# Network-level Polymorphic Shellcode Detection using Emulation

Michalis Polychronakis, Kostas Anagnostakis, and Evangelos Markatos

Institute of Computer Science
Foundation for Research and Technology – Hellas
Crete, Greece

DIMVA'06 - 13 July 2006

# Outline

- Introduction – related work
- Evasion techniques
- Emulation-based detection
- Performance evaluation
- Open issues

# Remote System Compromise

*Attacker/worm exploits a software vulnerability*

1. Place the attack code into a buffer
2. Divert the execution flow of the vulnerable process
   - Stack/heap/integer overflow
   - Format string abuse
   - Arbitrary data corruption
3. Execute the injected code (*shellcode*)
   - Performs arbitrary operations under the privileges of the process that has been exploited

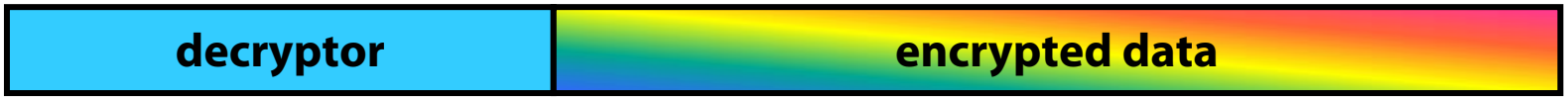
```
\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00
```

# Signature-based Detection

- Hand-crafted signatures
  - `GET default.ida?NNNNNNNNNNN…`
- Also for unknown attacks
  - Generic signatures for suspicious code sequences
  - NOP sleds, system calls, …
- Automated signature generation
  - Honeycomb, Earlybird, Autograph, PADS, Polygraph, Hamsa, …
  - Common idea: find invariant parts among multiple attack instances
  - Then turned into token subsequences ➔ regular expressions
  - Effective only for noisy worm-like attacks

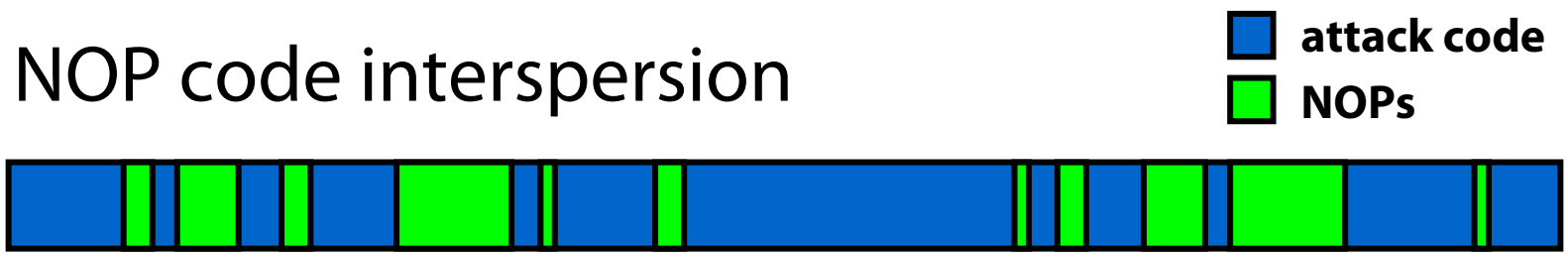
# **Polymorphism** (1/2)

- Naïve encryption




  - The decryptor remains the same in each attack instance
  - ***Easy to fingerprint using typical string signatures***

- NOP code interspersion




  - NOPs' type/position/number varies in each instance
  - ***Can be fingerprinted using regular expressions***

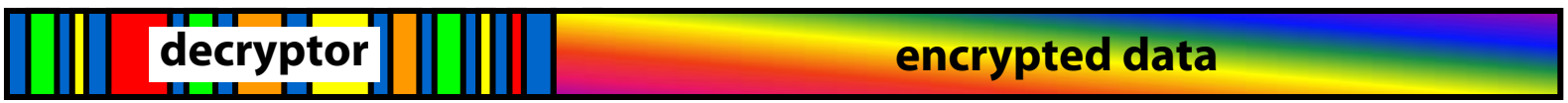
# **Polymorphism** (2/2)

- Code obfuscation/metamorphism
  - Instruction substitution
  - Code block transposition
  - Register reassignment
  - Dead code insertion

```
mov eax,0xF3   ➡   push 0xF3
                   pop eax
```

  - ***Hard to fingerprint using regexps if applied extensively***

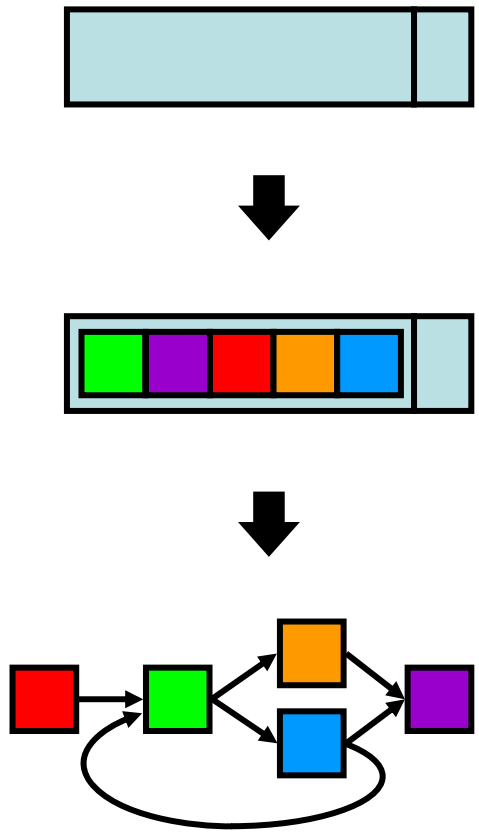
- **Combination of all techniques**




  - ***Signature extraction becomes infeasible***

# Static Analysis Based Detection

- Recent proposals heuristically identify malicious code inside network flows using static binary code analysis
  - [Kruegel'05, Chinchani'05, Payer'05, Wang'06]
- Step forward – beyond pattern-matching
  - Do not depend on invariant content
- Basic steps
  1. **Disassembly**
  2. **Control Flow Graph extraction**
- Initial approaches focused only on the shellcodes' sled component
  - Abstract Payload Execution [Kruegel'02] Pioneer network-level static analysis work
  - Orthogonal to above approaches



# **Static Analysis Resistant Shellcode** (1/4)

- Static binary code analysis is generally accurate for compiled and well-structured binaries
- Shellcode is not normal code!
  - Written/tweaked at assembly level: complete freedom…
- The attacker can specially craft the shellcode to hinder disassembly and CFG extraction
  - Anti-disassembly tricks
  - Indirect addressing `jmp ebx`
  - **Self-modifying code**

# Static Analysis Resistant Shellcode (2/4)

- Running example
  - Encrypted shellcode generated by the Countdown engine of the Metasploit Framework
  - Slightly modified with a self-modification

```
\x6A\x7F\x59\xE8\xFF\xFF\xFF\xFF\xC1\x5E\x80
\x46\x0A\xE0\x30\x4C\x0E\x0B\x02\xFA...
```

- Let's try to figure out what this code does

# Static Analysis Resistant Shellcode (3/4)

- Linear disassembly can be easily tricked

Linear Disassembly

```
00  6A7F       push byte +0x7f
02  59         pop ecx
03  E8FFFFFFFF call 0x7
08  C15E8046   rcr [esi-0x80],0x46
0C  0AE0       or ah,al
0E  304C0E0B   xor [esi+ecx+0xb],cl
12  02FA       add bh,dl
14
... <encrypted shellcode>
93
```

# Static Analysis Resistant Shellcode (3/4)

- Linear disassembly can be easily tricked

Linear Disassembly

```
00  6A7F        push byte +0x7f
02  59          pop ecx
03  E8FFFFFFFF  call 0x7
08  C15E8046    rcr [esi-0x80],0x46
0C  0AE0        or ah,al
0E  304C0E0B    xor [esi+ecx+0xb],cl
12  02FA        add bh,dl
14
... <encrypted shellcode>
93
```

**Jumps to the middle of itself**

# Static Analysis Resistant Shellcode (3/4)

- Linear disassembly can be easily tricked

Linear Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
08   C15E8046    rcr [esi-0x80],0x46
0C   0AE0        or ah,al
0E   304C0E0B    xor [esi+ecx+0xb],cl
12   02FA        add bh,dl
14
...  <encrypted shellcode>
93
```

Recursive Traversal Disassembly

```
00   6A7F        push byte +0xf
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   02FA        add bh,dl
14
...  <encrypted shellcode>
93
```

# Static Analysis Resistant Shellcode (3/4)

- Linear disassembly can be easily tricked

Linear Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
08   C15E8046    rcr [esi-0x80],0x46
0C   0AE0        or ah,al
0E   304C0E0B    xor [esi+ecx+0xb],cl
12   02FA        add bh,dl
14
... <encrypted shellcode>
93
```

