Vigilante: End-to-End Containment of Internet Worms

Paper by:
Manuel Costa, Jon Crowcroft, Miguel Castro, Ant Rowstron,
Lidong Zhou, Lintao Zhang, Paul Barham

Microsoft Research Cambridge
Microsoft Research Silicon Valley
University of Cambridge, Computer Laboratory

Presented by
Marcus Peinado, Microsoft Research

1980's to early 1990's

- Widespread adoption of personal computers
- Limited or no network connectivity
- Initially no hard drives; just floppy disks
- Single user operating systems
- Attack model: Somebody steals or tampers with my floppy disk.
- Limited attention to software security

Mid 1990's to early 2000's

- Broad internet adoption
- Massive improvements in hardware performance
- Massive increase in software complexity
- Multi-user operating systems
- New complex threats to computer security

Worms: Code Red

- Released July, August 2001
  - Infected 360,000 machines
  - Spread slowly (days)
  - Payload: (among others) DOS attack against www.whitehouse.gov

Worms: Slammer

- Released January 25, 2003
  - 75,000 vulnerable machines
  - Almost all of them infected within 10 minutes
  - No payload beyond worm propagation
  - Worm packets sent from infected machines saturated parts of the internet.
    - Exacerbated by crashes of internet routers.

Worms: Blaster

- Released: August 2003
  - 500,000 infected machines
  - Spread much more slowly than Slammer (days)
  - Author was found and sentenced to 18 months in jail.

Worms

- Each of these worms
  - Made newspaper headlines
  - Caused huge financial damages
  - Exploited vulnerabilities for which patches had been issued several months earlier
- There have been more highly-visible worms
  - But not many more

What happened next?

- Lots of work on techniques for avoiding attacks.
  - Some of them are practical.
  - Some of them are in widespread use.
    - Stack canaries, ASLR, NX, static analysis tools, pen-testing, fuzzing, software development standards
    - Developer awareness: check for buffer overflows etc.
    - User awareness: install patches asap; use AV, use firewalls
    - Response infrastructure: fast patch release, AV

- A new kind of attacker emerges
  - Interested in financial gain, rather than vandalism
  - Cyber warfare

Case study: Slammer

- Buffer overflow vulnerability in Microsoft SQL Server (MS02-039).
- Vulnerability of the following kind:

```
ProcessUDPPacket() {
      char SmallBuffer[ 100 ];

      UDPRecv( LargeBuff );
      strcpy( SmallBuf, LargeBuf );
      …
}
```

Case Study: Slammer

- Slammer is a single UDP packet
- Contains a string that overflows SmallBuffer,
  - Overwriting the return address on the stack
  - Placing the payload on the stack directly above the return address.
- Payload
  - `Repeat forever`

```
Dest_IP = random();
UDPSend( Dest_IP, SlammerPacket  );
```

Vigilante

The worm threat

- worms are a serious threat
  - worm propagation disrupts Internet traffic
  - attacker gains control of infected machines
- worms spread too fast for human response
  - Slammer scanned most of the Internet in 10 minutes
  - infected 90% of vulnerable hosts

worm containment must be automatic

Automatic worm containment

- previous solutions are **network centric**
  - analyze network traffic
  - generate signature and drop matching traffic or
  - block hosts with abnormal network behavior
- no vulnerability information at network level
  - false negatives: worm traffic appears normal
  - false positives: good traffic misclassified

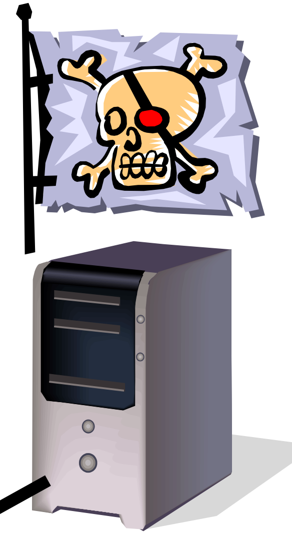
false positives are a barrier to automation

Vigilante's end-to-end architecture

- host-based detection
  - instrument software to analyze infection attempts
- cooperative detection without trust
  - detectors generate **self-certifying alerts** (SCAs)
  - detectors broadcast SCAs
- hosts generate filters to block infection

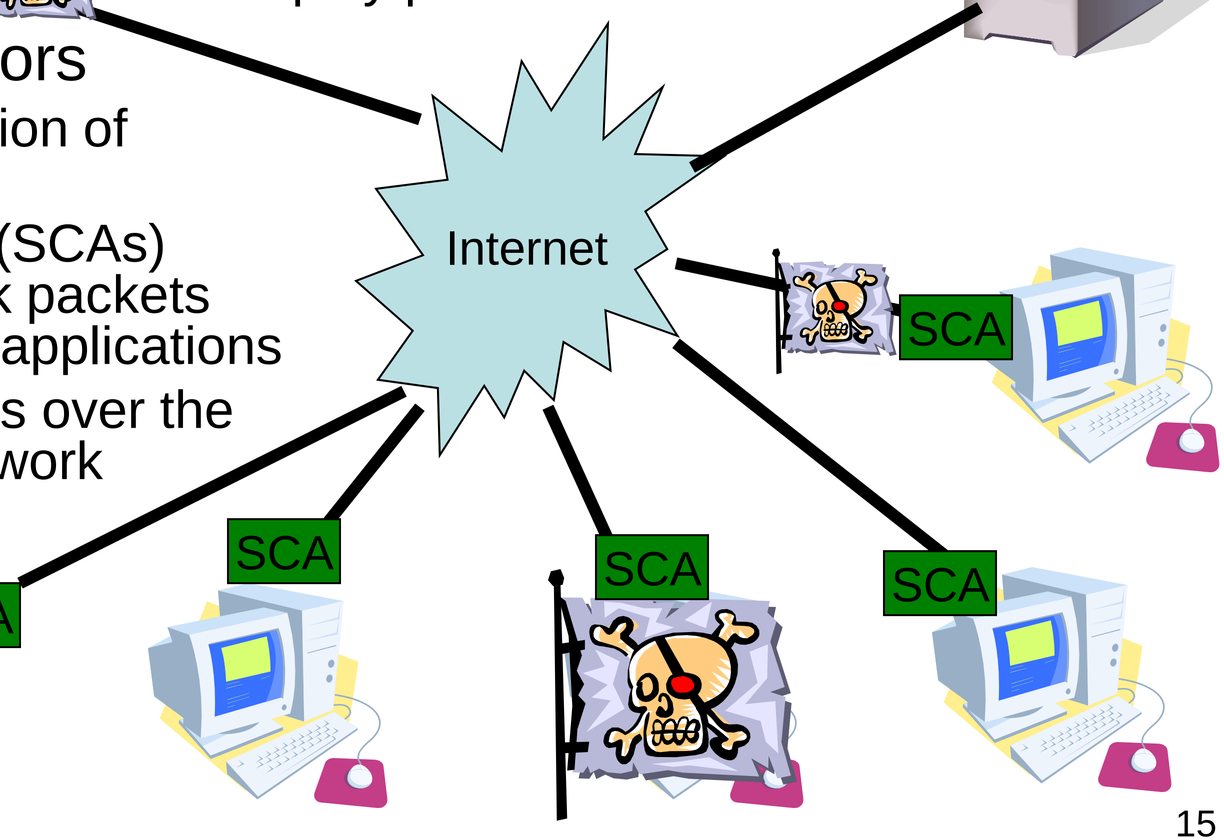
can contain fast spreading worms with small number of detectors and **without false positives**

