Vigilante: End-to-End Containment of Internet Worms

Manuel Costa

Joint work with:
Jon Crowcroft, Miguel Castro, Ant Rowstron,
Lidong Zhou, Lintao Zhang, Paul Barham

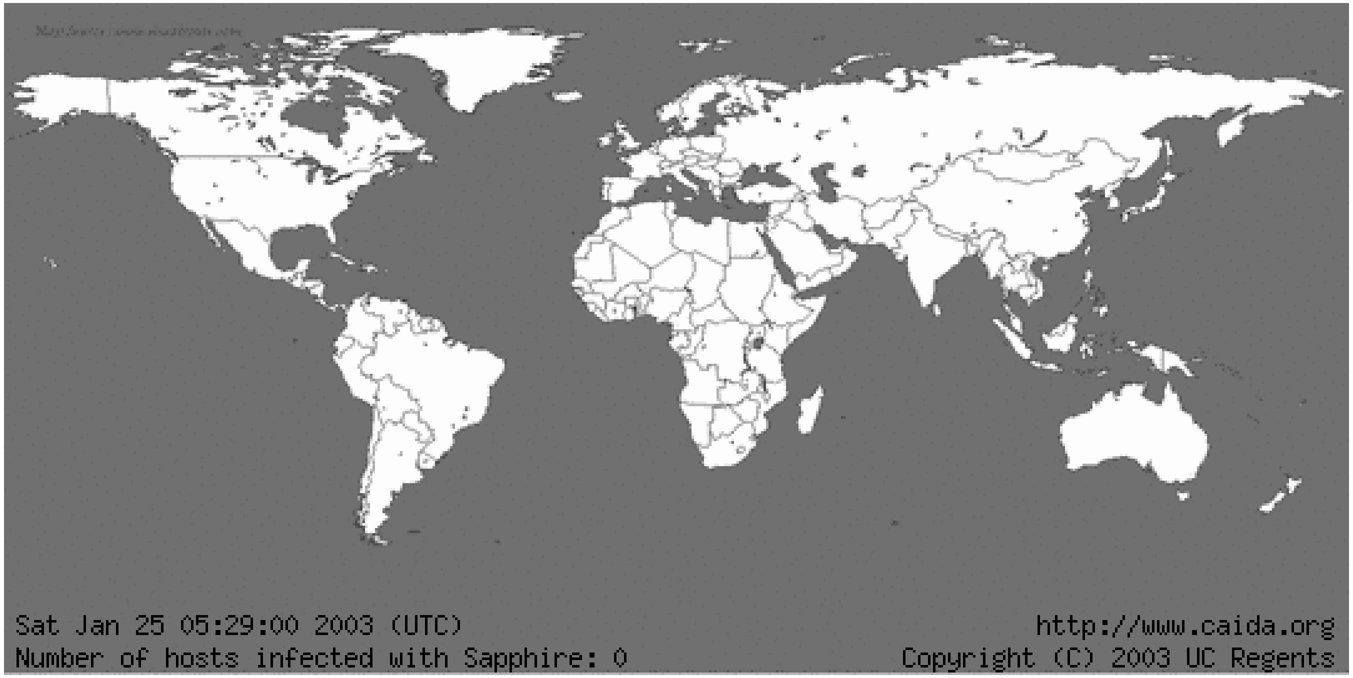
Microsoft Research
University of Cambridge, Computer Laboratory

The worm threat

- worms: self-propagating programs
  - spread without human help (unlike viruses)
- exploit low-level software defects
  - bypass network security protocols
- gain complete control of infected hosts
  - corrupt data, steal information, use hosts for spam/DoS

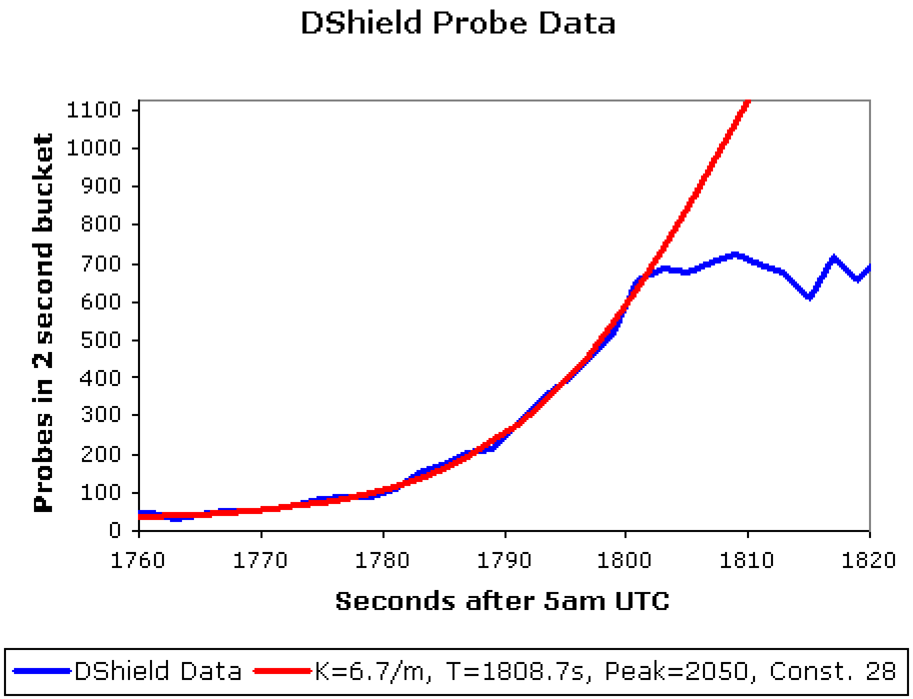
serious problems: banks closed, trains stopped, nuclear safety monitoring system disabled, …

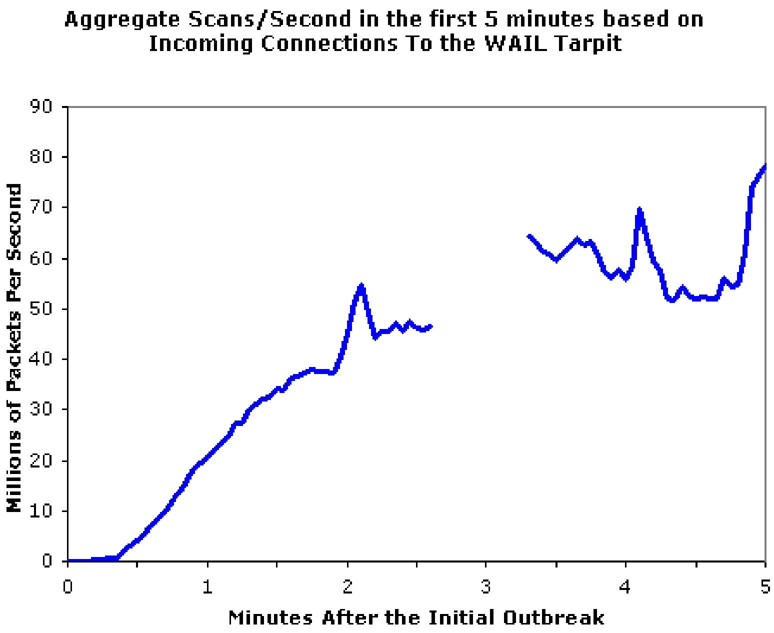
The Slammer incident



100 000 hosts infected
scanned 90% of Internet in 10 minutes

Slammer’s growth





exponential growth

doubled in size every 8.5 seconds

What can we do?