Recursive Traversal Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   02FA        add bh,dl
14
... <encrypted shellcode>
93
```

**much better, but not the real code that will be eventually executed!**

- Recursive traversal disassembly is still not enough…

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

```
00  6A7F        push byte +0x7f
02  59          pop ecx
03  E8FFFFFFFF  call 0x7
07  FFC1        inc ecx
09  5E          pop esi
0a  80460AE0    add [esi+0xa],0xe0
0e  304C0E0B    xor [esi+ecx+0xb],cl
12  02FA        add bh,dl
14
... <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
```

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   02FA        add bh,dl
14
... <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
pop ecx                    ecx = 0x7F
```

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   02FA        add bh,dl
14
... <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
pop ecx                 ecx = 0x7F
call 0x7                (push 0x8)
```

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

```
00  6A7F        push byte +0x7f
02  59          pop ecx
03  E8FFFFFFFF  call 0x7
07  FFC1        inc ecx
09  5E          pop esi
0a  80460AE0    add [esi+0xa],0xe0
0e  304C0E0B    xor [esi+ecx+0xb],cl
12  02FA        add bh,dl
14
... <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
pop ecx                 ecx = 0x7F
call 0x7               (push 0x8)
inc ecx                 ecx = 0x80
```

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   02FA        add bh,dl
14
...  <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
pop ecx                 ecx = 0x7F
call 0x7               (push 0x8)
inc ecx                 ecx = 0x80
pop esi                 esi = 0x8
```

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   02FA        add bh,dl
14
... <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
pop ecx                 ecx = 0x7F
call 0x7               (push 0x8)
inc ecx                 ecx = 0x80
pop esi                 esi = 0x8
add [esi+0xa],0xe0      ADD [12] 0xE0
```

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   E2FA        loop 0xe
14
...  <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
pop ecx                 ecx = 0x7F
call 0x7               (push 0x8)
inc ecx                 ecx = 0x80
pop esi                 esi = 0x8
add [esi+0xa],0xe0      ADD [12] 0xE0
```

**Self-modification**

```
0x02FA   +   0xE0   =   0xE2FA
add bh,dl           →   loop 0xe
```

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   E2FA        loop 0xe
14
...  <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
pop ecx                 ecx = 0x7F
call 0x7               (push 0x8)
inc ecx                 ecx = 0x80
pop esi                 esi = 0x8
add [esi+0xa],0xe0      ADD [12] 0xE0
xor [esi+ecx+0xb],cl    XOR [93] 0x80
```

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   E2FA        loop 0xe
14
... <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
pop ecx                 ecx = 0x7F
call 0x7               (push 0x8)
inc ecx                 ecx = 0x80
pop esi                 esi = 0x8
add [esi+0xa],0xe0      ADD [12] 0xE0
xor [esi+ecx+0xb],cl    XOR [93] 0x80
loop 0xe               (ecx = 0x7F)
```

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   E2FA        loop 0xe
14
...  <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
pop ecx                 ecx = 0x7F
call 0x7               (push 0x8)
inc ecx                 ecx = 0x80
pop esi                 esi = 0x8
add [esi+0xa],0xe0      ADD [12] 0xE0
xor [esi+ecx+0xb],cl    XOR [93] 0x80
loop 0xe               (ecx = 0x7F)
xor [esi+ecx+0xb],cl    XOR [92] 0x7F
```

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   E2FA        loop 0xe
14
...  <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
pop ecx                 ecx = 0x7F
call 0x7               (push 0x8)
inc ecx                 ecx = 0x80
pop esi                 esi = 0x8
add [esi+0xa],0xe0      ADD [12] 0xE0
xor [esi+ecx+0xb],cl    XOR [93] 0x80
loop 0xe               (ecx = 0x7F)
xor [esi+ecx+0xb],cl    XOR [92] 0x7F
loop 0xe               (ecx = 0x7E)
```

# Static Analysis Resistant Shellcode (4/4)

- Self-modifying code can hide the real CFG

Recursive Traversal Disassembly

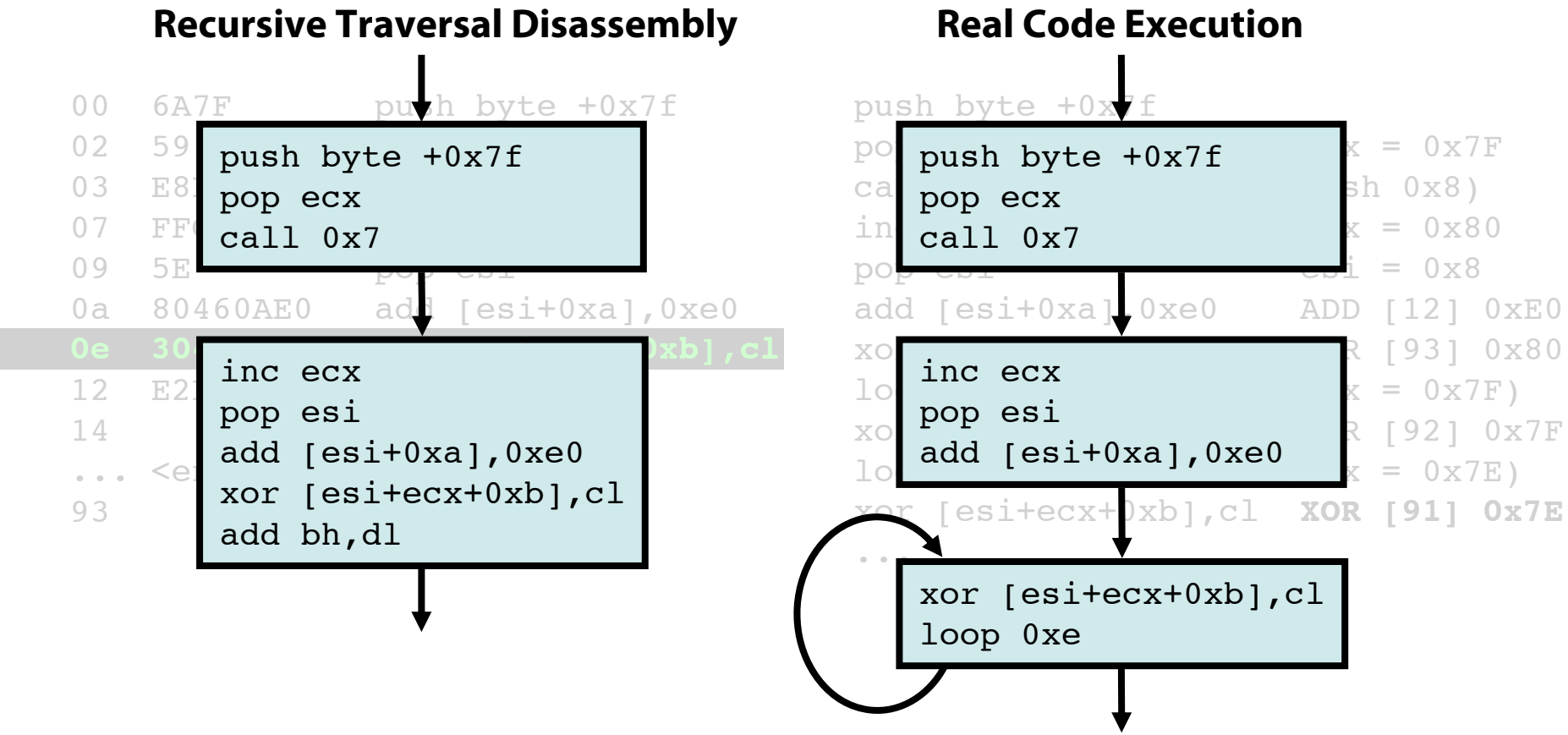
```
00   6A7F        push byte +0x7f
02   59          pop ecx
03   E8FFFFFFFF  call 0x7
07   FFC1        inc ecx
09   5E          pop esi
0a   80460AE0    add [esi+0xa],0xe0
0e   304C0E0B    xor [esi+ecx+0xb],cl
12   E2FA        loop 0xe
14
...  <encrypted shellcode>
93
```

Real Code Execution

```
push byte +0x7f
pop ecx                 ecx = 0x7F
call 0x7               (push 0x8)
inc ecx                 ecx = 0x80
pop esi                 esi = 0x8
add [esi+0xa],0xe0      ADD [12] 0xE0
xor [esi+ecx+0xb],cl    XOR [93] 0x80
loop 0xe               (ecx = 0x7F)
xor [esi+ecx+0xb],cl    XOR [92] 0x7F
loop 0xe               (ecx = 0x7E)
xor [esi+ecx+0xb],cl    XOR [91] 0x7E
...
```