Worm containment

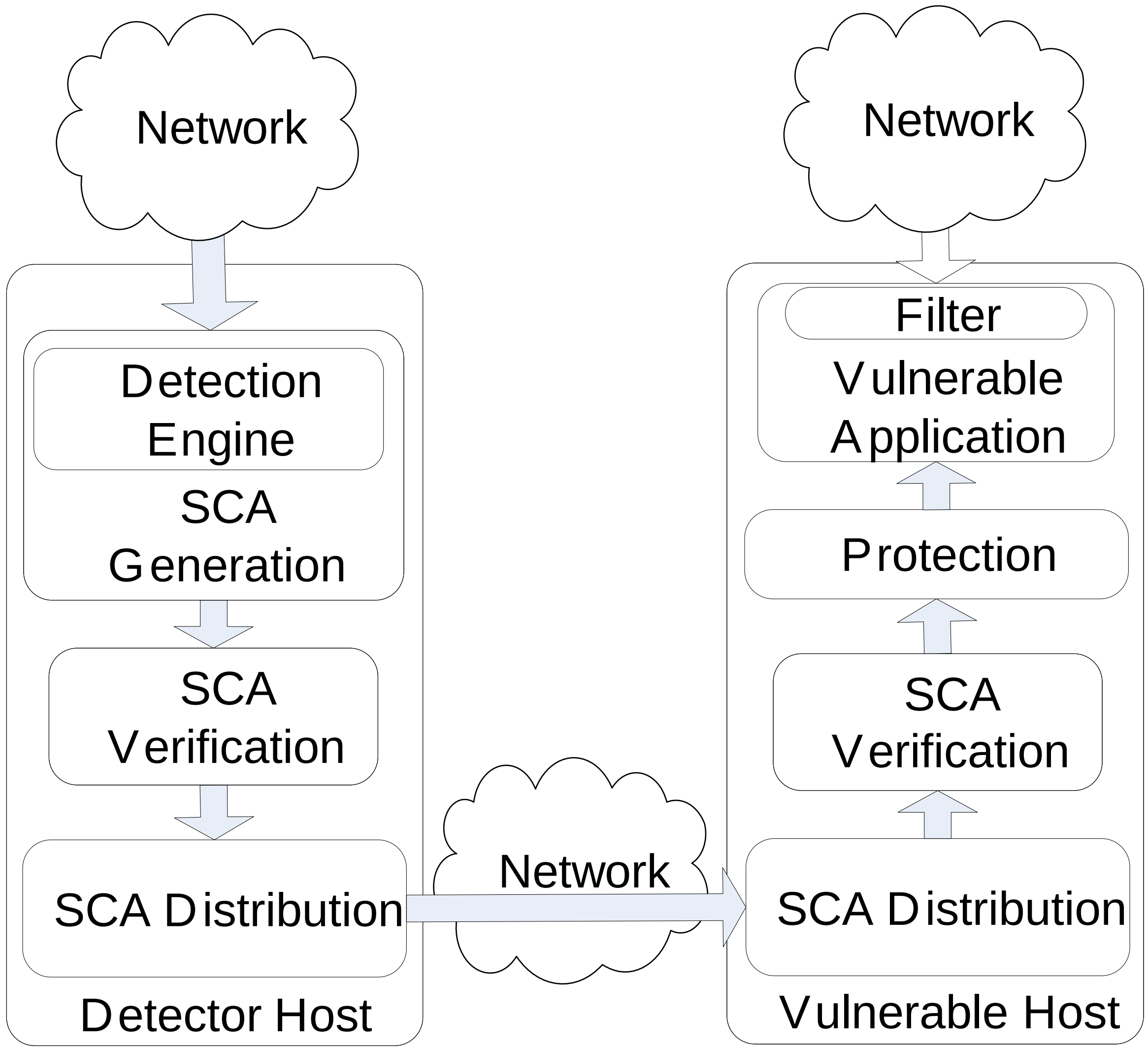


- Receive SCAs
- Verify SCAs
- Generate packet filters from SCAs
- Deploy packet filters

- Vigilante Detectors
  - Analyze execution of application
  - Produce alerts (SCAs) based on attack packets and vulnerable applications
  - Broadcast SCAs over the Pastry P2P network



Vigilante's components



Outline

- self-certifying alerts (SCAs)
- detection and generation of SCAs
- generation of vulnerability filters
- evaluation

Self-certifying alerts

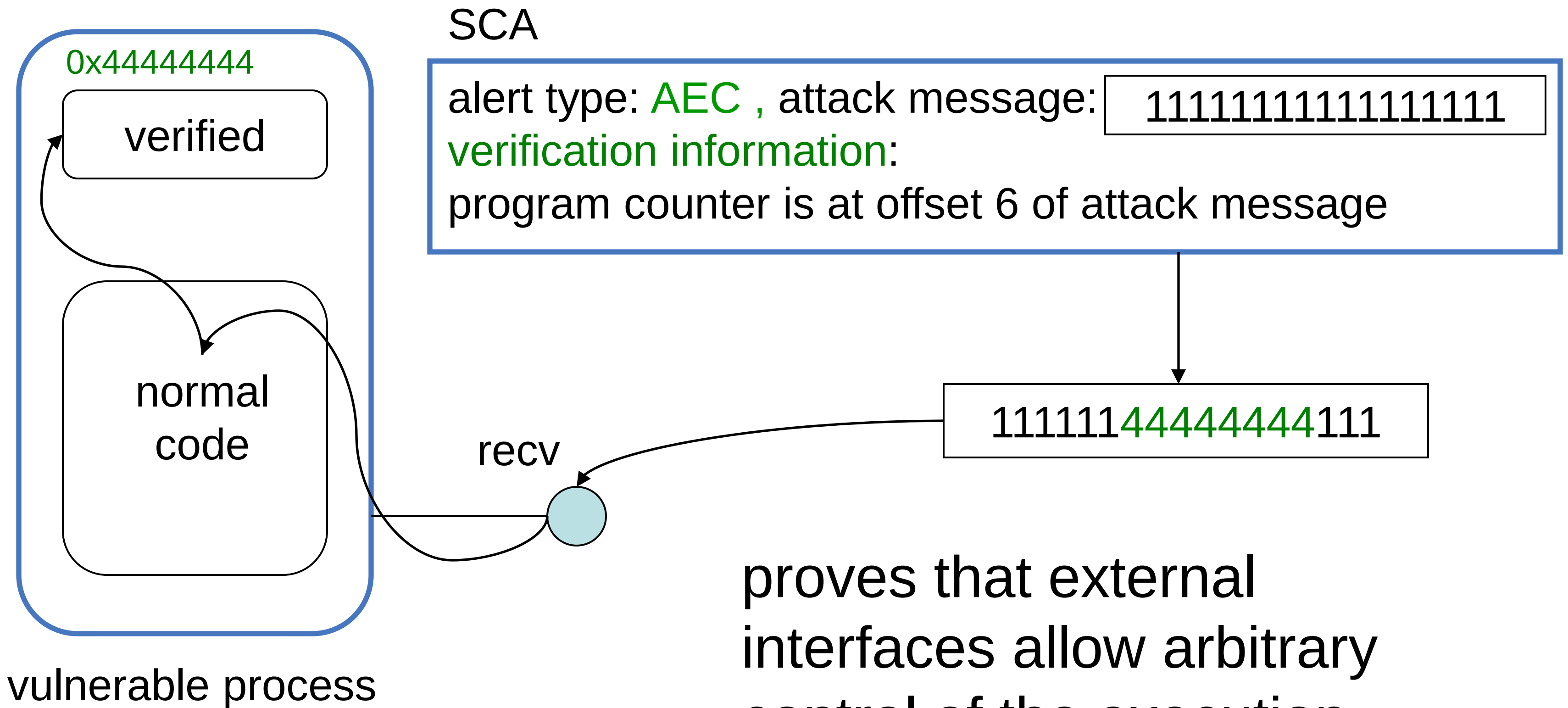
- **identify an application vulnerability**
  - describe how to exploit a vulnerability
  - contain a log of events
  - contain verification information
- **enable hosts to verify if they are vulnerable**
  - replay infection with modified events
  - verification has no **false positives**

