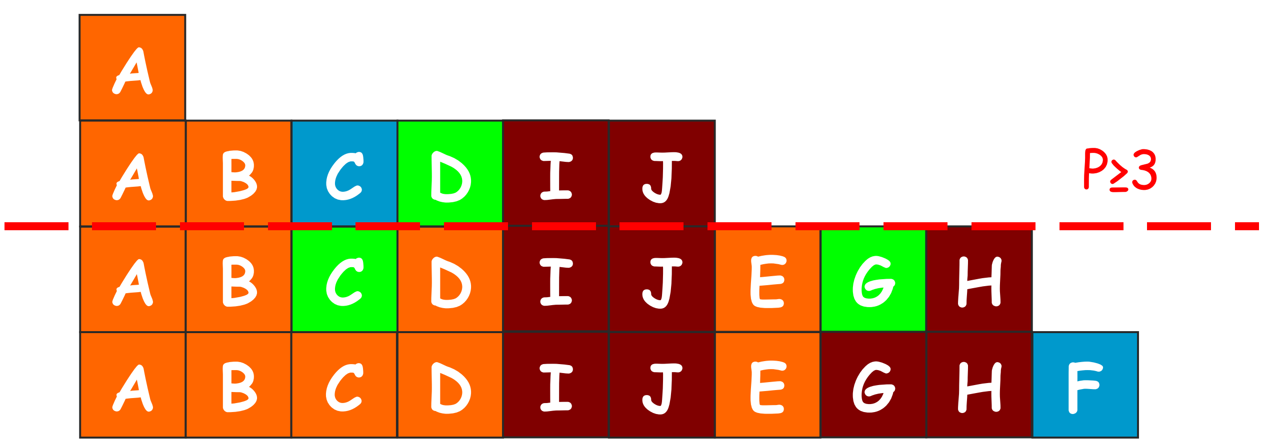
f0	C F
f1	C D G
f2	A B D
f3	A C E
f4	A B E
f5	A B D
f6	H I J
f7	I H J
f8	G I J

Select Most Frequent Content Block

Signature:

W: target coverage in suspicious flow pool

P: minimum occurrence to be selected



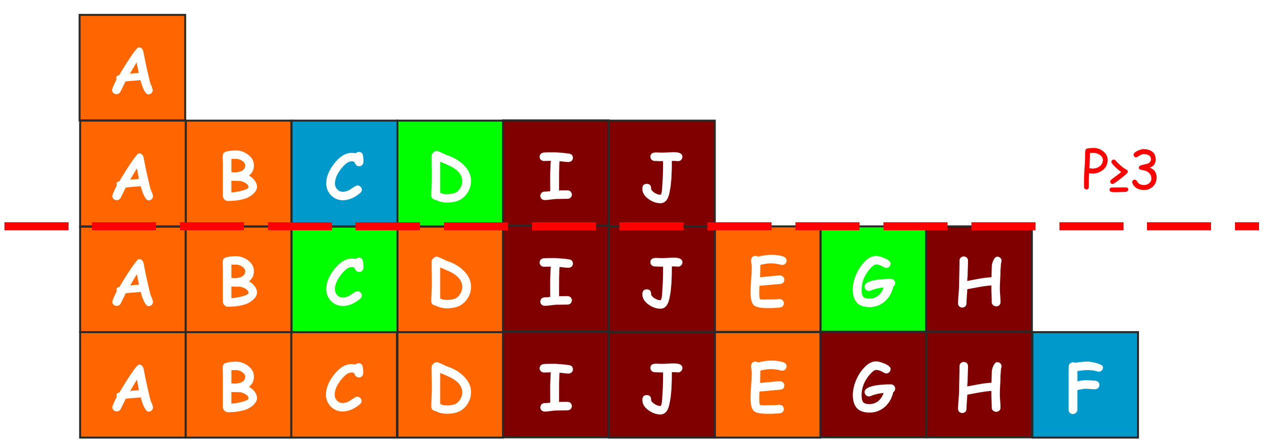
f0	C F
f1	C D G
f2	A B D
f3	A C E
f4	A B E
f5	A B D
f6	H I J
f7	I H J
f8	G I J

Select Most Frequent Content Block

Signature: A

W: target coverage in suspicious flow pool

P: minimum occurrence to be selected



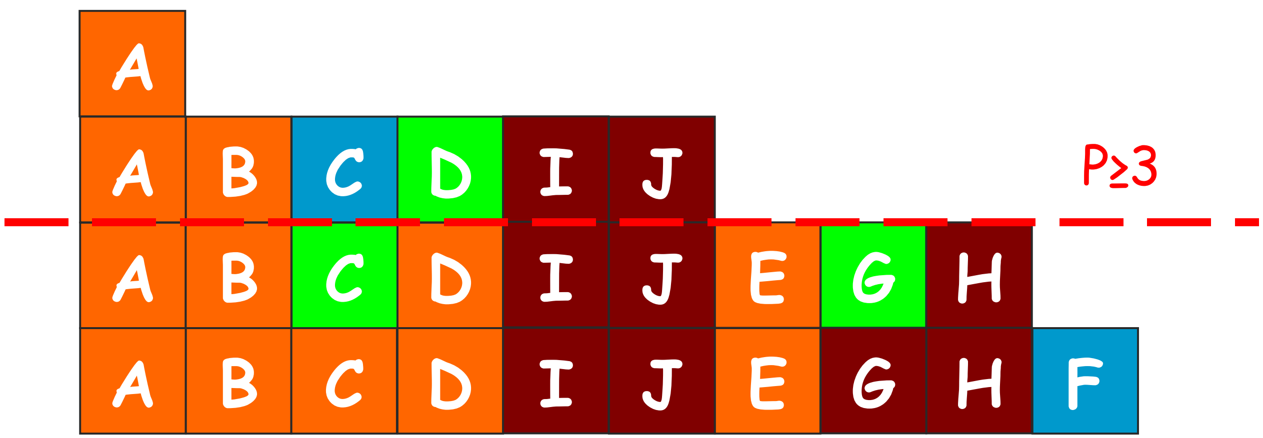
f0	C F
f1	C D G
f2	A B D
f3	A C E
f4	A B E
f5	A B D
f6	H I J
f7	I H J
f8	G I J