# Static Analysis Resistant Shellcode (4/4)

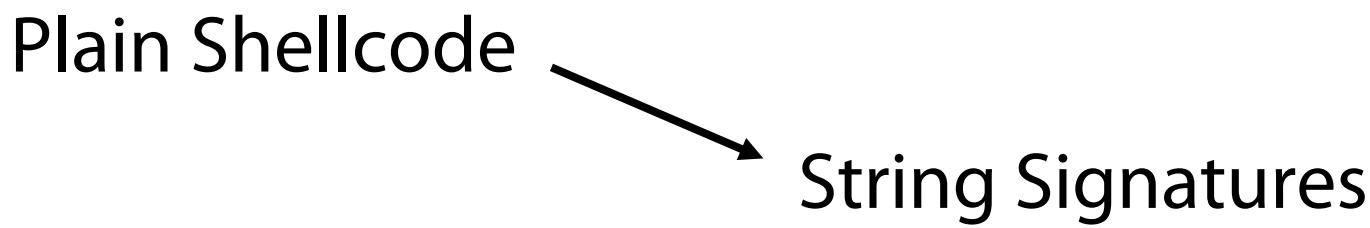
- Self-modifying code can hide the real CFG



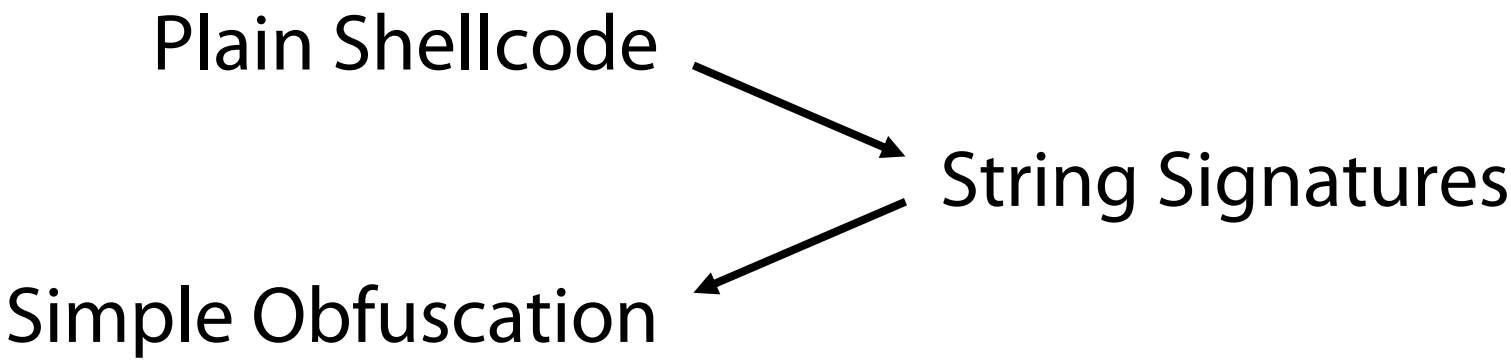
# Attacks – Defenses Coevolution

Plain Shellcode

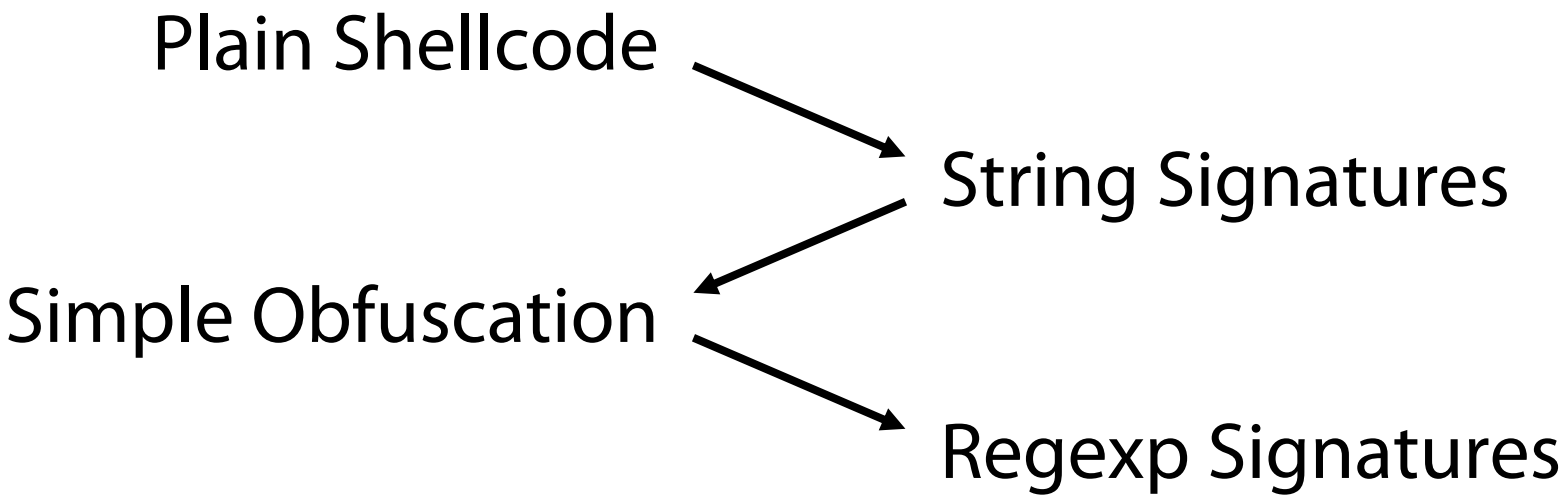
# Attacks – Defenses Coevolution



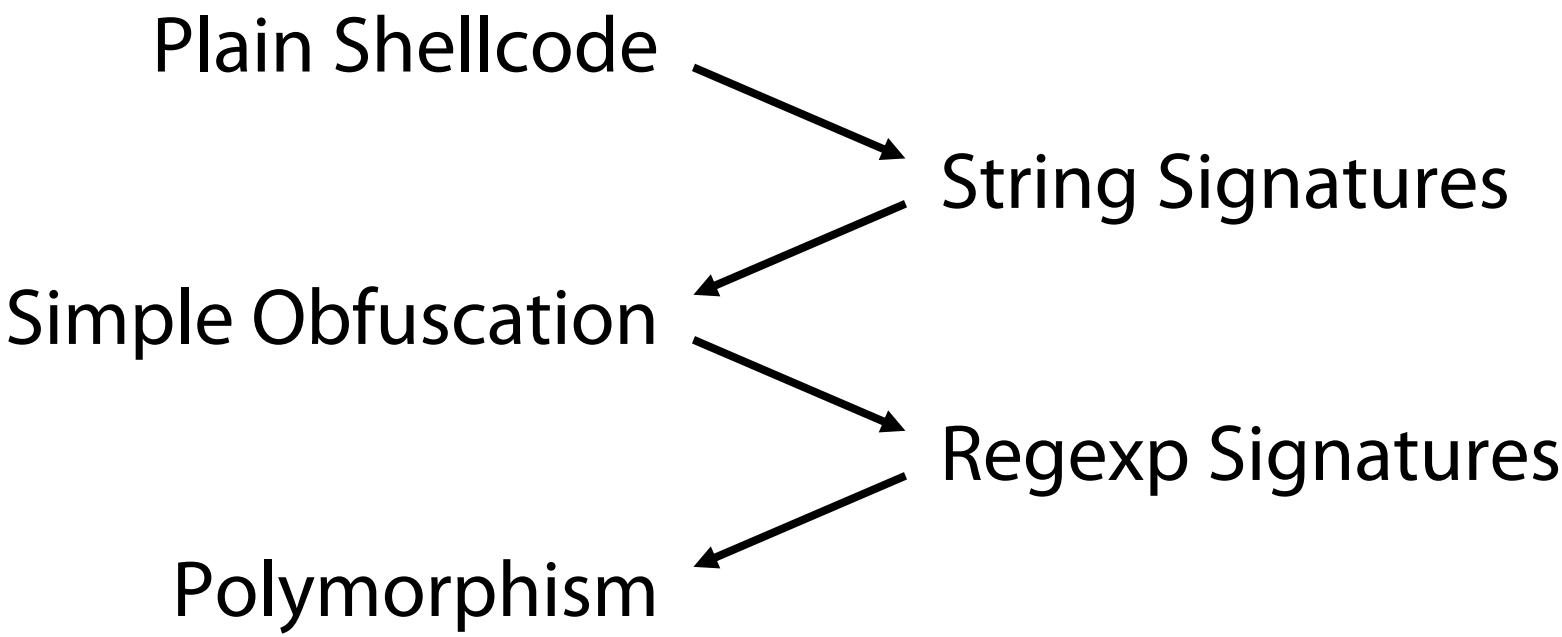
# Attacks – Defenses Coevolution



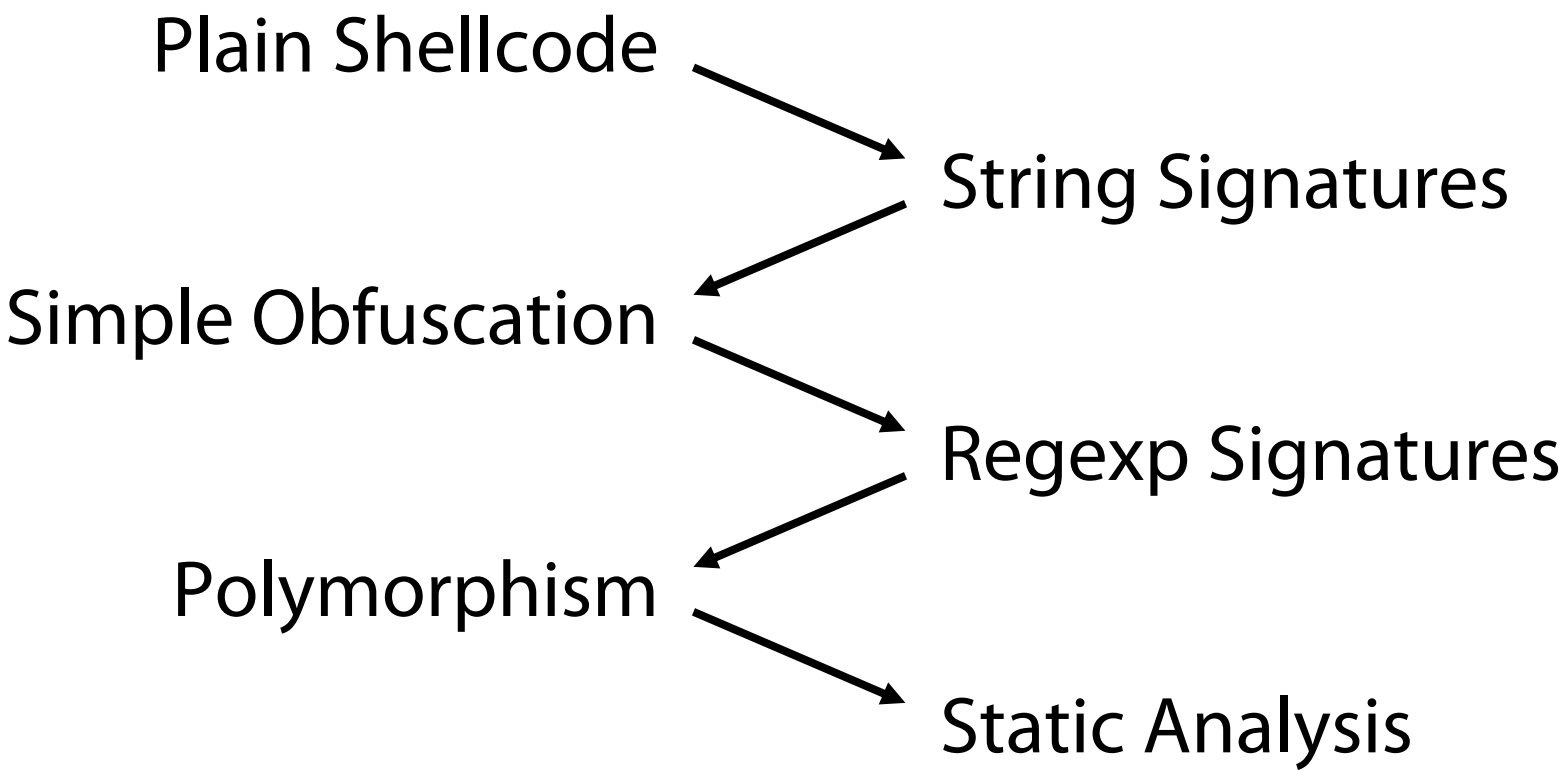
# Attacks – Defenses Coevolution



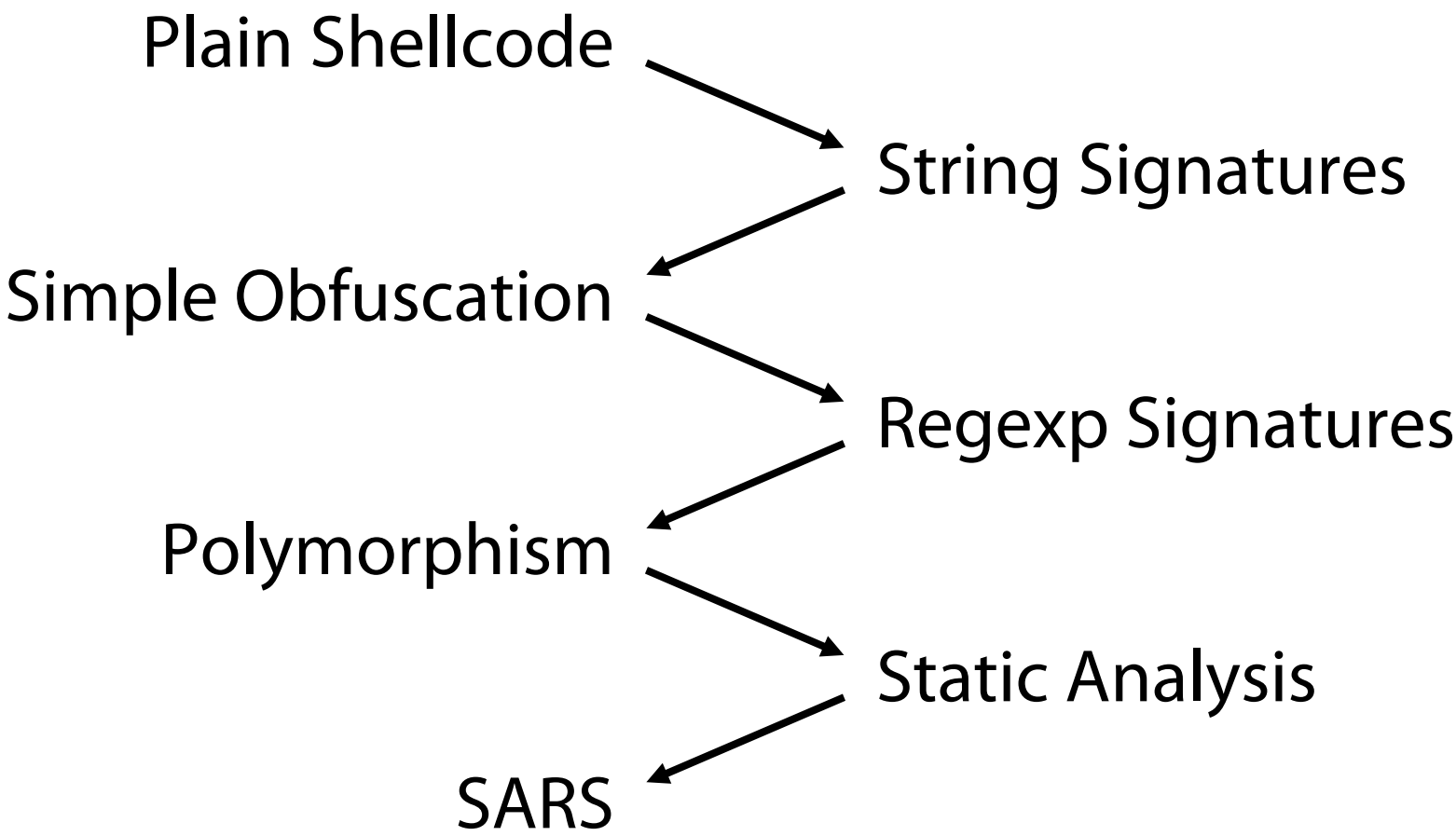
# Attacks – Defenses Coevolution



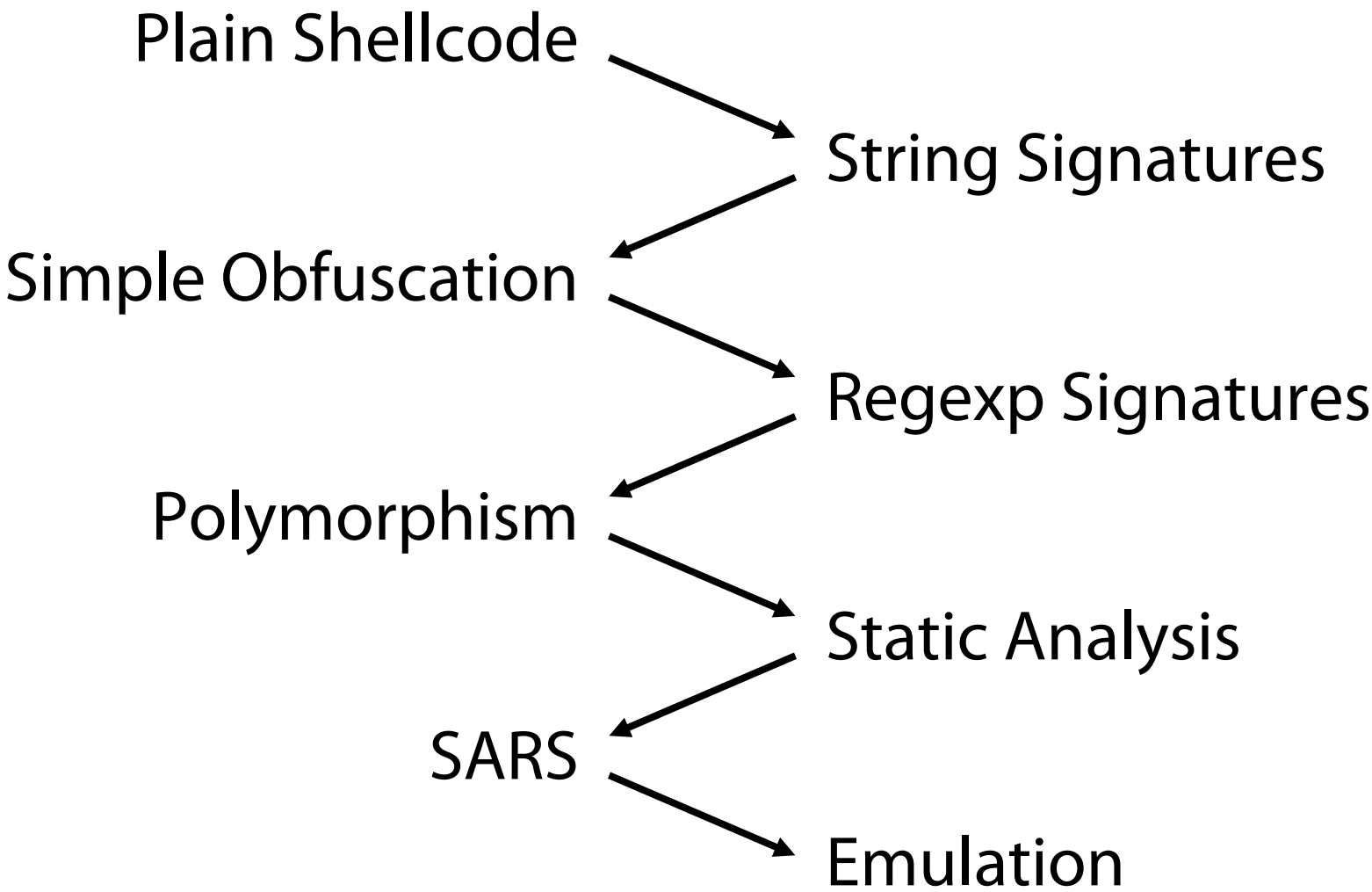
# Attacks – Defenses Coevolution



# Attacks – Defenses Coevolution



# Attacks – Defenses Coevolution



# Network-level Polymorphic Shellcode Detection

- **Motivation:** Highly obfuscated code will not reveal its actual form until it is executed