enable cooperative worm containment without trust

SCA types

- arbitrary code execution (ACE)
  - attacker can execute code in message
  - code injection
- arbitrary execution control (AEC)
  - attacker can load a value in message into the PC
  - no code injection (e.g. return into libc)
- arbitrary function argument (AFA)
  - attacker can call function with arbitrary argument
  - data-only attacks, no abnormal control flow

Verifying an AEC alert



proves that external interfaces allow arbitrary control of the execution

verification is independent of detection mechanism

verification information enables independence

SCA generation

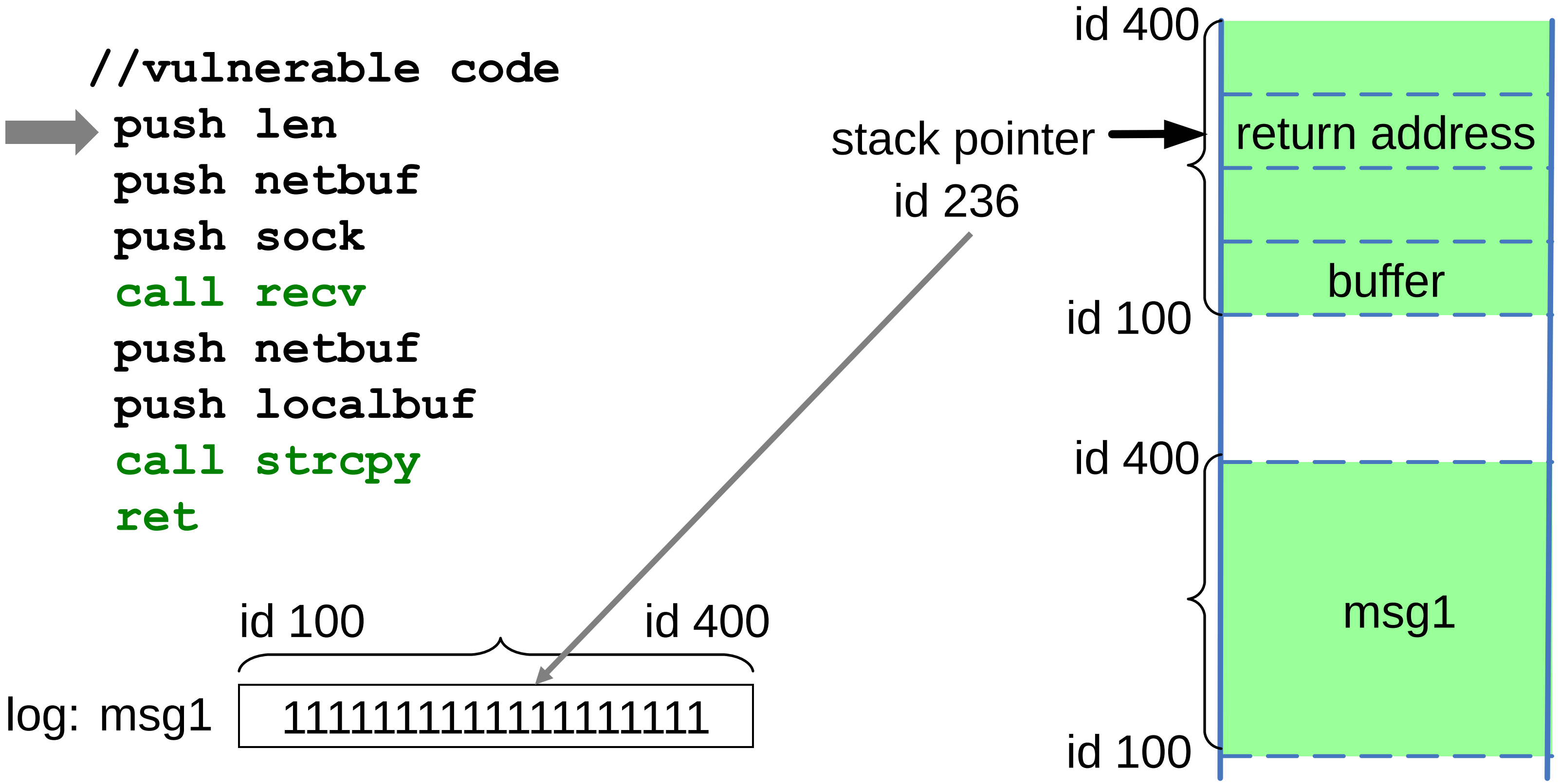
- log events
- generate SCA when worm is detected
  – compute verification information
  – search log for relevant events
  – generate tentative version of SCA
  – repeat until verification succeeds
- detectors may guide search
  – dynamic dataflow analysis is one such detector

Detection

- **dynamic dataflow analysis**
- track the flow of data from input messages
  - mark memory as dirty when data is received
  - track all data movement
- trap the worm before it executes any instructions
  - track control flow changes
  - trap execution of input data
  - trap loading of data into the program counter

Detection and SCA Generation

```
//vulnerable code
push len
push netbuf
push sock
call recv
push netbuf
push localbuf
call strcpy
ret
```



SCA: AEC, 11111111111111111111, pc at offset 136

high coverage

direct extraction of verification information

Cooperative worm containment

- SCA enables cooperative containment
  - any host can be a detector
  - hosts can run high-overhead detection engines
  - hosts can run different detection engines
  - small TCB for SCA verification

cooperation enables low false negative rate

SCA broadcast

- uses secure overlay: Pastry
  - hosts join overlay
  - detectors flood alerts over overlay links
- denial-of-service prevention
  - per-link rate limiting
  - per-hop filtering and verification
  - controlled disclosure of overlay membership