Select Most Frequent Content Block

Signature: A

W: target coverage in suspicious flow pool

P: minimum occurrence to be selected



f0	C F
f1	C D G
~~f2~~	~~A B D~~
~~f3~~	~~A C E~~
~~f4~~	~~A B E~~
~~f5~~	~~A B D~~
f6	H I J
f7	I H J
f8	G I J

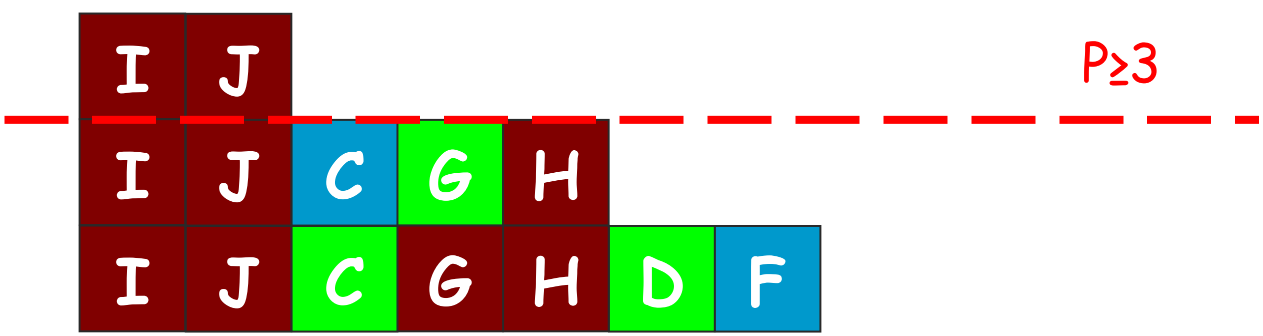
Select Most Frequent Content Block

Signature: A I

f0	C F
f1	C D G
~~f2~~	~~A B D~~
~~f3~~	~~A C E~~
~~f4~~	~~A B E~~
~~f5~~	~~A B D~~
~~f6~~	~~H I J~~
~~f7~~	~~I H J~~
~~f8~~	~~G I J~~