- **Main idea:** execute each input request as if it were executable code
- **Goal:** identify the execution behavior inherent in polymorphic shellcodes

# **Network-level Emulation** (1/2)

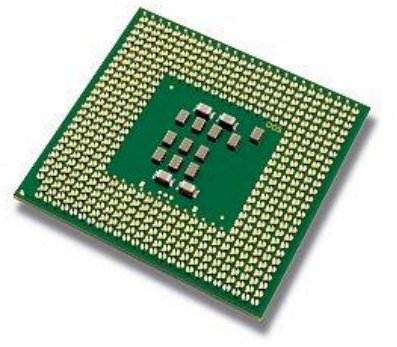
- Is it possible to execute the shellcode using only information available at the network level?
  - No access to the vulnerable host
- The execution of a polymorphic shellcode consists of two sequential parts
  1. Decryption
  2. Actual shellcode execution
- Focus on the decryption process
  - Generic, independent of the exploit/vulnerability/OS

# **Network-level Emulation** (2/2)

- Polymorphic shellcode engines (so far) produce *self-contained* decryptor code
  - **Position-independent:** will run from any location in the vulnerable process' address space
  - **Mandatory GetPC code:** for finding its absolute address in memory (x86 has no indirect memory addressing)
  - **Known operand values:** (encrypted payload size, decryption key, …) – revealed during execution
- Can be executed using merely a CPU emulator
  - Without any host-level information

# Detector

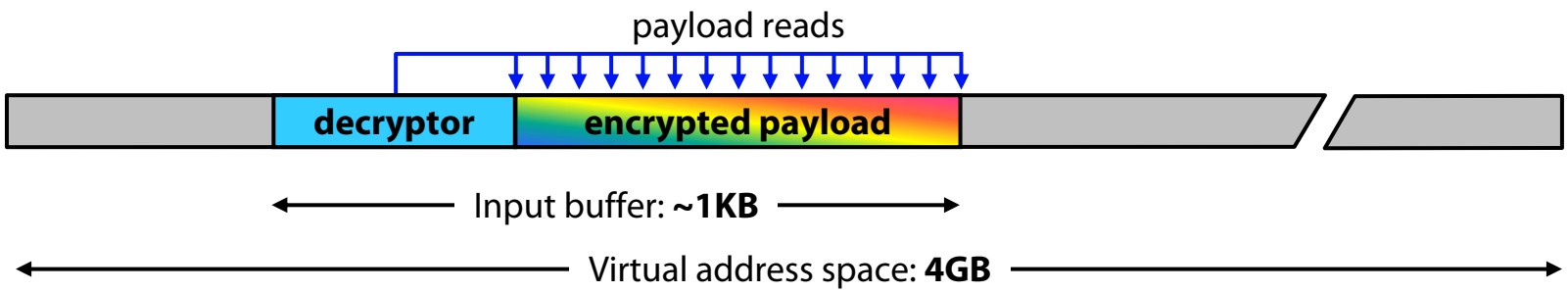
- Input: TCP streams or UDP packets
- CPU emulator
  - Randomized state before each new execution



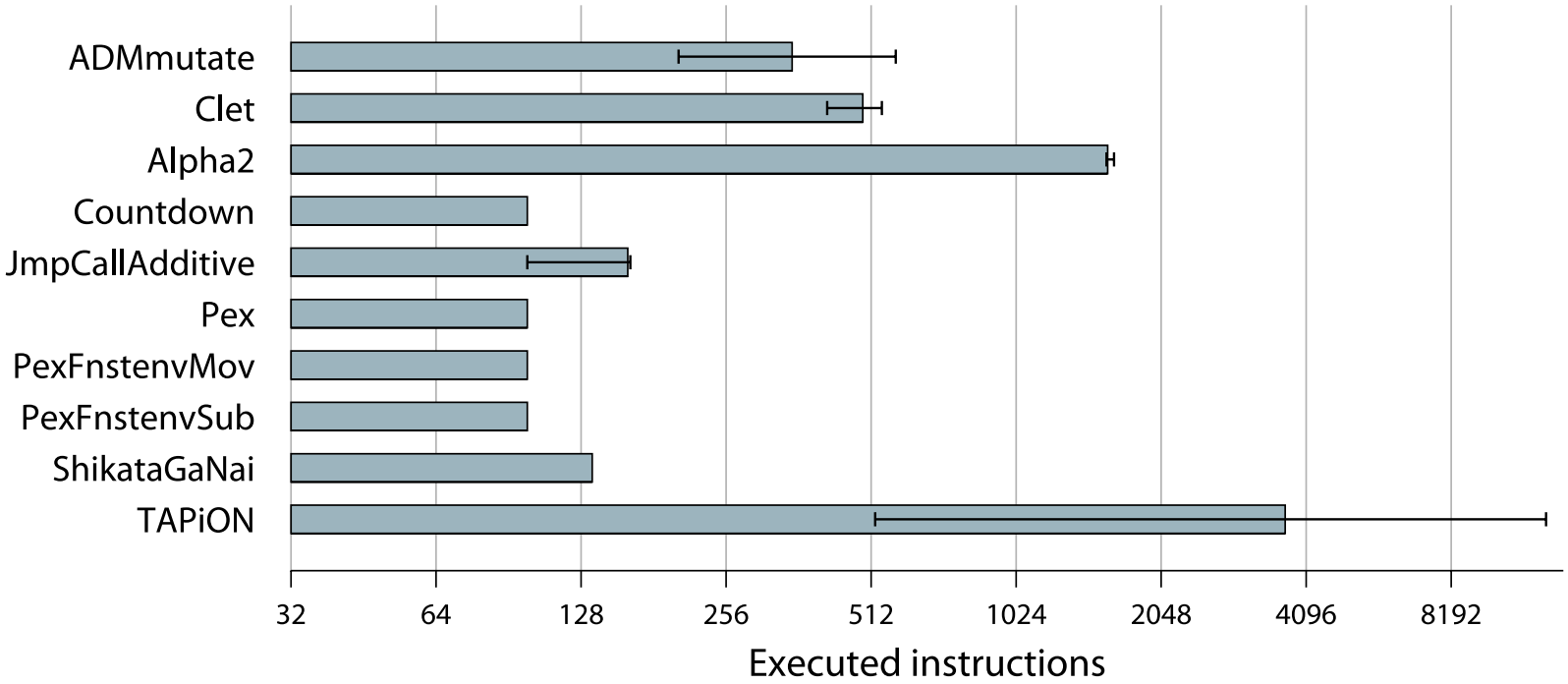
- We don't know the starting position of the shellcode in the input stream
  - Start execution from each byte
  - Performance optimization: skip NULL-byte-delimited regions smaller than 50 bytes
- Execution Threshold
  - Sometimes "*endless*" or *infinite* loops occur in random code
  - Dynamic detection and squashing of provably infinite loops