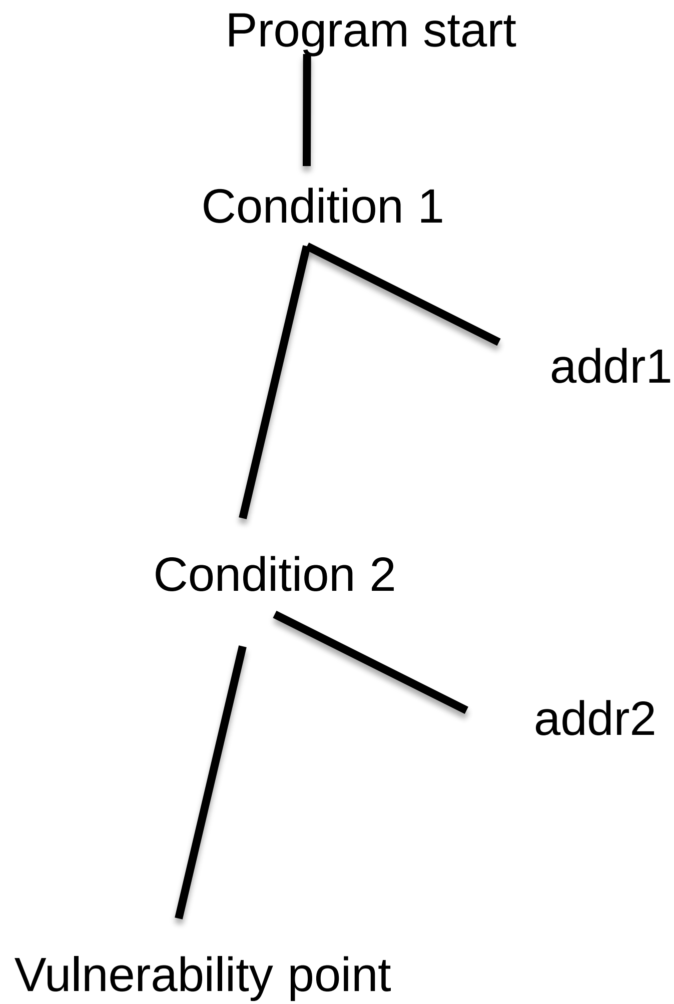
hosts receive SCAs with high probability

Protection

- hosts generate filter from SCA
- **dynamic data and control flow analysis**
  - run vulnerable application in a sandbox
  - track control and data flow from input messages
  - compute conditions that determine execution path
  - filter blocks messages that satisfy conditions

Execution trace filters

- …
- cmp eax,buf[23]
- jne addr1
- …
- …
- test ecx, buf[13]
- je addr2
- …
- …
- mov eax,buf[20]
- call eax



Generating filters for vulnerabilities

```
    //vulnerable code
    mov al,[msg]
    mov cl,0x3
    cmp al,cl
    jne L2    //msg[0] == 3 ?
    xor eax,eax
L1  mov [esp+eax+4],cl
    mov cl,[eax+msg+1]
    inc eax
    test cl,cl
    jne L1    //msg[i] == 0 ?
L2   ret
```

attack:	0x3	0x24	0x67	0x42	0x1
filter:	**=3**	**≠0**	**≠0**	**≠0**	**≠0**

Match!

mutation:	0x3	0x12	0x28	0x63	0x4

look at the program, not at the messages

find control flow decisions that enable the attack

Filters

- capture generic conditions
  - dataflow graphs of CPU instructions
- safe and efficient
  - no side effects, no loops
- accumulating all control flow decisions limits the amount of polymorphism tolerated
  - two filter design alleviates this
  - details in the paper, still improving

Properties of execution trace filters

- Central question:
  - What if the exploit mutates?
  - Will the filter still cover exploits that differ from the exploit the detector saw?
- Good:
  - Any byte in the input that does not alter the execution path of the application can be changed.
  - Immune to a large class of mutations.
- Bad:
  - Mutations that alter the execution path of the application can bypass the filter.

HTLM Exploit

- <title> … </title>
- <body>
- <IMG …>… </IMG>
- <A …> …</A>
- …

- Arbitrary sequence of HTLM tags

- Tag that exploits the vulnerability
- <script> exploit </script>

- Arbitrary sequence of HTML tags

- </body>

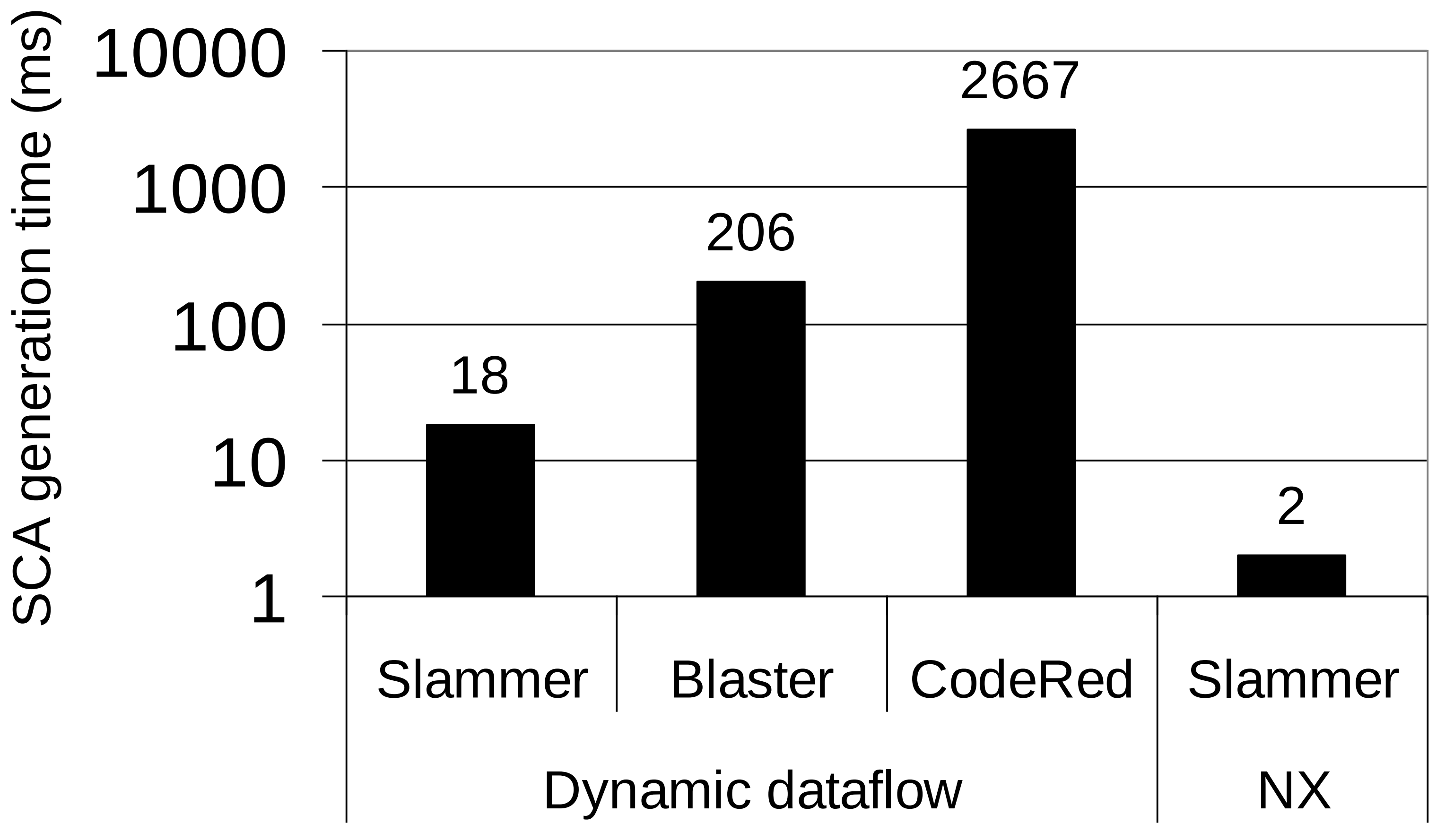
- All the irrelevant tags on the page affect the execution trace.
- Thus, the attacker can thwart execution trace filters by adding irrelevant input.