W: target coverage in suspicious flow pool

P: minimum occurrence to be selected



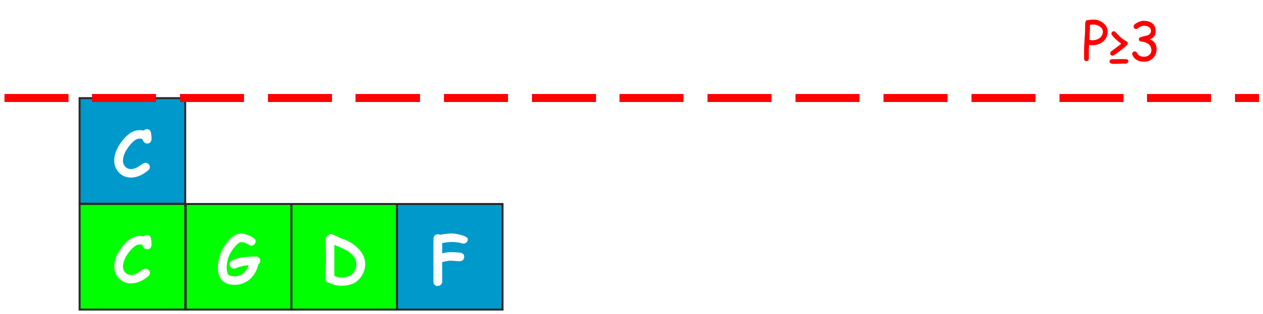
Select Most Frequent Content Block

Signature: A I

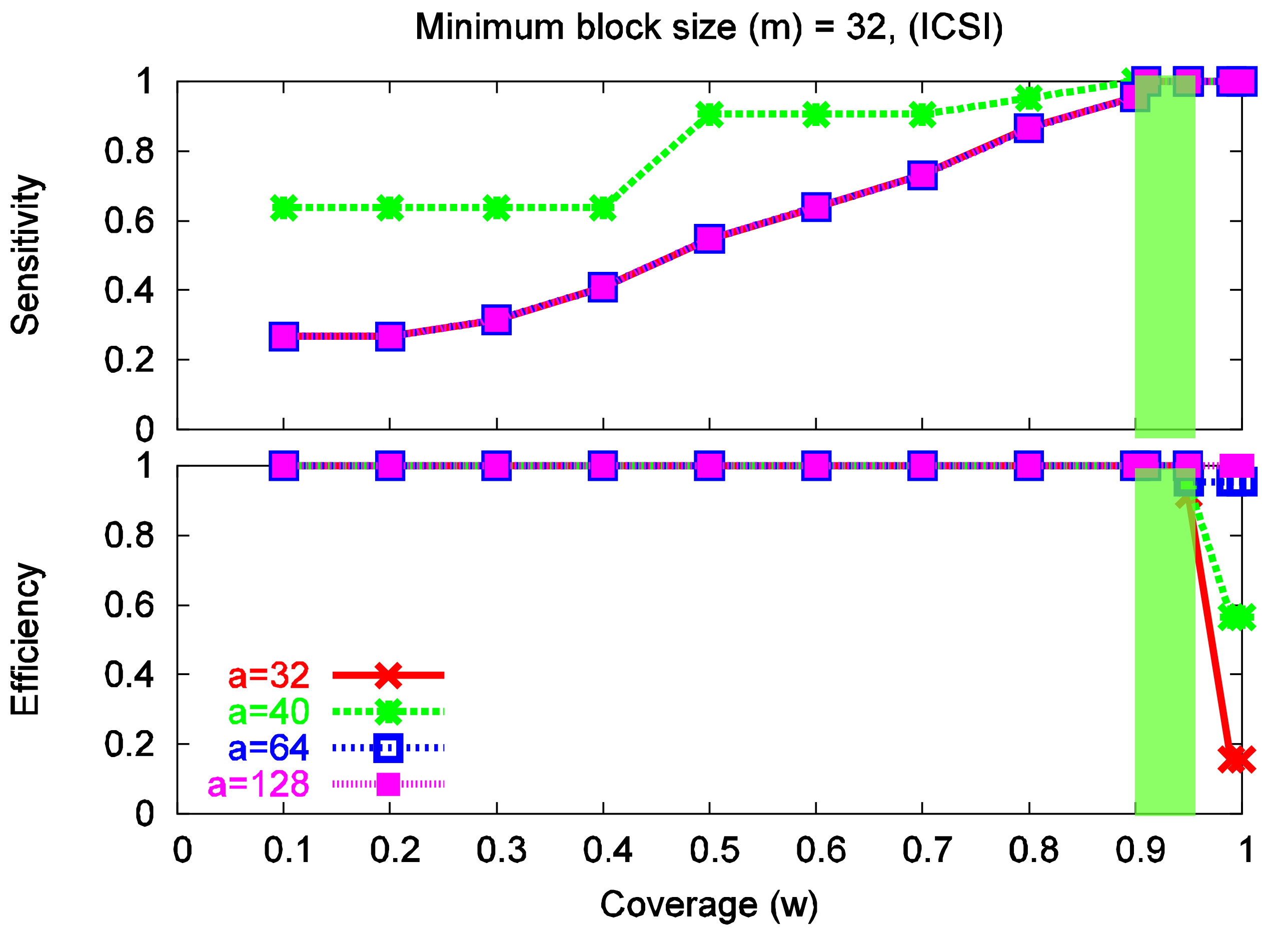
W: target coverage in suspicious flow pool

P: minimum occurrence to be selected

f0	C F
f1	C D G
~~f2~~	~~A B D~~
~~f3~~	~~A C E~~
~~f4~~	~~A B E~~
~~f5~~	~~A B D~~
~~f6~~	~~H I J~~
~~f7~~	~~I H J~~
~~f8~~	~~G I J~~



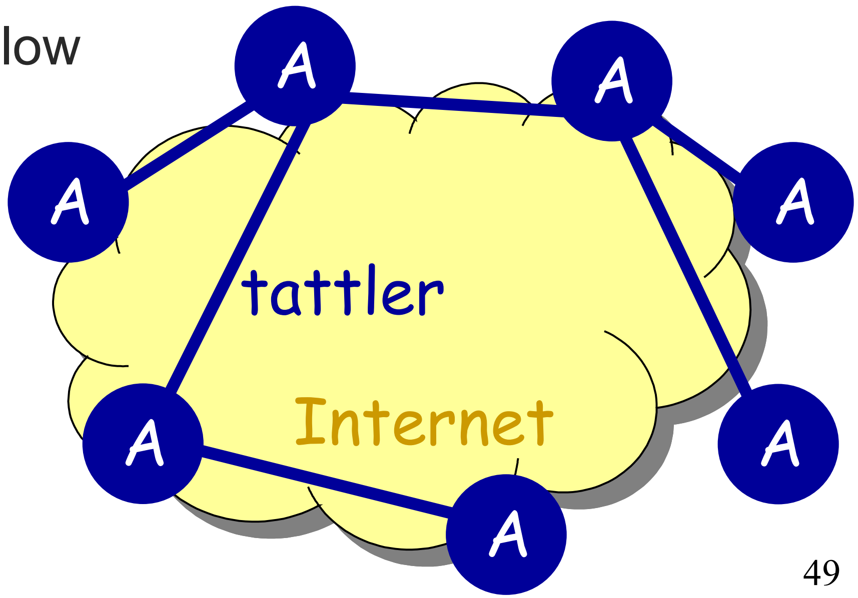
Signature Quality



- Larger block sizes generate more specific signatures
- A range of w (90-95%, workload dependent) produces a good signature