- prevention: eliminate vulnerabilities
  - static analysis, verification
  - better programming languages
  - testing, code reviews
- containment: detect and block worm
  - contain epidemic to small fraction of vulnerable hosts

containment must be automatic: worms spread too fast for human response

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

contains fast spreading worms:
no false positives, deployable today

Outline

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

Detection

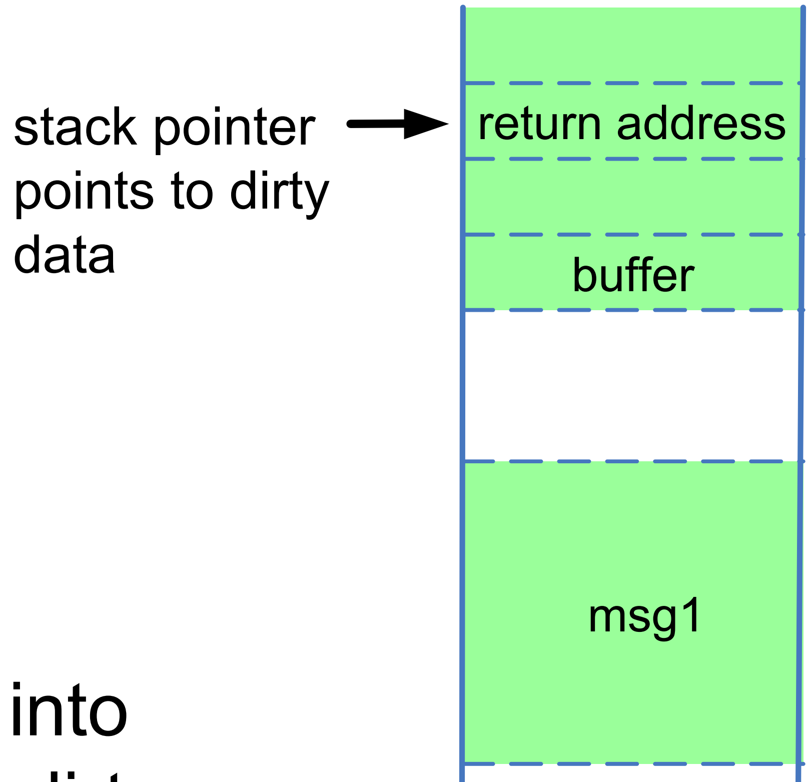
- diverse set of detection mechanisms
- diverse set of implementations
- detection mechanisms can have high-coverage

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
  - trap execution of dirty data
  - trap loading of dirty data into program counter
- high-coverage: stack, function pointers, …

Dynamic dataflow analysis

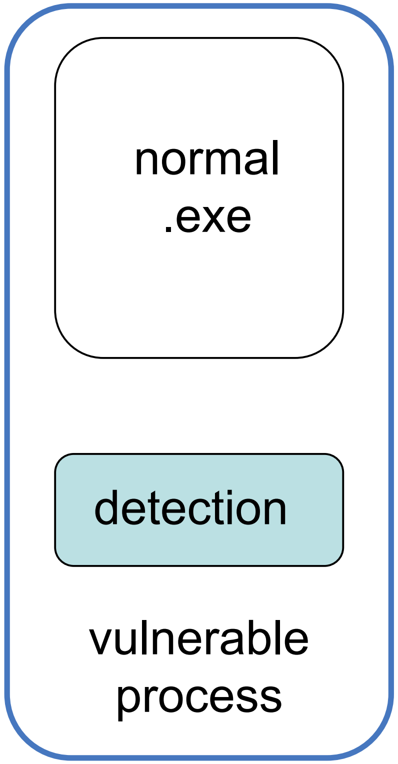
```
//vulnerable code
 push len
 push netbuf
 push sock
 call recv
 push netbuf
 push localbuf
 call strcpy
 ret
```



alert: value loaded into program counter is dirty

Dynamic dataflow analysis

- works with normal binaries
- instrumentation at runtime



Where are the detectors?

- general detectors are expensive
- centralized detectors can be attacked
- any host can be a detector
  - load sharing, high coverage, resilience
  - detectors create self-certifying alerts

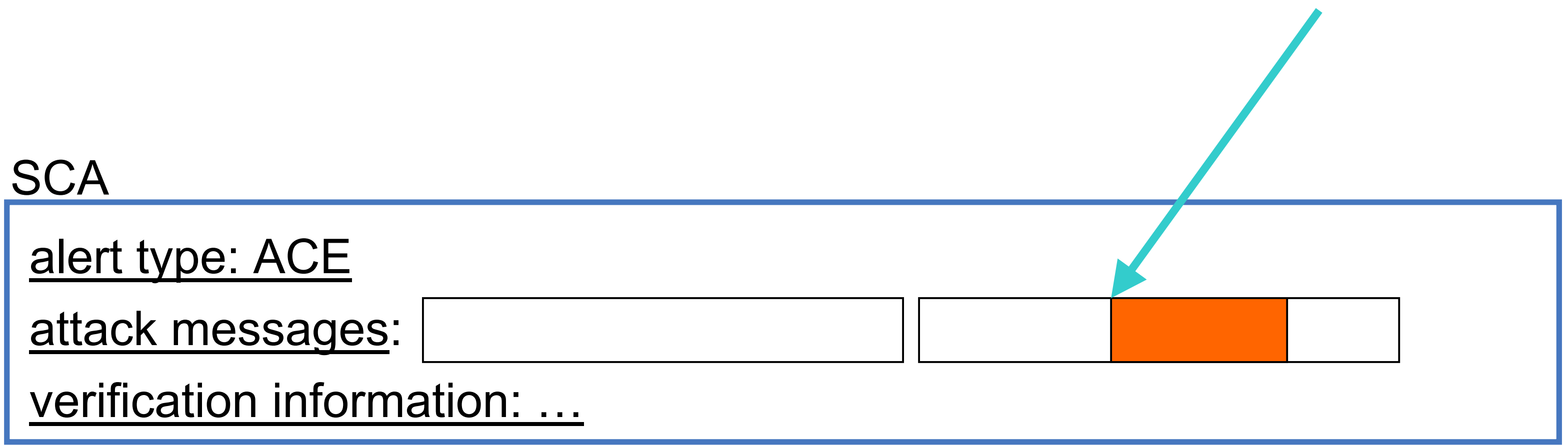
Self-certifying alerts

- **machine-verifiable proofs of vulnerability**
  - identify an application and a type of vulnerability
  - contain log of attack messages
  - contain **verification information**
- **enable hosts to verify if they are vulnerable**
  - replay infection with modified messages
  - verification has no false positives