- Follow up work by the authors and others tries to address this problem.

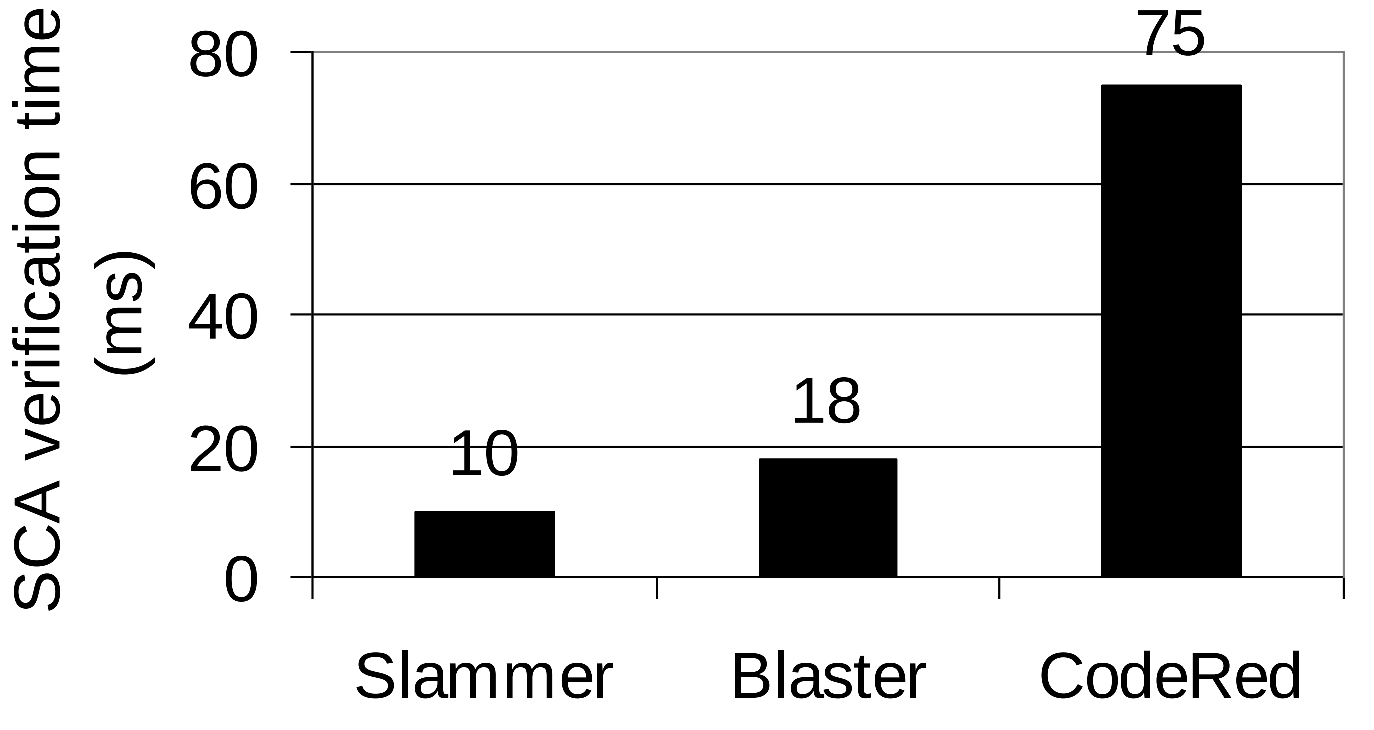
Evaluation

- three real worms:
  - Slammer (SQL server), Blaster (RPC), CodeRed (IIS)
- measurements of prototype implementation
  - SCA generation and verification
  - filter generation
  - filtering overhead
- simulations of SCA propagation with attacks