# Detection Heuristic

1. GetPC code (just a hint)
   - Execution of a suspicious instruction: `call, fstenv/fnstenv, fsave/fnsave`
2. Memory reads from *distinct* locations of the input buffer (*Payload reads)*
   - Low probability of payload reads in random data (~1KB vs 4GB)
   - Before each execution, the buffer is mapped to a random location



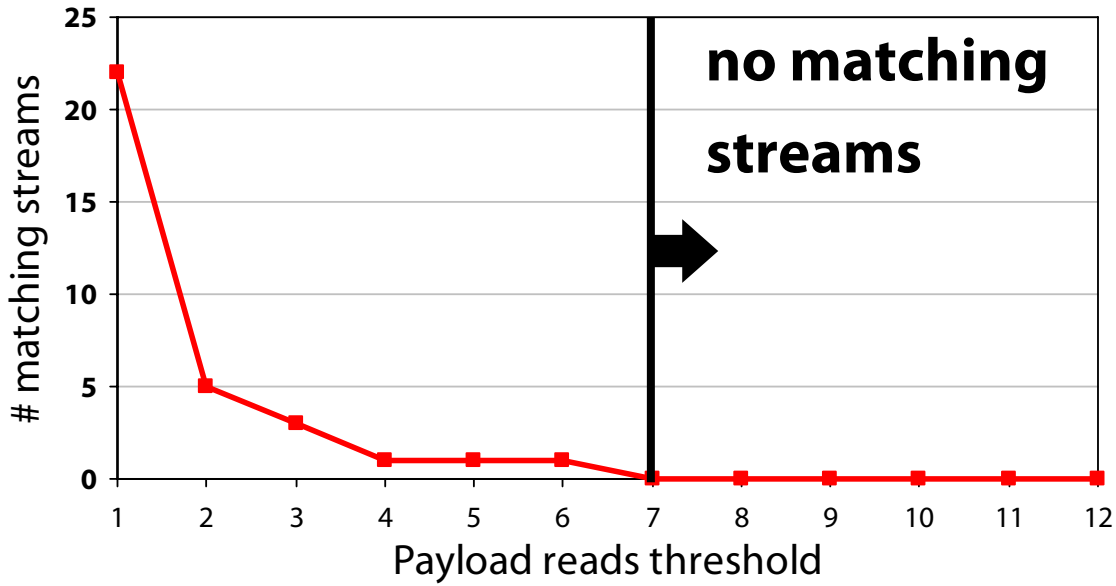
# Evaluation: Correct Execution




- Off-the-shelf polymorphic shellcode engines
- Original shellcode is 128 bytes, 1000 mutations with each engine
- ***In all cases the shellcode is decrypted correctly***

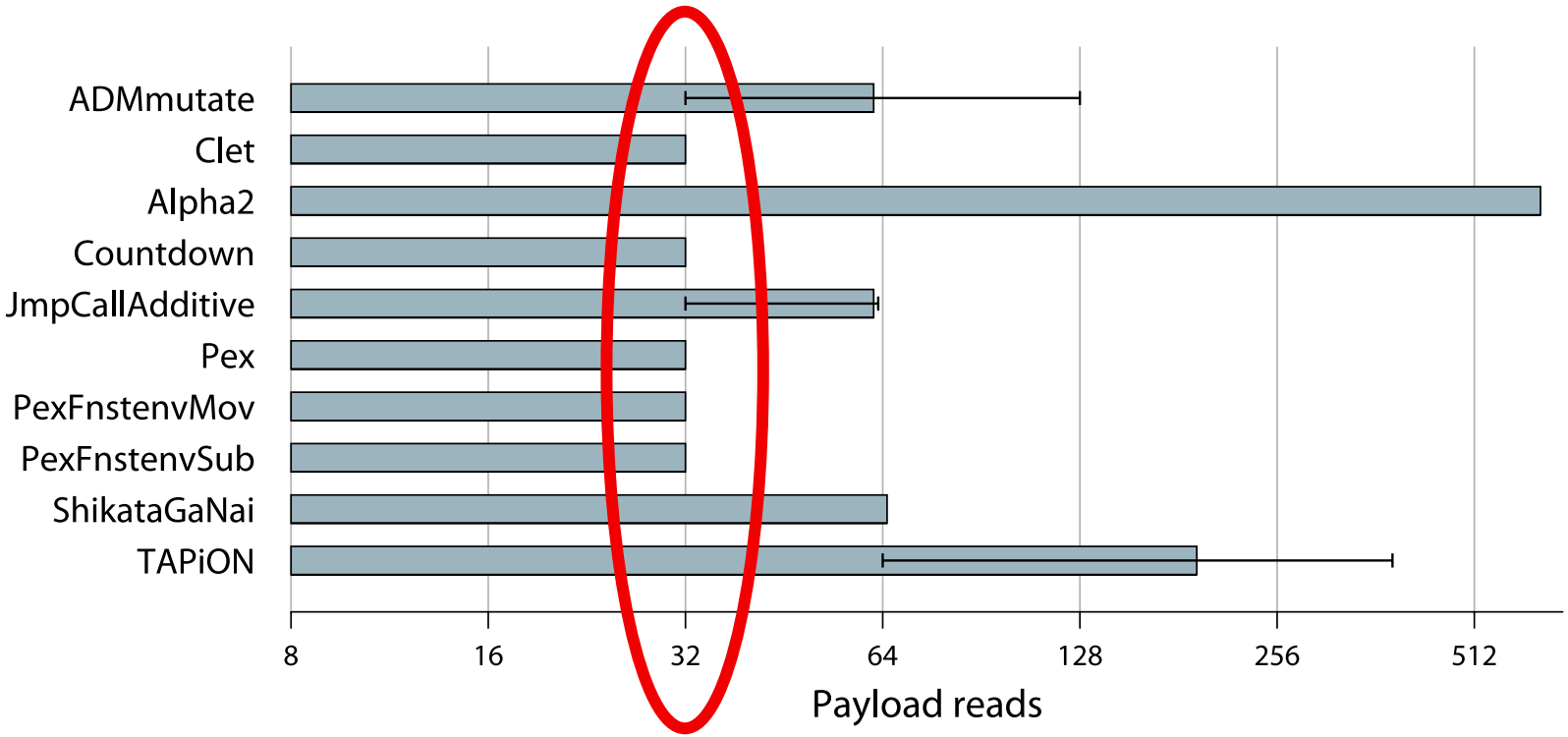
# Evaluation: False Positives / Heuristic Tuning

- Benign traffic traces and 61GB of random data
  - More than 2 million input streams