SCA types

- arbitrary code execution (ACE)
- arbitrary execution control (AEC)
- arbitrary function argument (AFA)

what can the attacker do? → inject code
what is the verification information? → code location

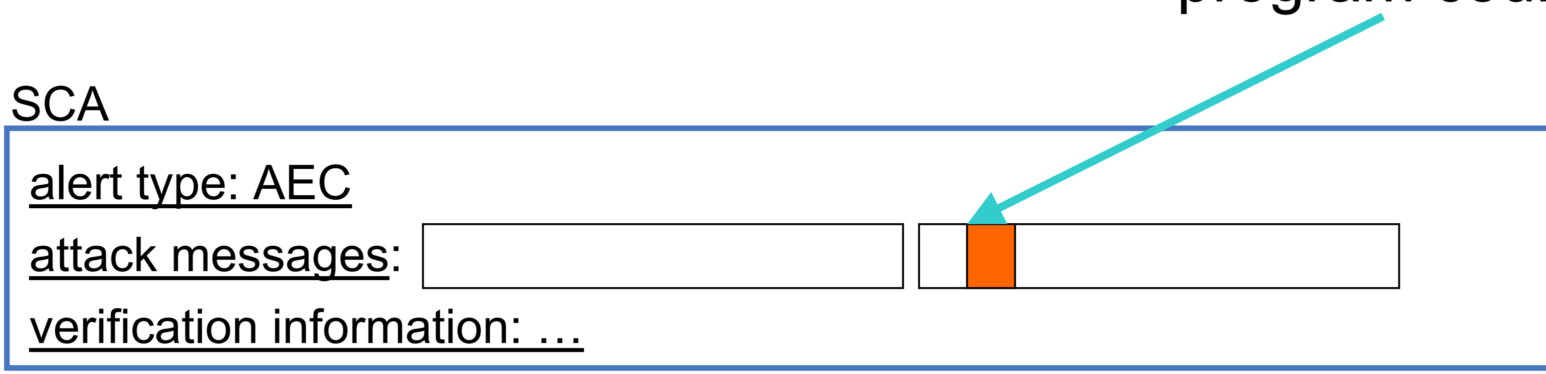


SCA types

- arbitrary code execution (ACE)
- arbitrary execution control (AEC)
- arbitrary function argument (AFA)

what can the attacker do? → force a control flow transfer

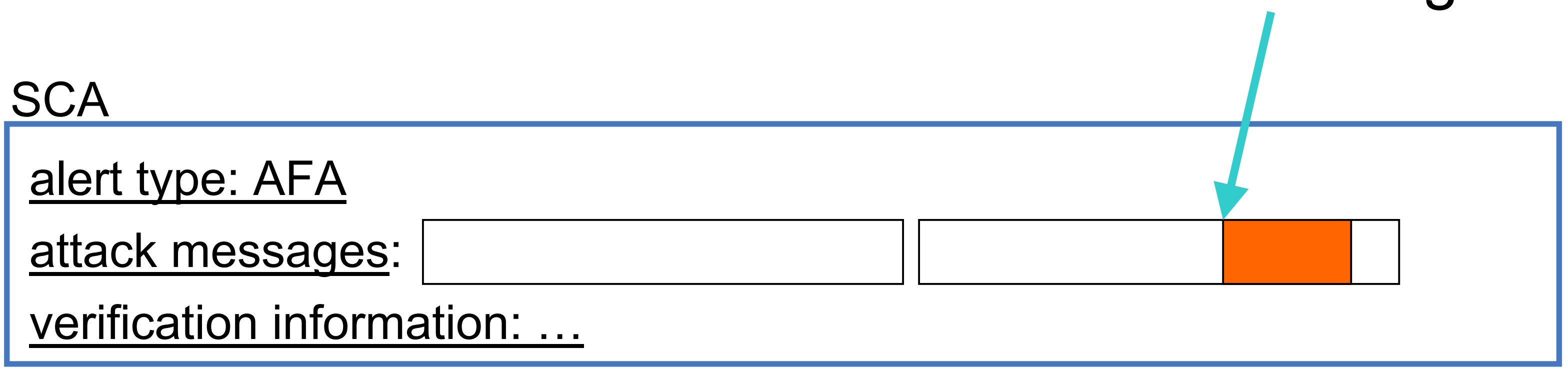
what is the verification information? → location of program counter



SCA types

- arbitrary code execution (ACE)
- arbitrary execution control (AEC)
- arbitrary function argument (AFA)

what can the attacker do?→supply an argument to a function
what is the verification information?→function name
location of argument

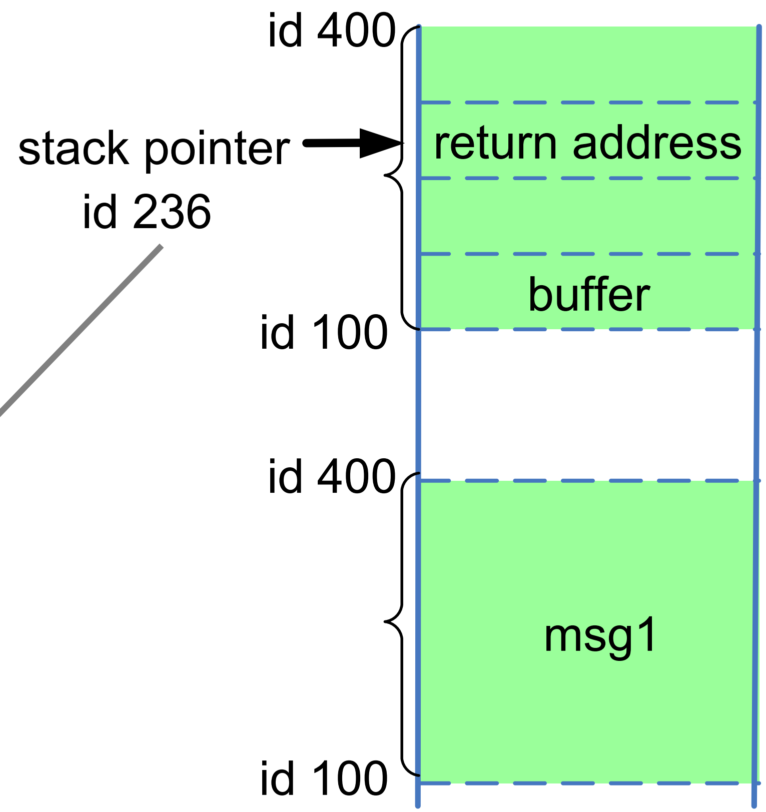


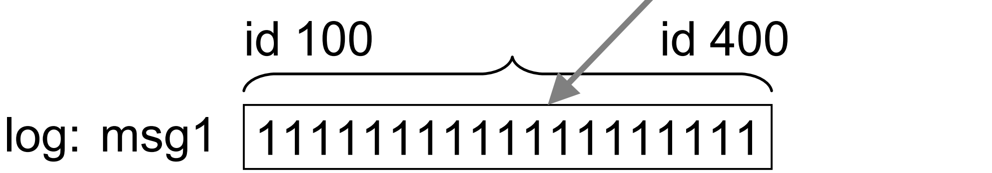
SCA generation

- log messages
- generate SCA when worm is detected
  - search log for relevant messages
  - compute verification information
  - generate tentative version of SCA
  - repeat until verification succeeds
- detectors may guide search