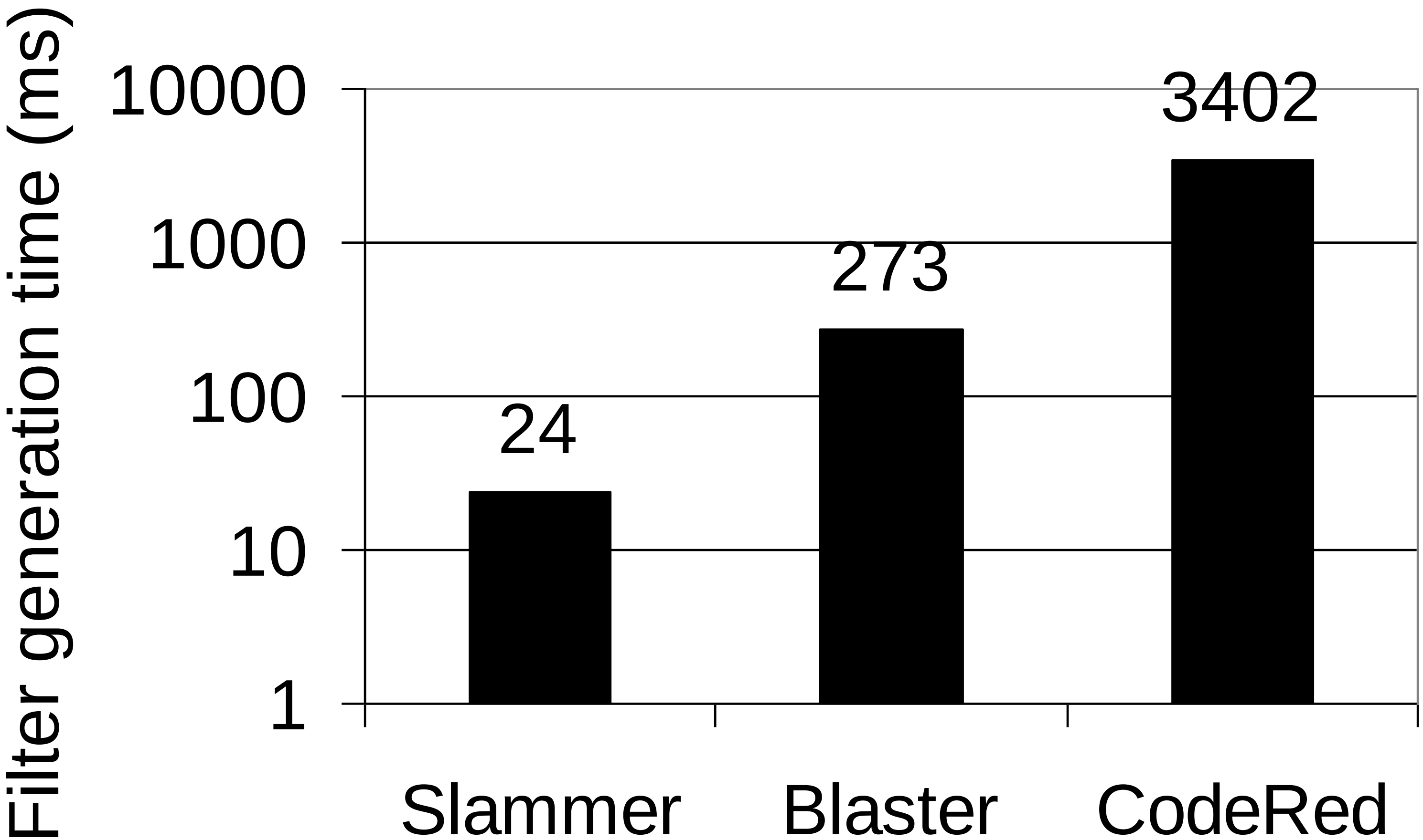
Time to generate SCAs



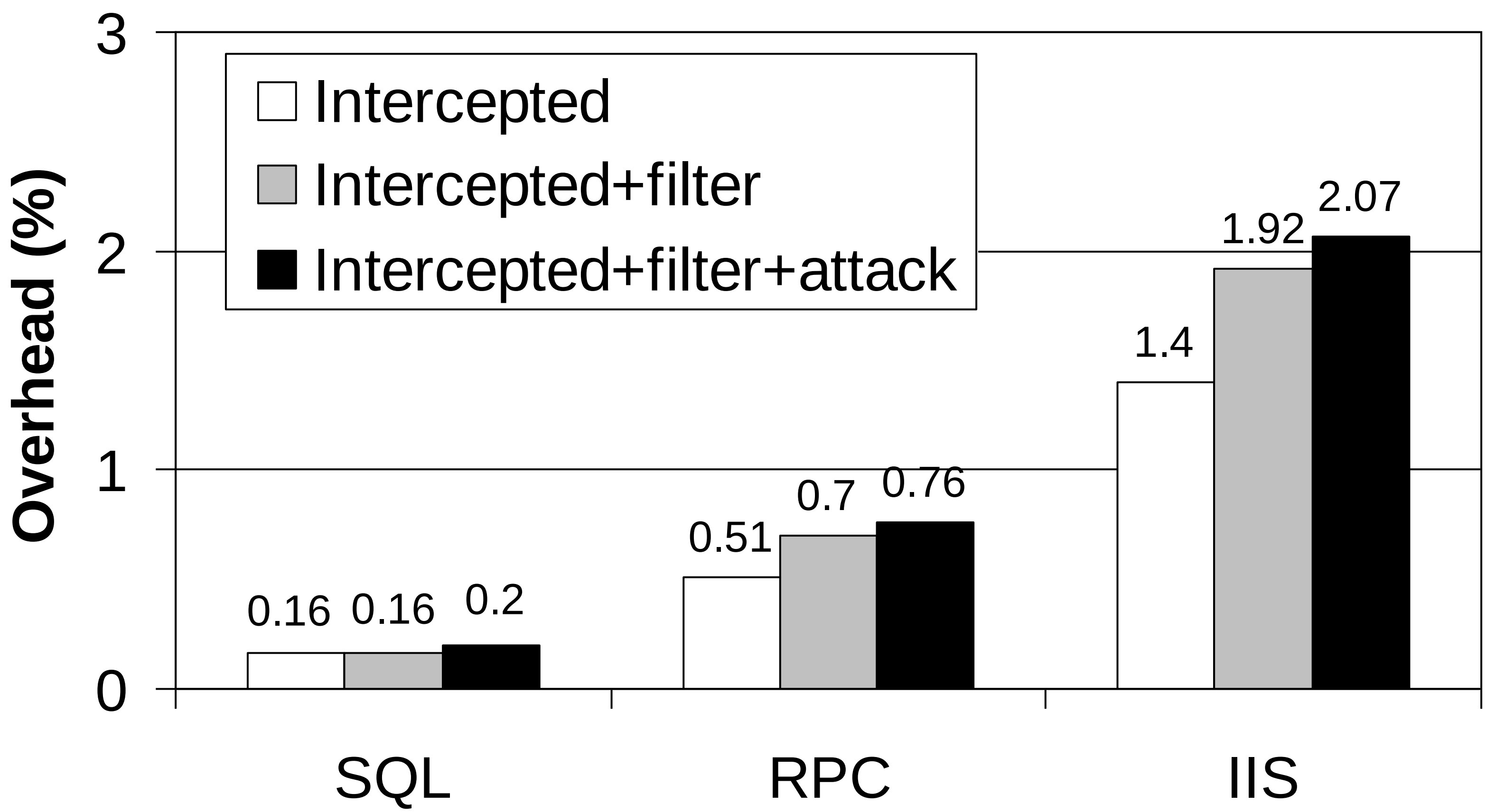
Time to verify SCAs



Time to generate filters



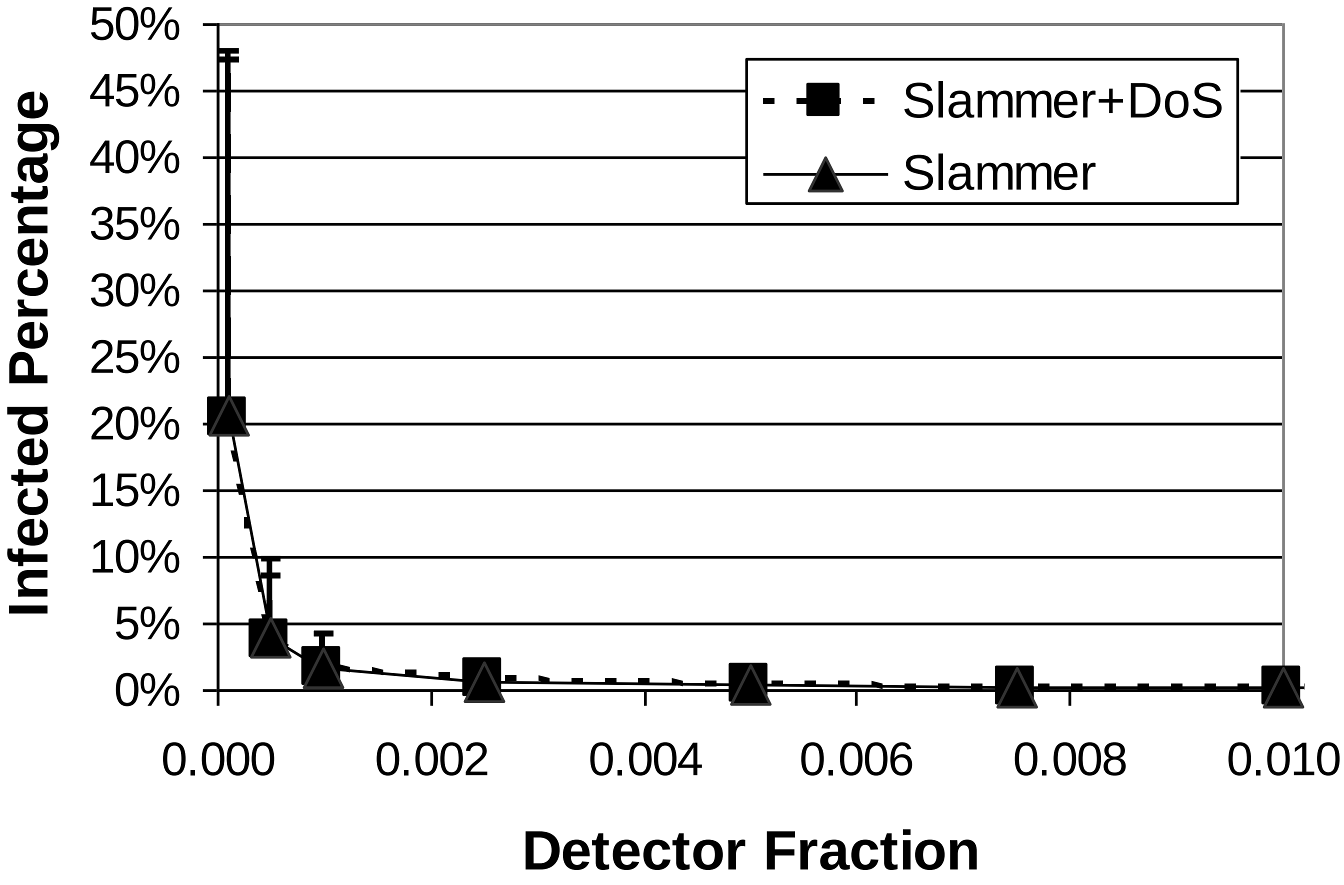
Filtering overhead



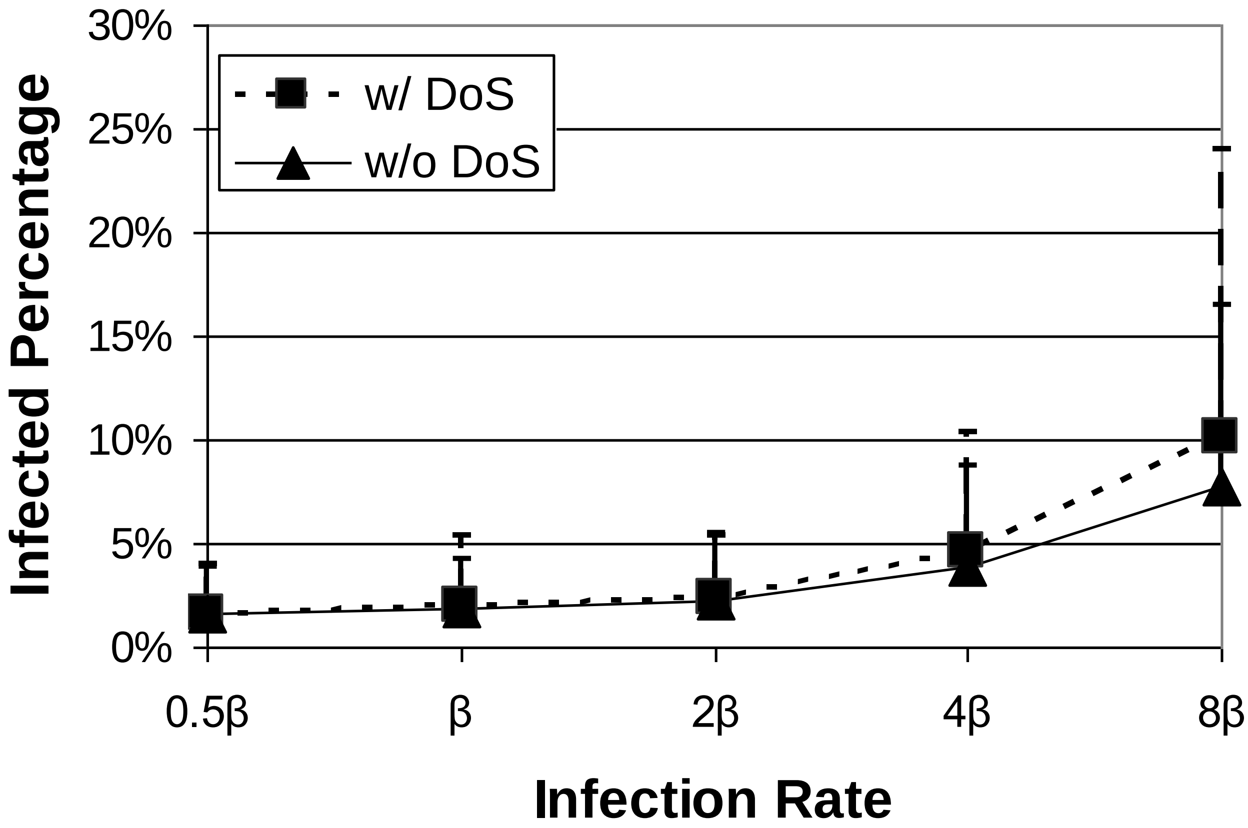
Simulating SCA propagation

- Susceptible/Infective epidemic model
- 500,000 node network on GeorgiaTech topology
- network congestion effects
  - RIPE data gathered during Slammer's outbreak
  - delay/loss increase linearly with infected hosts
- DoS attacks
  - infected hosts generate fake SCAs
  - verification increases linearly with number of SCAs