Signature Generation Speed

- Bounded by worm payload accumulation speed
  - Aggressiveness of scanner detection heuristic
    - s: # of failed connection peers to detect a scanner
  - # of payloads enough for content analysis
    - θ: suspicious flow pool size to trigger signature generation
- Single Autograph
  - Worm payload accumulation is slow
- Distributed Autograph
  - Share scanner IP list
  - Tattler: limit bandwidth consumption within a predefined cap

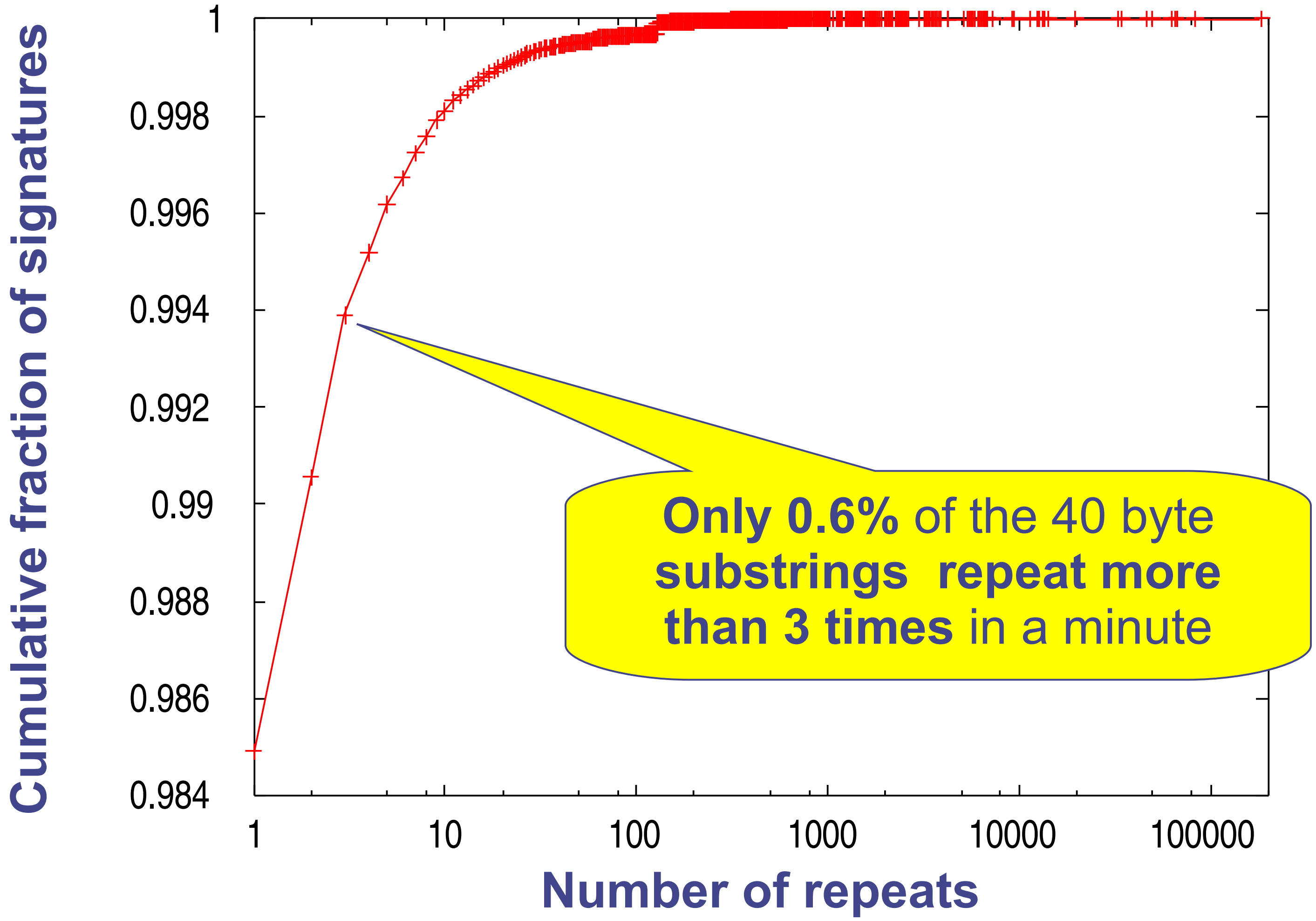


Another approach: Earlybird [Singh]

- Use overlapping fixed-size blocks (40 bytes), not COPP [next few slides]
- Inspect packets, not flows
- Assume some (relatively) unique invariant bitstring *W* across all instances of a particular worm
- Two consequences
  - **Content Prevalence**: *W* will be more common in traffic than other bitstrings of the same length
  - **Address Dispersion**: the set of packets containing *W* will address a disproportionate number of distinct sources and destinations
- *Content sifting*: find *W*'s with high content prevalence and high address dispersion and drop that traffic

Slide: S Savage

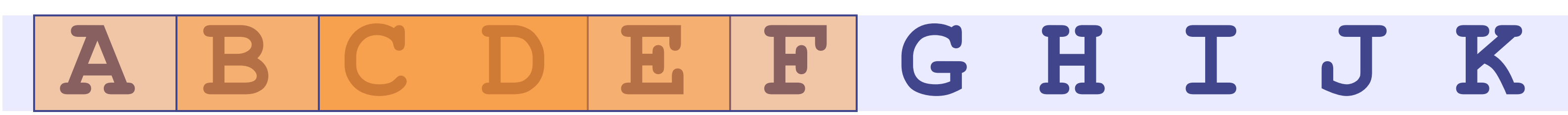
Observation:
High-prevalence strings are rare



(Stefan Savage, UCSD *)

Which substrings to index?