Generating an AEC alert

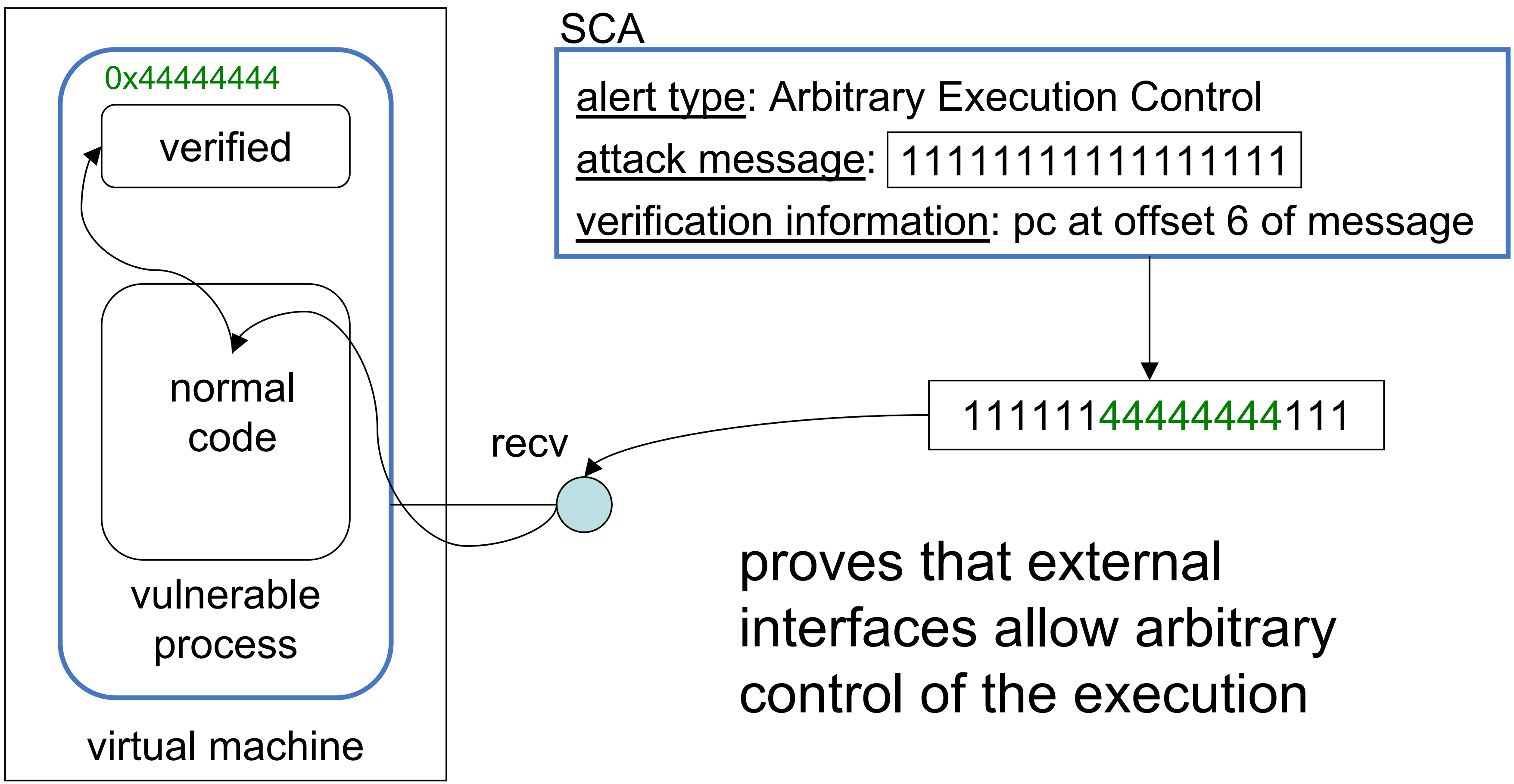
```
//vulnerable code
push len
push netbuf
push sock
call recv
push netbuf
push localbuf
call strcpy
ret
```





SCA: AEC, 11111111111111111111, pc at offset 136

Verifying an AEC alert



proves that external interfaces allow arbitrary control of the execution

verification is independent of detection mechanism

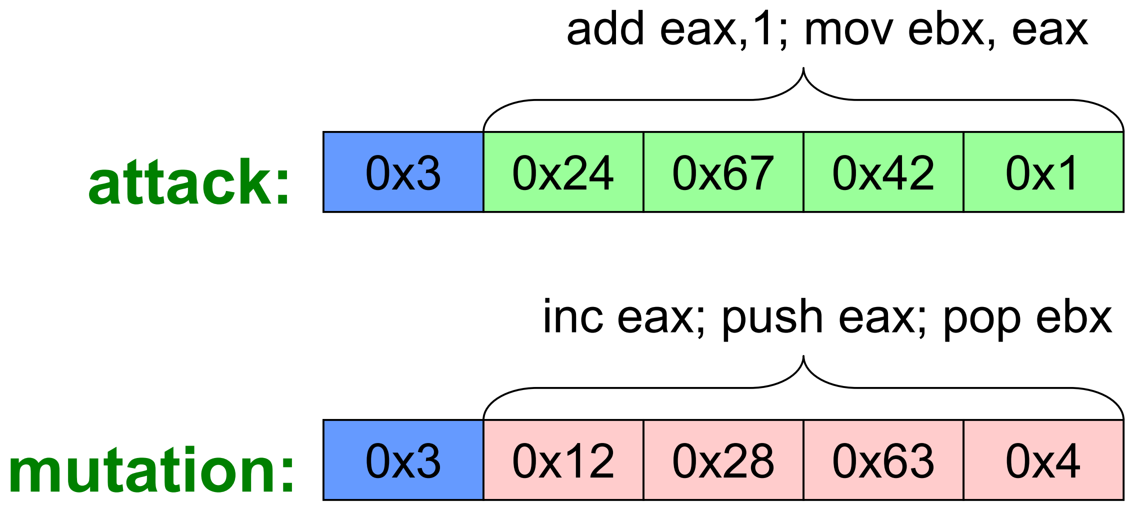
SCA broadcast

- uses overlay of superpeers
  - Akamai-like overlay with added security
  - 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
- mutations make protection difficult (as in real diseases)