Containing Slammer

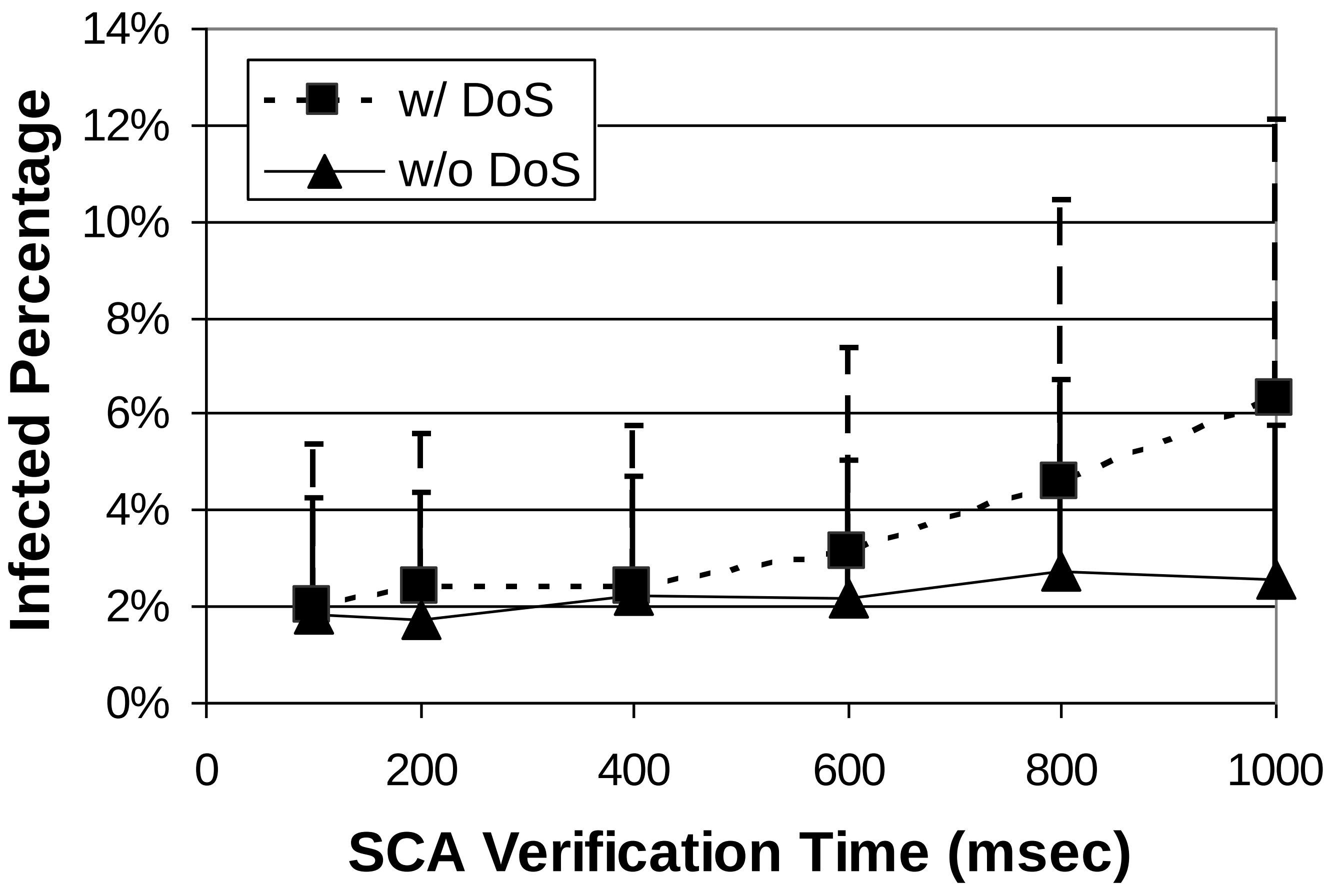


Increasing infection rate

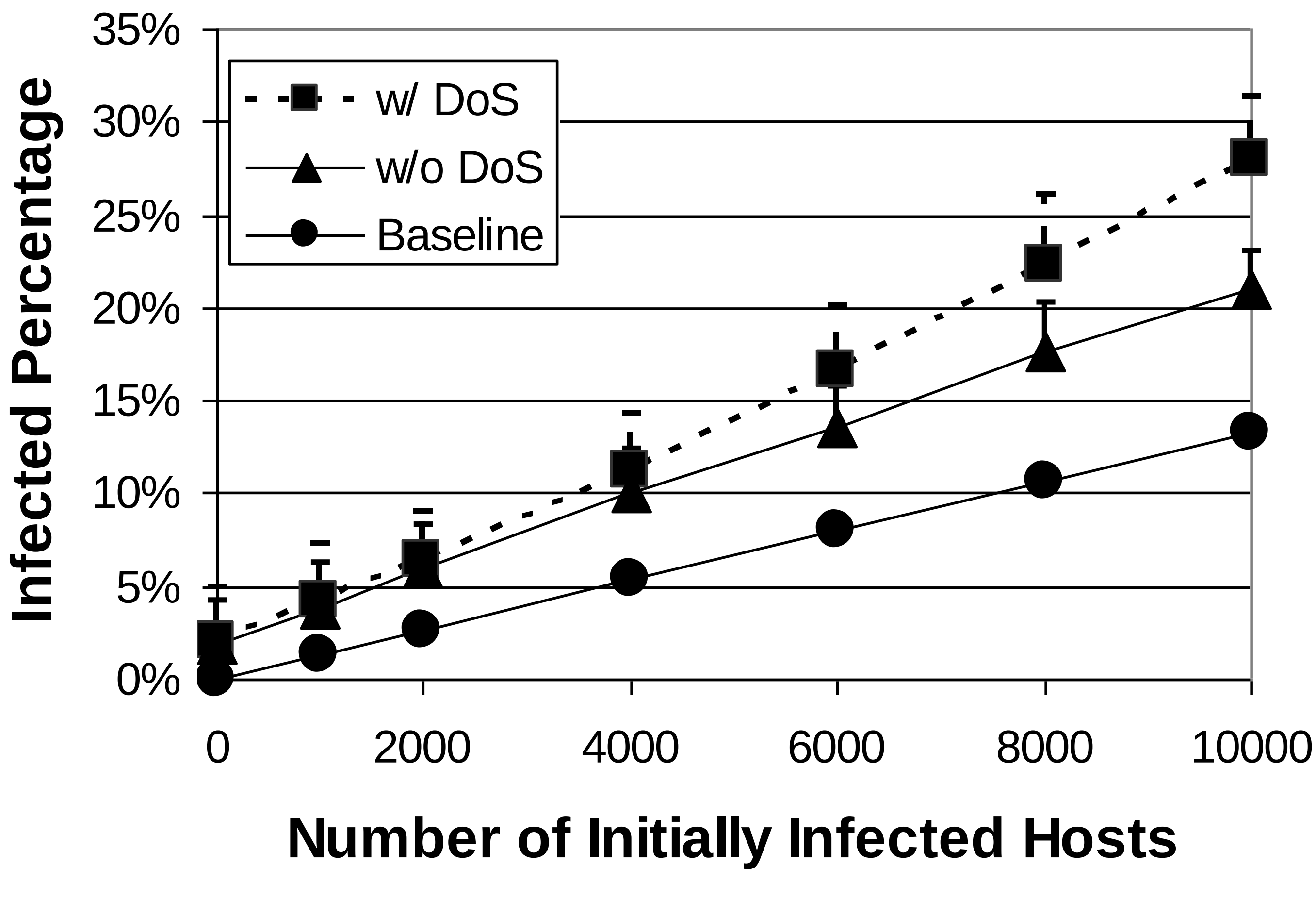


(ß is Slammer's infection rate)

Increasing verification time



Increasing seed hosts



Conclusion

- Vigilante can contain worms automatically
  - requires no prior knowledge of vulnerabilities
  - no false positives
  - low false negatives
  - works with today's binaries