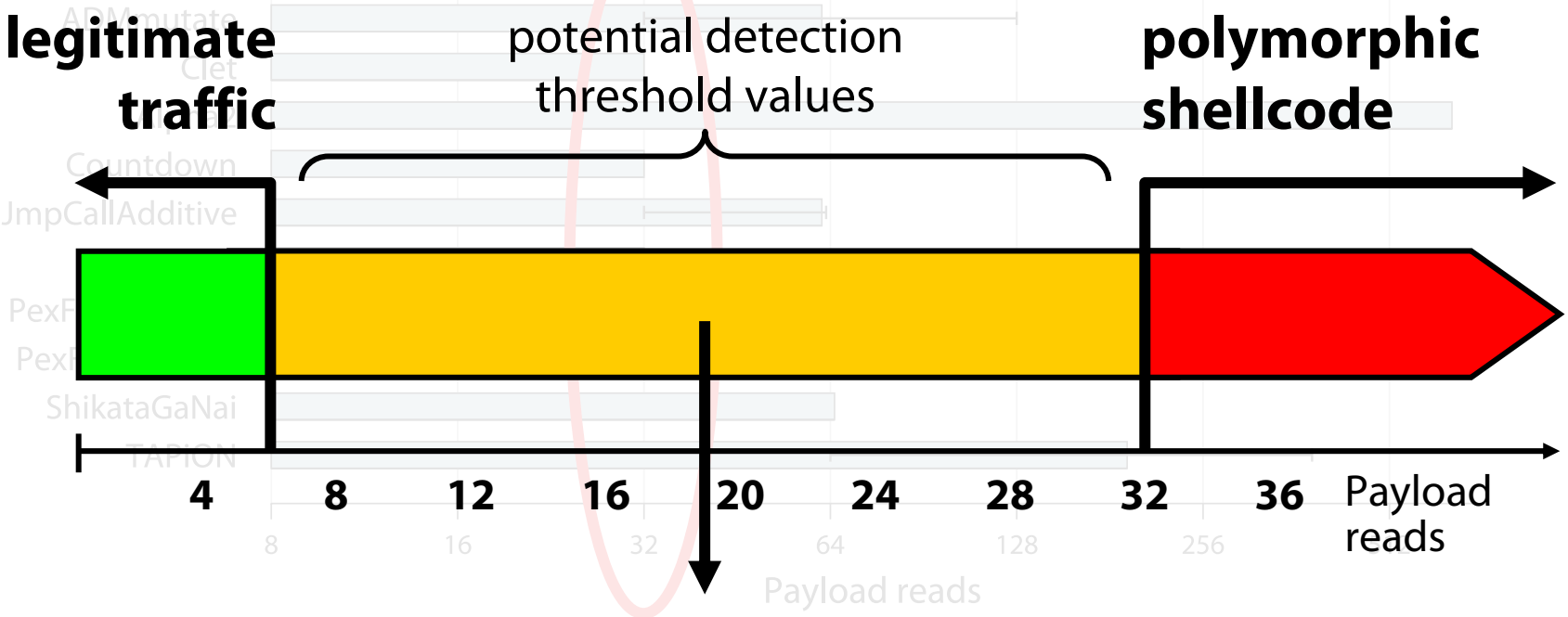

- Requiring the execution of some GetPC code followed by 7 or more payload reads gives **zero** false positives

# Payload Reads for Complete Decryption




- Benign data: 1-6 accidental payload reads in extremely rare cases
- Polymorphic shellcodes: **at least 32** payload reads for a conservatively small 128-byte shellcode

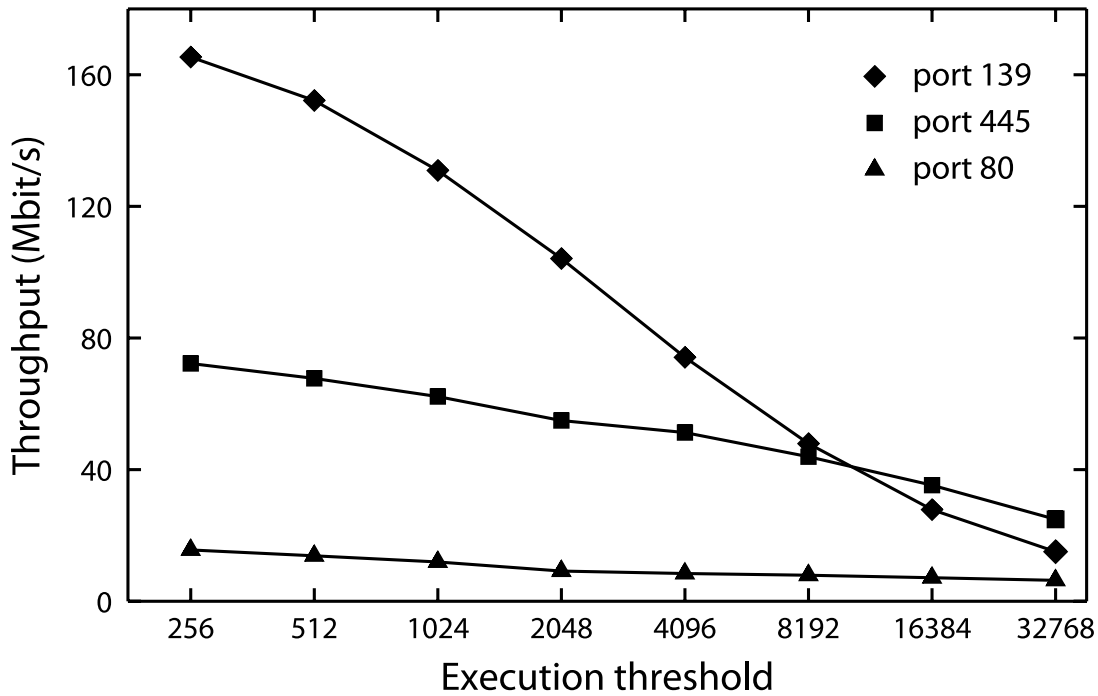
# Payload Reads for Complete Decryption




**Choose the median**

- Benign data: 1-6 accidental payload reads in extremely rare cases
- Polymorphic shellcodes: **at least 32** payload reads for a conservatively small 128-byte shellcode
- *Even more increased resilience to false positives*

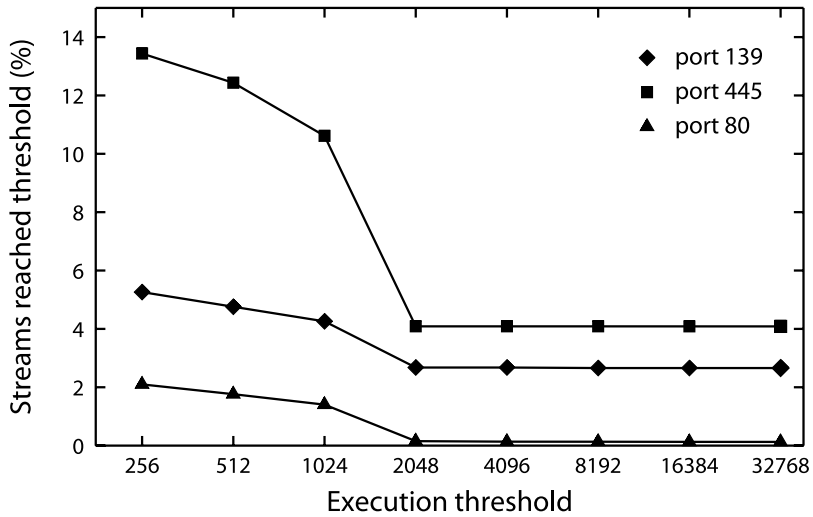
# Evaluation: Processing Cost




- Higher XT ➔ longer processing time per input ➔ lower throughput
- Incoming server traffic is usually much less than outgoing traffic
- NULL-byte optimization not effective for port 80 (mostly ASCII data)
  - Could take advantage of other delimiters (CRLF, protocol framing)

# Open Issues: "Endless" Loops

- Evasion by placing "endless" loops before/within the decryptor code
  - The execution threshold is reached before any sign of polymorphic behavior
- Endless loops occur in less than 5% of the benign traffic
  - Even if loops are used for evasion, useful as a first-level detector
  - Send all traffic reaching the exec threshold to a honeypot
- Squashing of provably infinite loops provides some mitigation

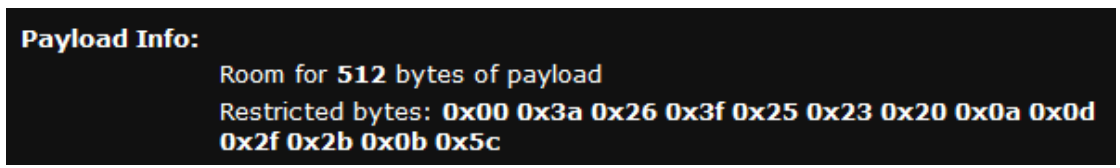



- Can we do better than this? Can we also skip the execution of seemingly "endless" (but not infinite) loops?
  - The loop can compute something useful, like the decryption key
  - Static analysis strikes back?

# Open Issues: Non-Polymorphic Shellcode

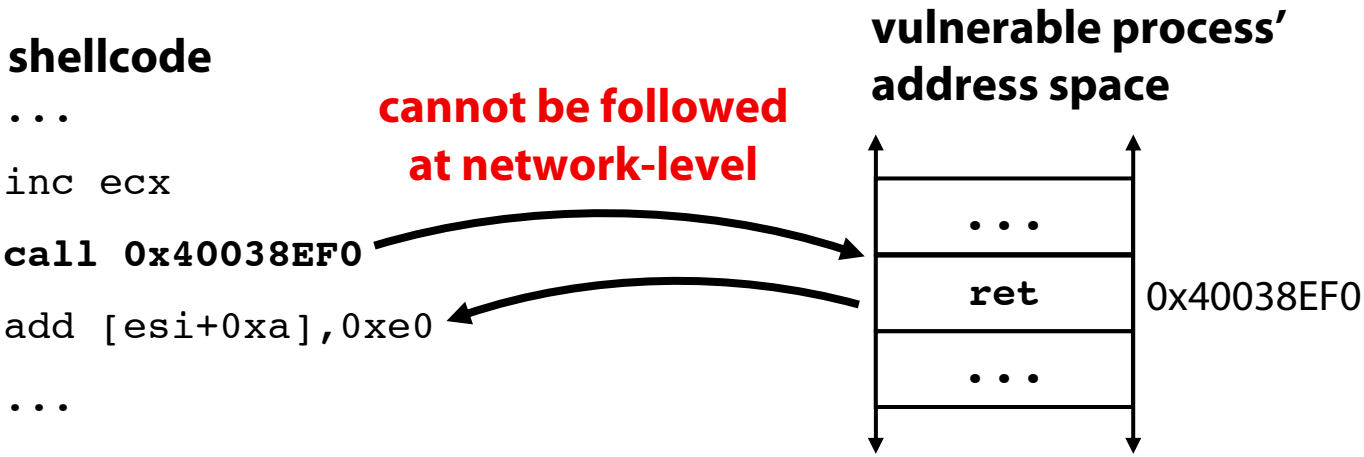
- What about plain or completely metamorphic code?
  - Does not decrypt its body
  - No self modifications
- Existing methods: search for exposed system calls, suspicious code sequences, …
- Shellcode "packing" is becoming essential
  - IDS Evasion!
  - Avoidance of restricted bytes

    http://www.metasploit.com/projects/Framework/exploits.html



# Open Issues: Non-Self-Contained Code

- Although current polymorphic/encryption engines produce self-contained code, non-self-contained code may be possible
- Take advantage of addresses with a priori known contents
  - e.g., initialize registers or jump to existing code
  - Should remain constant across all vulnerable systems (not always feasible)