Filter generation

- **dynamic data and control flow analysis**
  - track control and data flow from input messages
  - compute conditions that determine execution path
  - filter blocks messages that satisfy conditions
- uses full data flow information
  - dataflow graphs for dirty data and CPU flags
  - record decisions on conditional instructions

Generating filters for vulnerabilities

```
    //vulnerable code
    //recv msg
    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

Filters as program slices

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

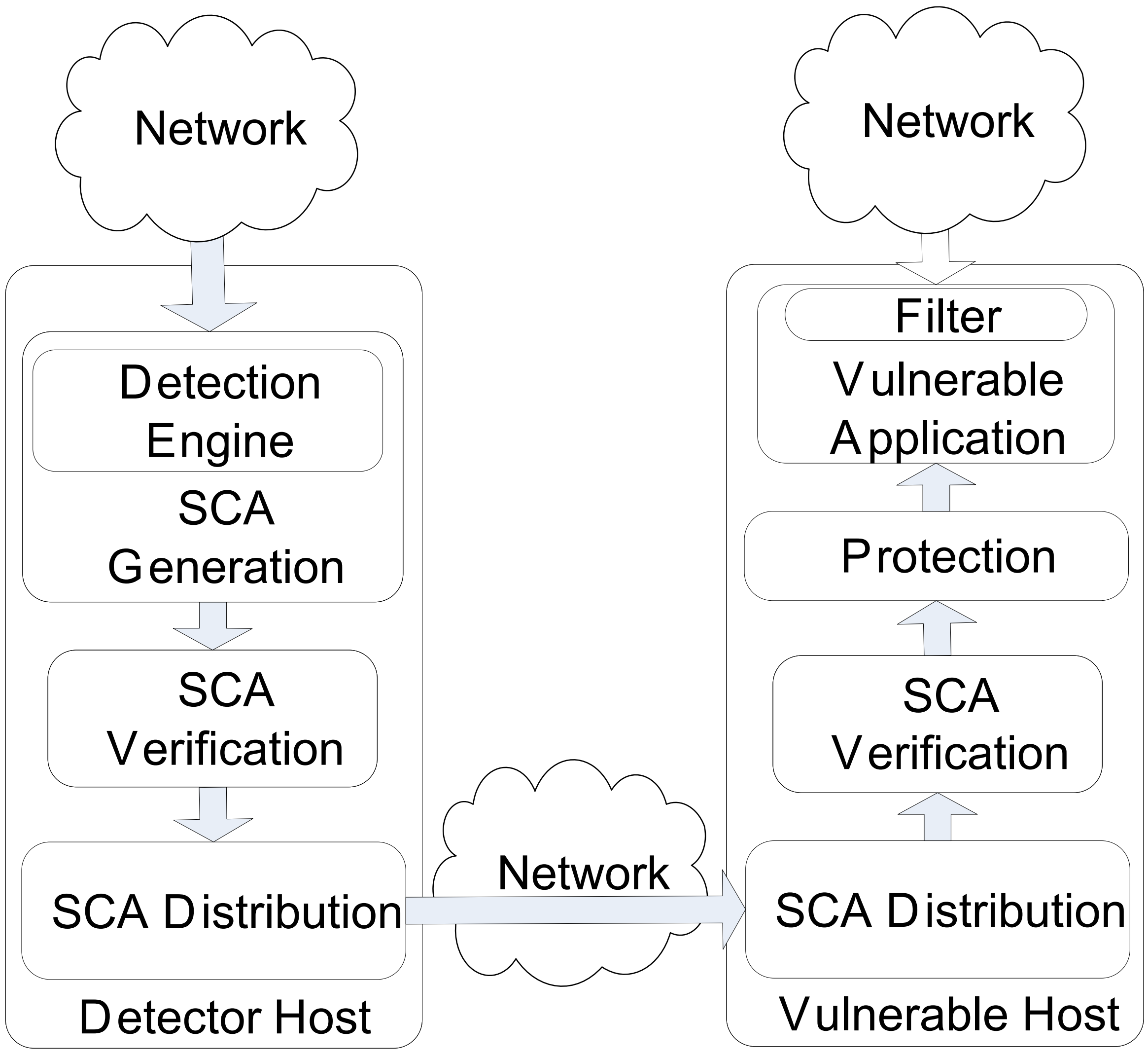
```
//recv msg
cmp msg[0],3; jne out
test msg[1],msg1[1]; je out
...
mov al,[msg]
mov cl,0x3
cmp al,cl
jne L2   //msg[0] == 3 ?
xor eax,eax
mov [esp+eax+4],cl
mov cl,[eax+msg+1]
inc eax
test cl,cl
jne L1   //msg[i] == 0 ?
ret
```

filters are a subset of the program’s instructions

Filters

- capture generic conditions
  - filters are assembly programs
- safe and efficient
  - no side effects, no loops
- two-filter design reduces false negatives
  - still improving

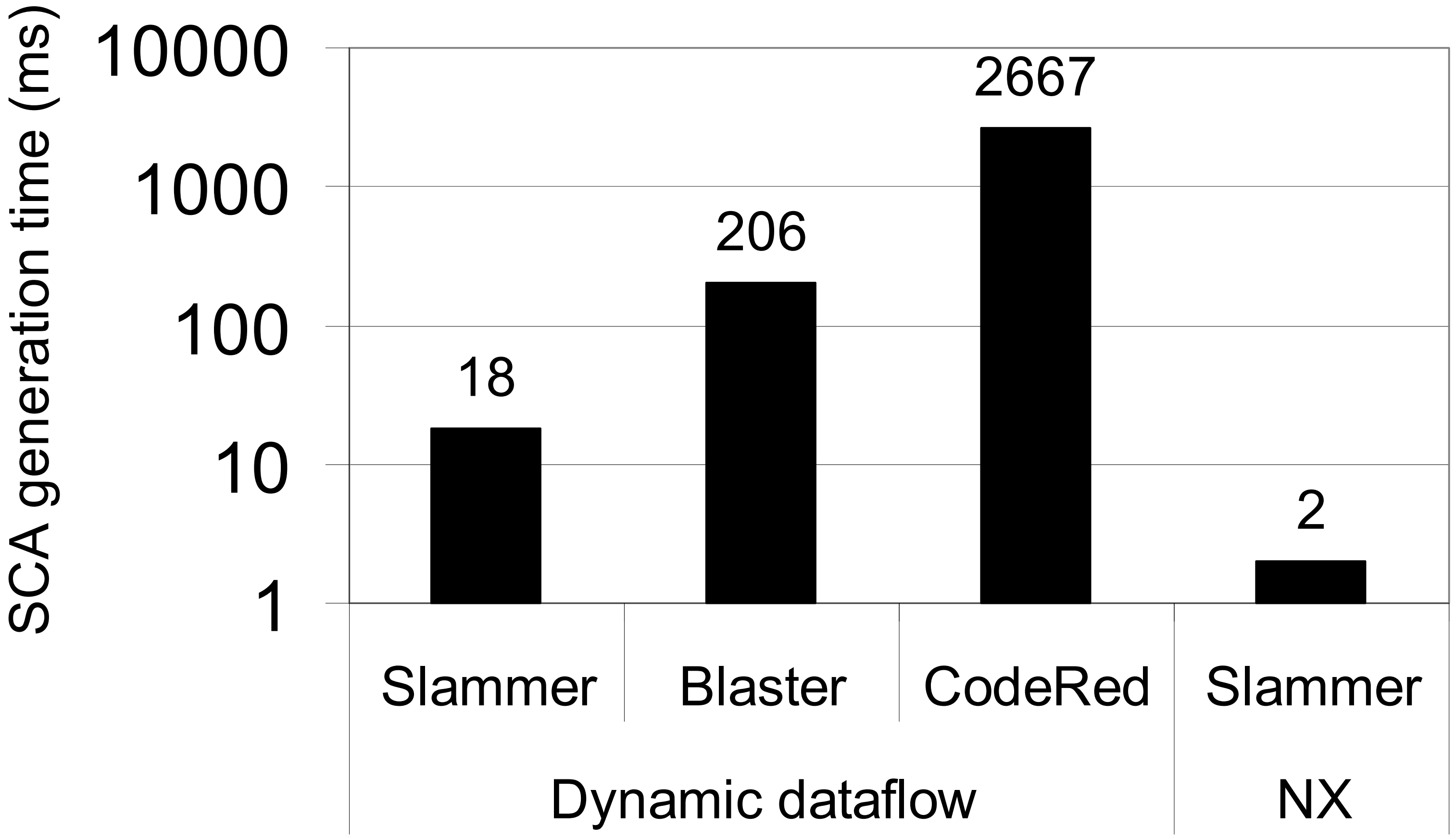
Putting it all together