- **Approach 1: Index all substrings**
  - Way too many substrings → too much computation → too much state
- **Approach 2: Index whole packet**
  - Very fast but trivially evadable (e.g., Witty, Email Viruses)
- **Approach 3: Index all contiguous substrings of a fixed length ‘S’**
  - Can capture all signatures of length ‘S’ and larger



(Stefan Savage, UCSD *)

How to subsample?

- **Approach 1: sample packets**
  - If we chose 1 in N, detection will be slowed by N
- **Approach 2: sample at particular byte offsets**
  - Susceptible to simple evasion attacks
  - No guarantee that we will sample same sub-string in every packet
- **Approach 3: sample based on the hash of the substring**
  - Like COPP, but chose strings to remember, not partition points this way

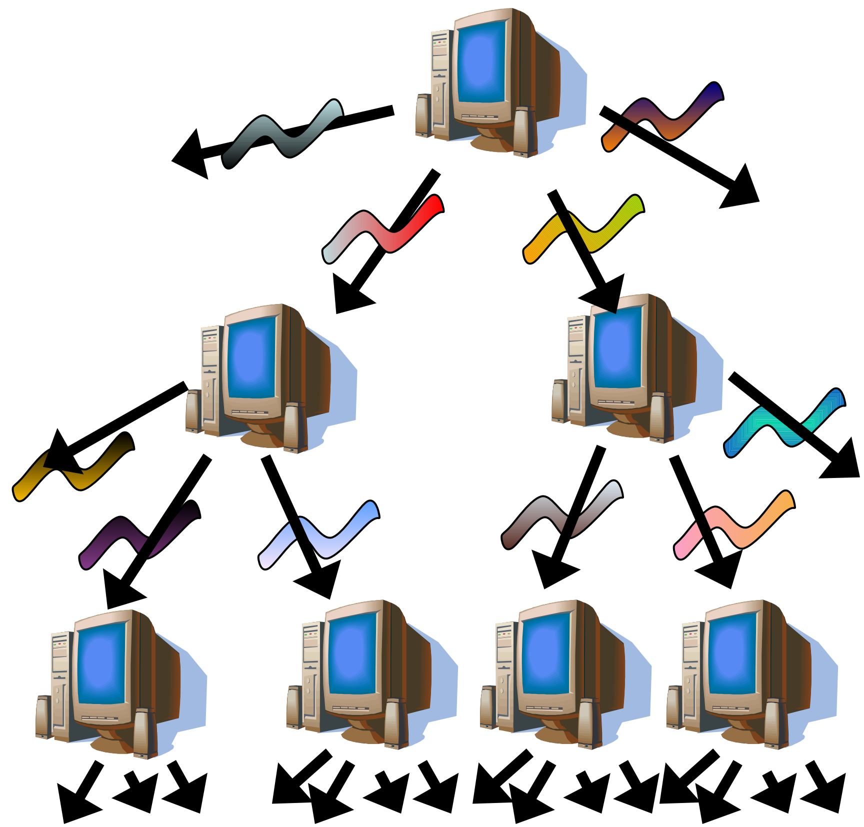
(Stefan Savage, UCSD *)

Earlybird contributions

- Fast ways to track blocks with minimal state
- Multistate filters
  - Hash blocks into multiple tables of counters
  - Increment low counter
  - Consider block high-prevelance if all counters high
- Scalable bitmap counters for detecting dispersion
  - 5x memory usage reduction, modest error

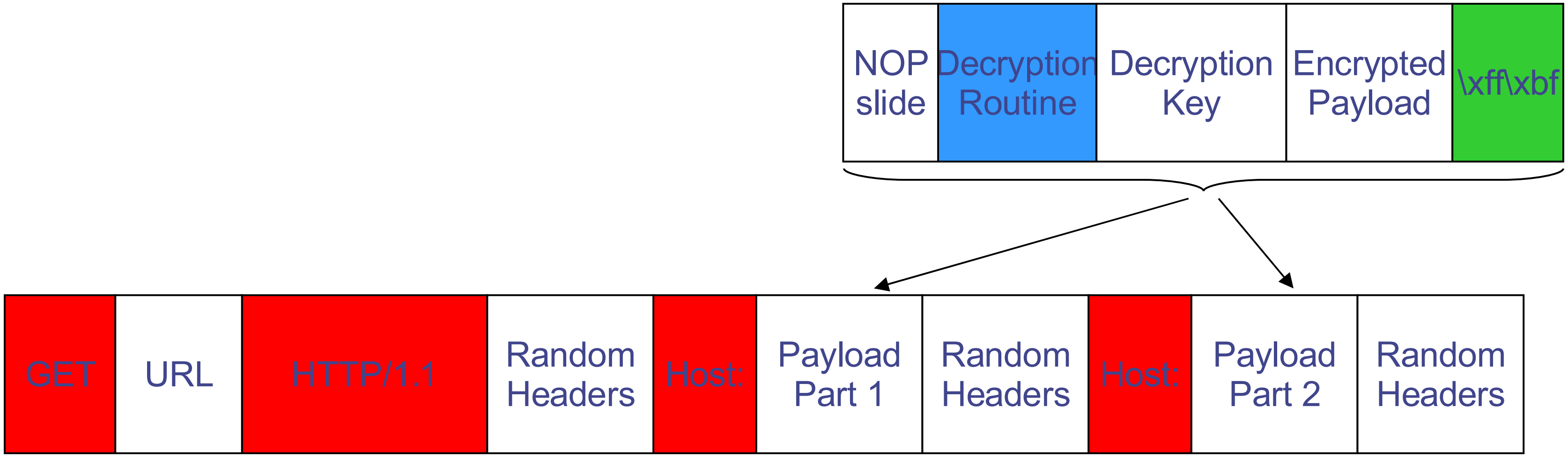
What about polymorphic worms?

- Polymorphic worms minimize invariant content
  - Encrypted payload
  - Obfuscated decryption routine
- Polymorphic tools already available
  - Clet, ADMmutate



Slides: Brad Karp

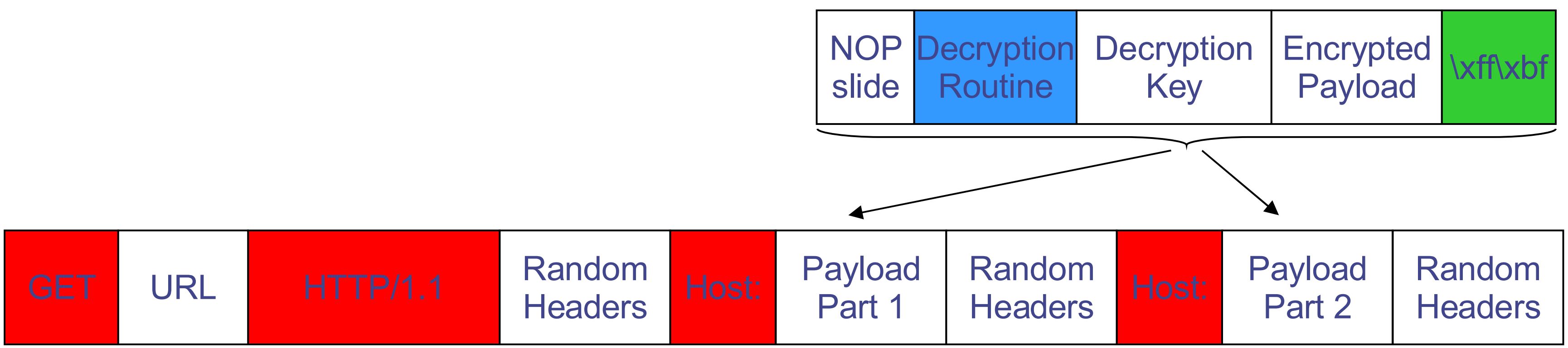
Good News: Still *some* invariant content



- Protocol framing
  - Needed to make server go down vulnerable code path
- Overwritten Return Address
  - Needed to redirect execution to worm code
- Decryption routine
  - Needed to decrypt main payload
  - BUT, code obfuscation can eliminate patterns here

Bad News: Previous Approaches Insufficient

- Previous approaches use a common substring
- Longest substring
  - "HTTP/1.1"
  - 93% false positive rate
- Most specific substring
  - "\xff\xbf"
  - .008% false positive rate (10 / 125,301)



Polygraph signatures [Newsome]

- Borrow ideas from Biology
  - Motif finding is common task when analyzing DNS
  - Can use same algorithms for worm analysis
- Types of signature:
- Conjunction: Flow matches signature if it contains all tokens in signature
  - E.g., “GET” and “HTTP/1.1” and “\r\nHost:” and “\r\nHost:” and “\xff\xbf”
- Token subsequence: match if all tokens in order
  - E.g., GET.*HTTP/1.1.*\r\nHost:.*\r\nHost:.*\xff\xbf

Limitations of previous techniques

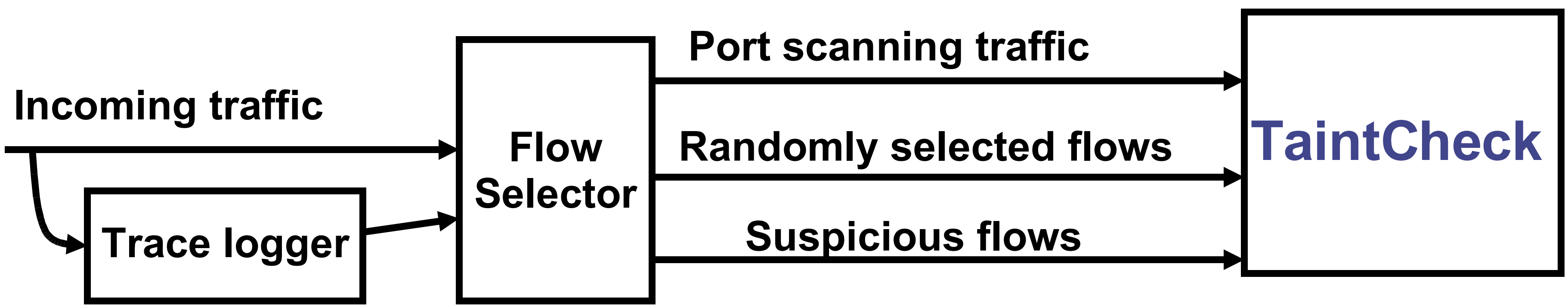
- False positives
  - E.g., Earlybird triggers on some P2P traffic
  - Requires manual whitelist generation
- False negatives
  - If you tune for low false positives, could miss ones
  - Or take so long that it is too late
- Problem would be simpler if we could classify flows without error

How to recognize malicious flows?