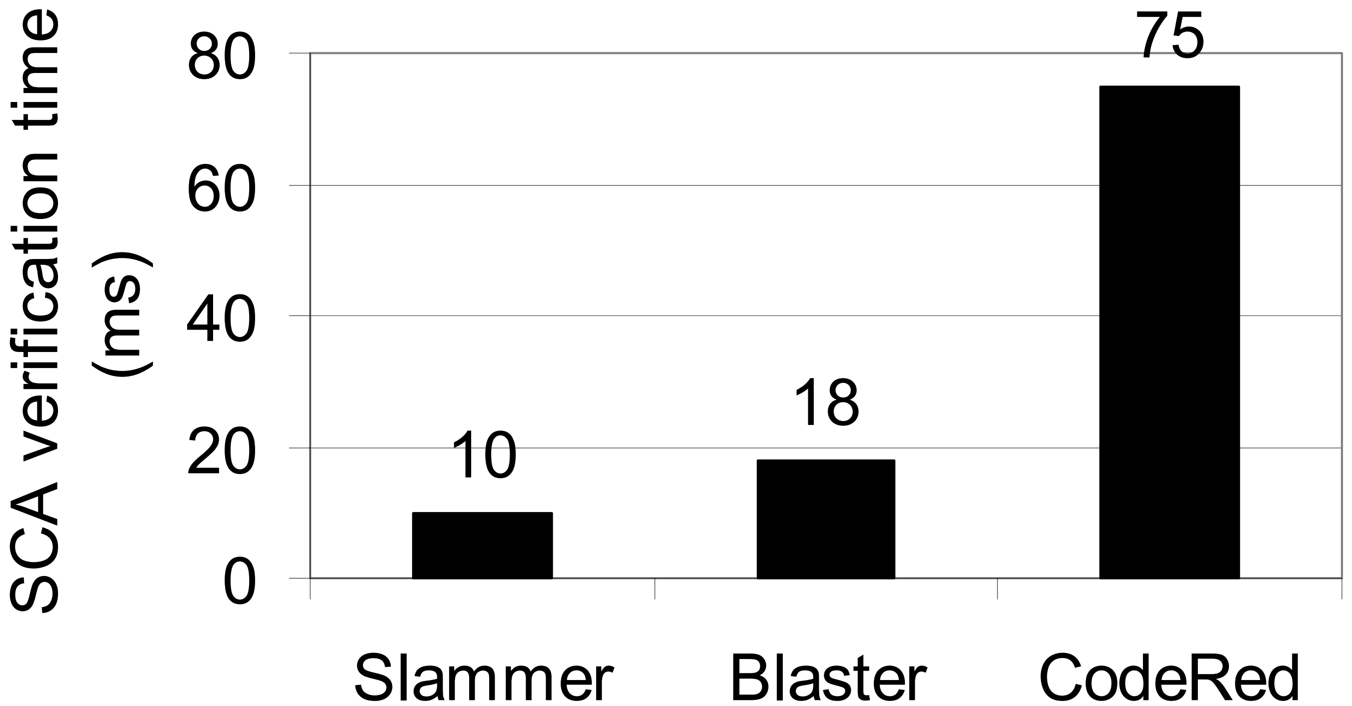
Evaluation

- 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

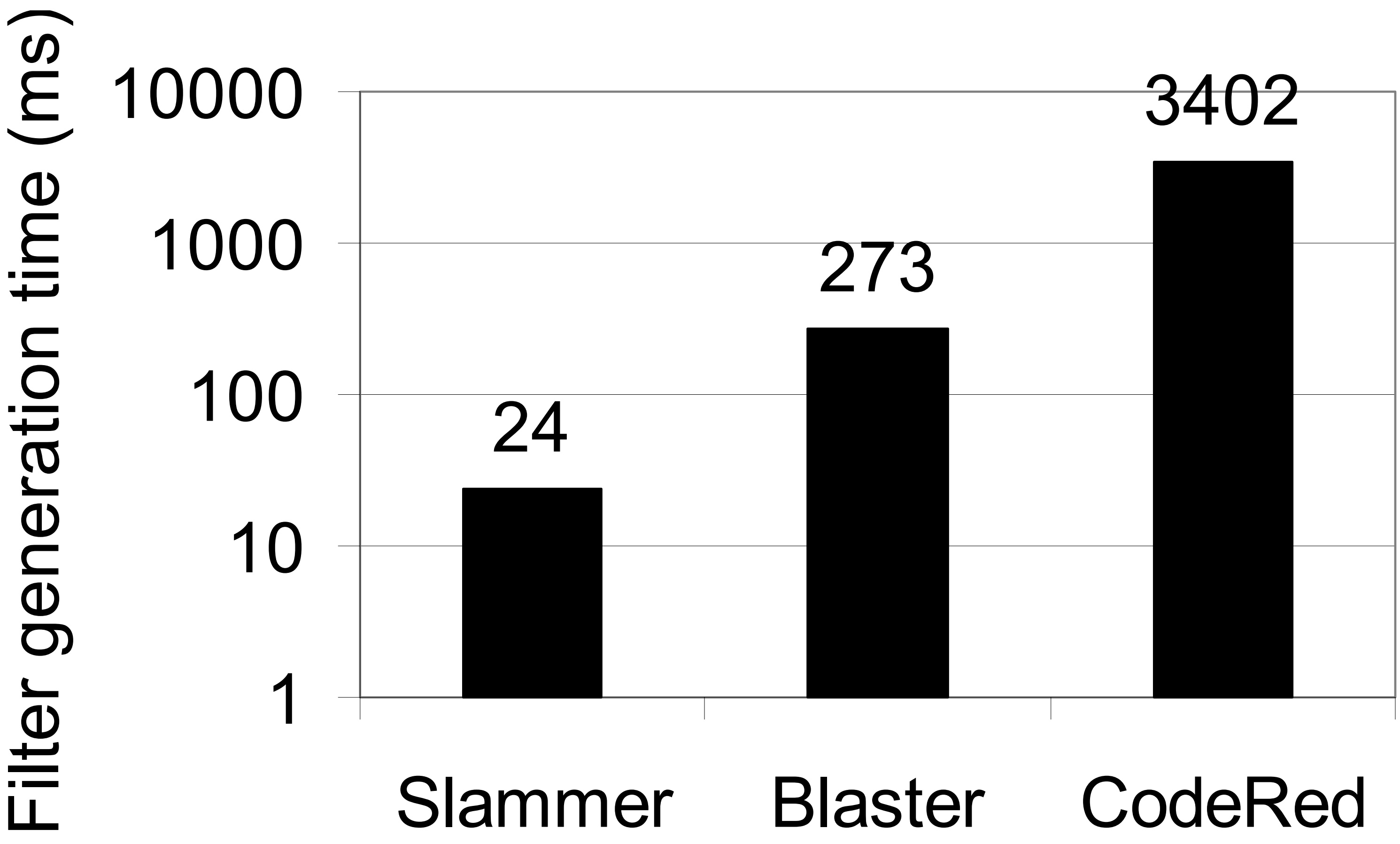
Time to generate SCAs



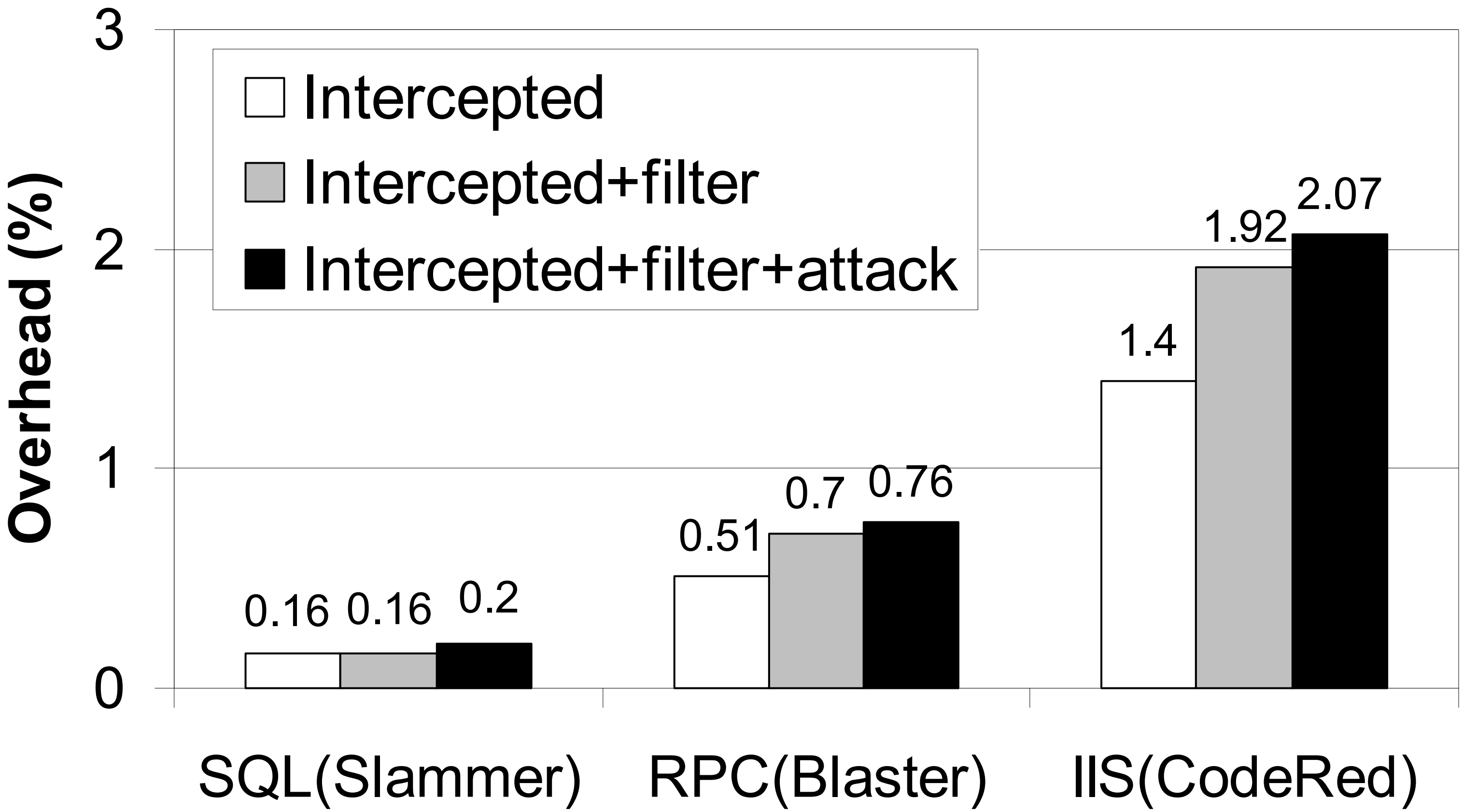
Time to verify SCAs



Time to generate filters



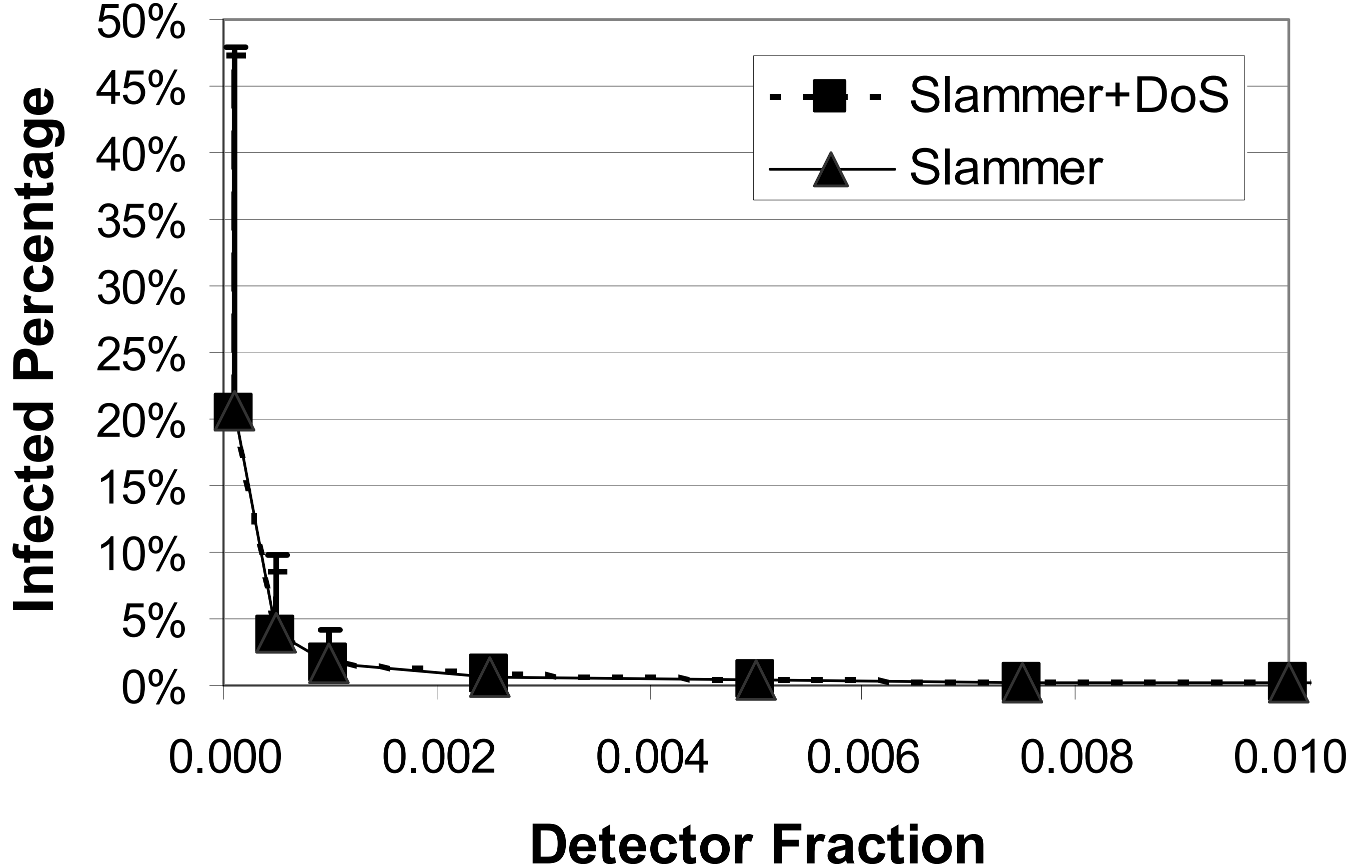
Filtering overhead



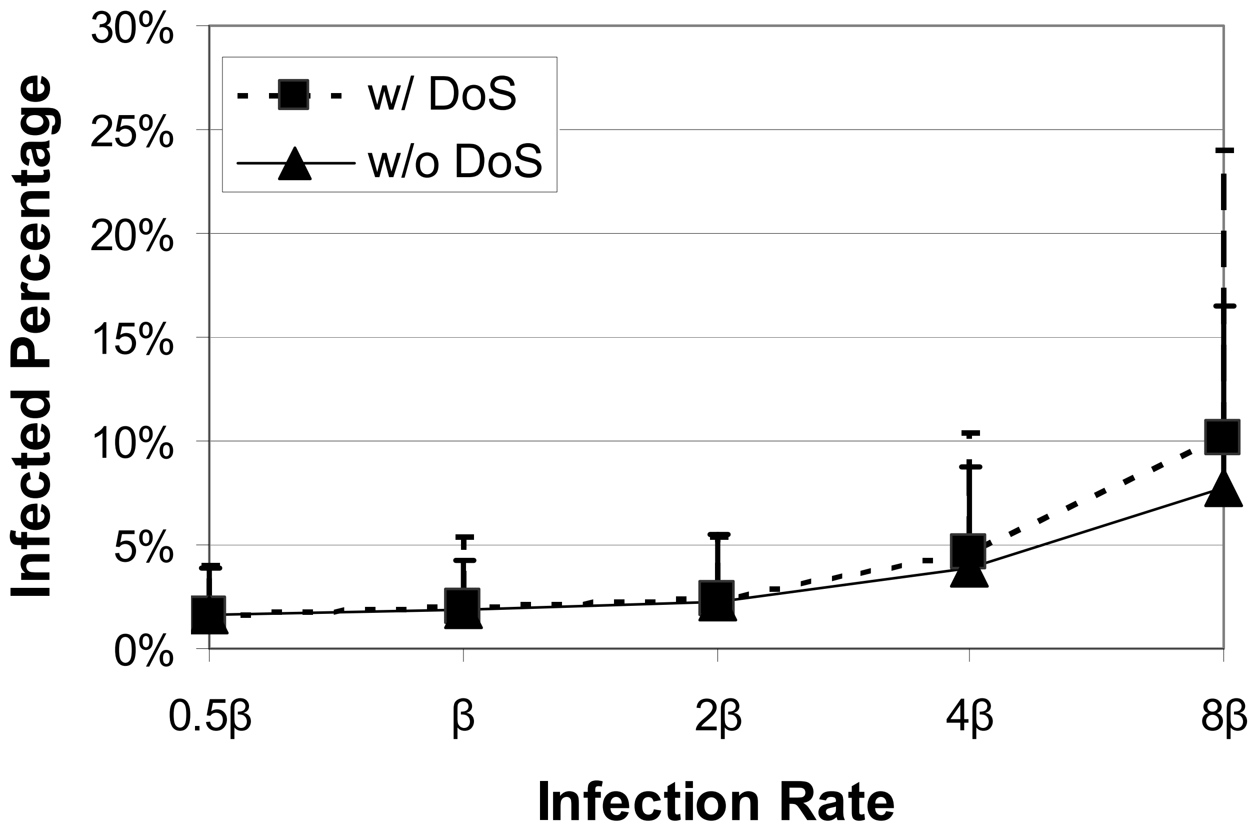
Simulating SCA propagation

- 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)

Related Work

- network related
  - signatures: Honeycomb, Autograph, EarlyBird, Polygraph; throttling [Williamson02]; scanning detectors [Weaver04]
- host related
  - program shepherding [Kiriansky02]; perl taint mode, concurrent work similar to dynamic dataflow analysis: [Suh04], Minos, TaintCheck; [Sidiroglou-Douskos05] related host-based system
- keep applications running despite attacks
  - failure oblivious computing, abort/rollback: DIRA, STEM, Rx
- human assisted, vulnerability specific detectors/filters
  - Shield, IntroVirt

Conclusion

- Vigilante can contain worms automatically
  - requires no prior knowledge of vulnerabilities
  - no false positives
  - low false negatives
  - deployable today