- Autograph, Earlybird use very crude metrics
  - Create hitlist worm to avoid port scanning
  - Earlybird 40-byte strings might have false positives
  - Attackers might intentionally poison detecter [Paragraph]
- Wouldn't it be great if we could test payloads?
  - Feed packet to application
  - Detect if it exploits a buffer overrun, etc.
- TaintCheck [Newsome]
  - Run application in environment where can detect this
  - Goal: Avoid false alarms

Fast, Low-Cost Distributed Detection

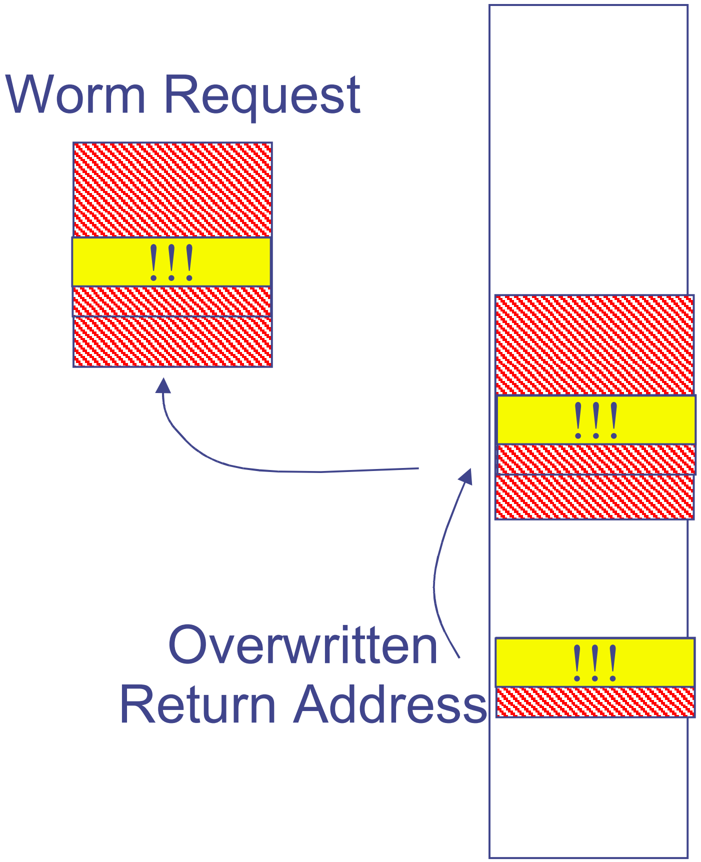
- Low load servers & Honeypots:
  - Monitor all incoming requests
  - Monitor port scanning traffic
- High load servers:
  - Randomly select requests to monitor
  - Select suspicious requests to monitor
    - When server is abnormal
      - E.g., server becomes client, server starts strange network/OS activity
    - Anomalous requests



How TaintCheck works

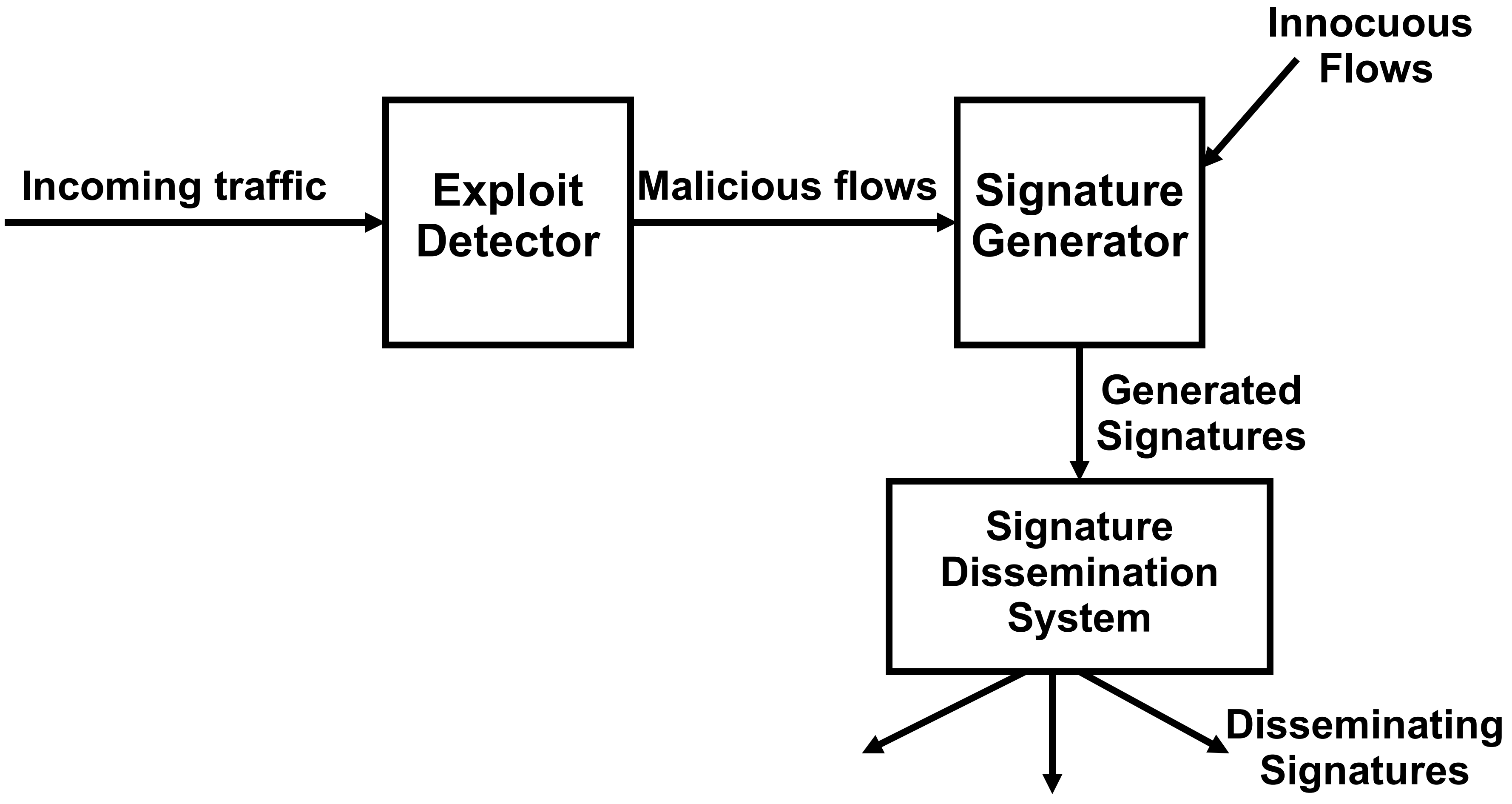
- Run application under valgrind x86 emulator
- Keep 4-byte pointer to taint struct for each byte
  - TaintSeed – mark bytes read from network
  - TaintTracker – propagate taint where data flows [no condition codes, so not completely airtight]
  - TaintAssert – check data not misused (e.g., jump target should not be data from network)
- Things that can be checked
  - Untrusted format string, buffer overflow, double free, heap smash

Semantic-based Signature Generation (I)



- Identifying invariants using semantic-based analysis
- Example invariants (I):
  - Identify overwrite value
  - Trace back to value in original request
- Experiment: ATPHttpd exploit
  - Identified overwrite return address
  - Used top 3 bytes as signature
  - Signature had 1 false positive out of 59,280 HTTP requests

Sting Architecture



Sting Evaluation

- Slammer worm attack:
  - 100,000 vulnerable hosts
  - 4000 scans per second
  - Effective contact rate r: 0.1 per second
- Sting evaluation I:
  - 10% deployment, 10% sample rate
  - Dissemination rate: 2*r = 0.2 per second
  - Fraction of protected vulnerable host: 70%
- Sting evaluation II:
  - 1% deployment, 10% sample rate
  - 10% vulnerable host protected for dissemination rate 0.2 per second
  - 98% vulnerable host protected for dissemination rate 1 per second

Generic Exploit Blocking

- Idea
  - Write signature to block all future attacks on a vulnerability
  - Different from writing a signature for a specific exploit!
- Step #1: Characterize the vulnerability "shape"
  - Identify fields, services or protocol states that must be present in attack traffic to exploit the vulnerability
  - Identify data footprint size required to exploit the vulnerability
  - Identify locality of data footprint; will it be localized or spread across the flow?
- Step #2: Write a generic signature that can detect data that "mates" with the vulnerability shape
- Similar to Shield research from Microsoft

Slide: Carey Nachenberg, Symantec

Generic Exploit Blocking Example #1

Consider MS02-039 Vulnerability (SQL Buffer Overflow):

Field/service/protocol

UDP port 1434

Packet type: 4

Minimum data footprint

Packet size > 60 bytes

Data Localization

Limited to a single packet

```
BEGIN
 DESCRIPTION: MS02-039
 NAME: MS SQL Vuln
 TRANSIT-TYPE: UDP
 TRIGGER: ANY:ANY->ANY:1434
 OFFSET: 0, PACKET
 SIG-BEGIN
   "\x04<getpacketsize(r0)>
   <inrange(r0,61,1000000)>
   <reportid()>"
 SIG-END
END
```

Slide: Carey Nachenberg, Symantec

Generic Exploit Blocking Example #2

Consider MS03-026 Vulnerability (RPC Buffer Overflow):

Field/service/protocol

RPC request on TCP/UDP 135

szName field in CoGetInstanceFromFile func.

Minimum data footprint

Arguments > 62 bytes

Data Localization

Limited to 256 bytes from start of RPC bind command

```
BEGIN
 DESCRIPTION: MS03-026
 NAME: RPC Vulnerability
 TRANSIT-TYPE: TCP, UDP
 TRIGGER: ANY:ANY->ANY:135
 SIG-BEGIN
   "\x05\x00\x0B\x03\x10\x00\x00
    (about 50 more bytes...)
    \x00\x00.*\x05\x00
    <forward(5)><getbeword(r0)>
    <inrange(r0,63,20000)>
    <reportid()>"
 SIG-END
END
```

Slide: Carey Nachenberg, Symantec