- Augment the network-level detector with host-level information
  - e.g., invariant parts of the address space of each protected process

# Summary

- Pattern matching / static analysis not enough
  - Highly polymorphic and self-modifying code
- Network-level emulation
  - Detects self-modifying polymorphic shellcode
- Preliminary experimental results are promising
  - Accurate detection of shellcode from all known off-the-shelf polymorphic shellcode engines at 10-100 Mbps
- Open issues that have to be explored

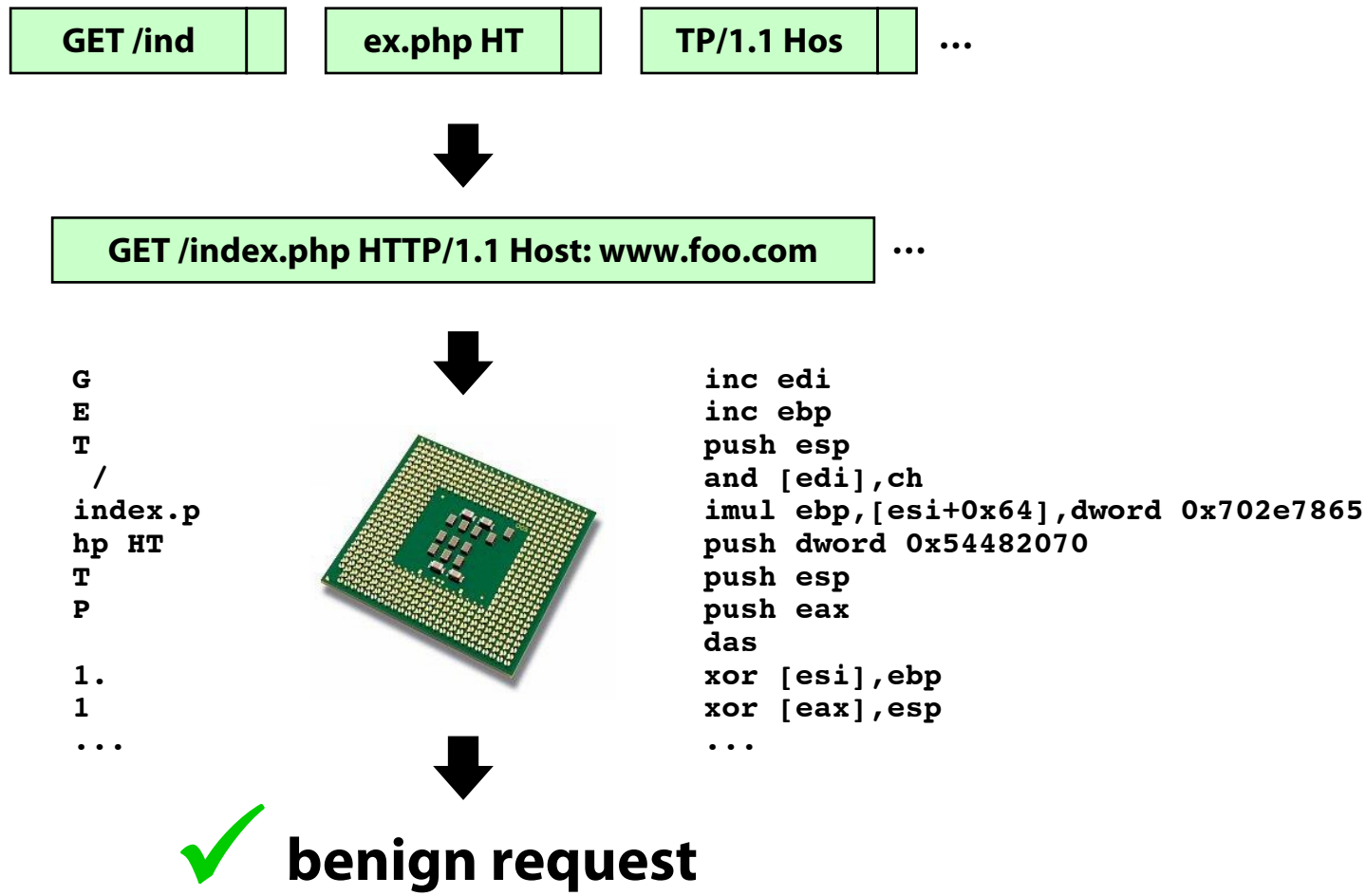
Network-level Polymorphic Shellcode Detection using Emulation

# thank you!

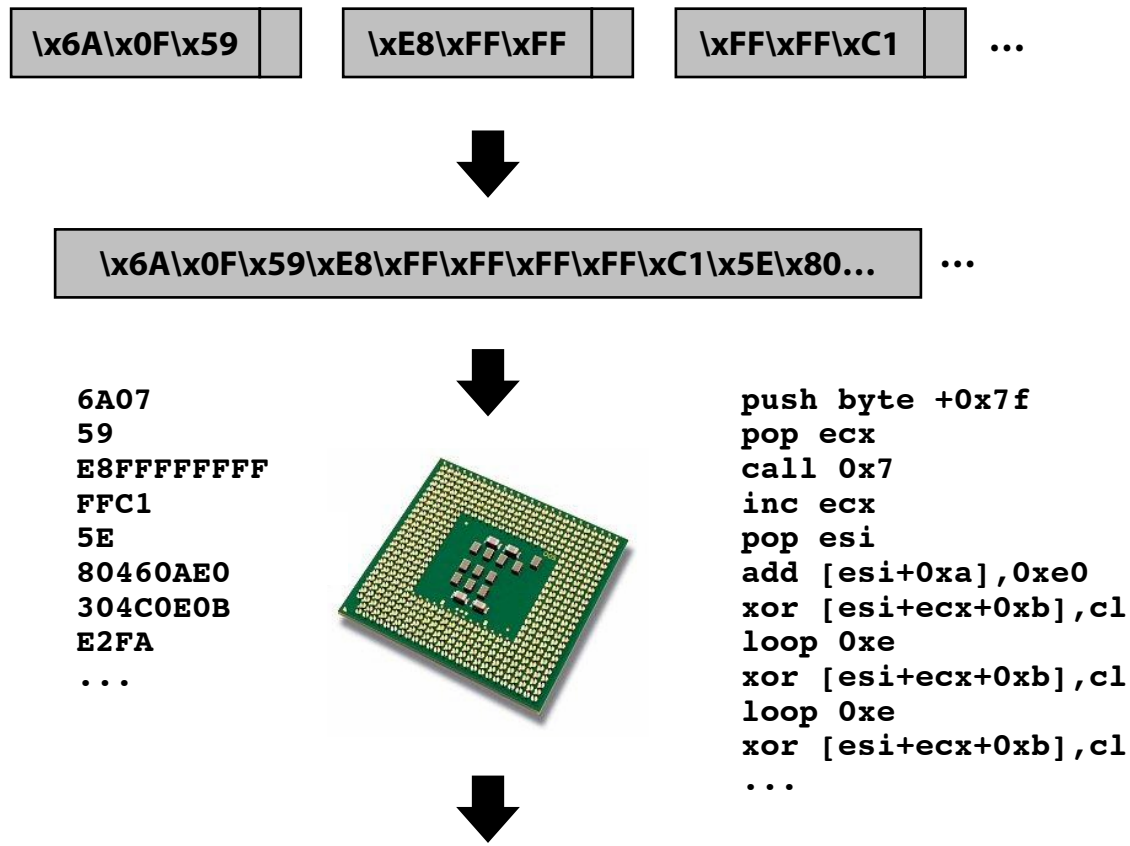
contact:
Michalis Polychronakis, mikepo@ics.forth.gr
Kostas Anagnostakis, kostas@i2r.a-star.edu.sg
Evangelos Markatos, markatos@ics.forth.gr

# fallback slides

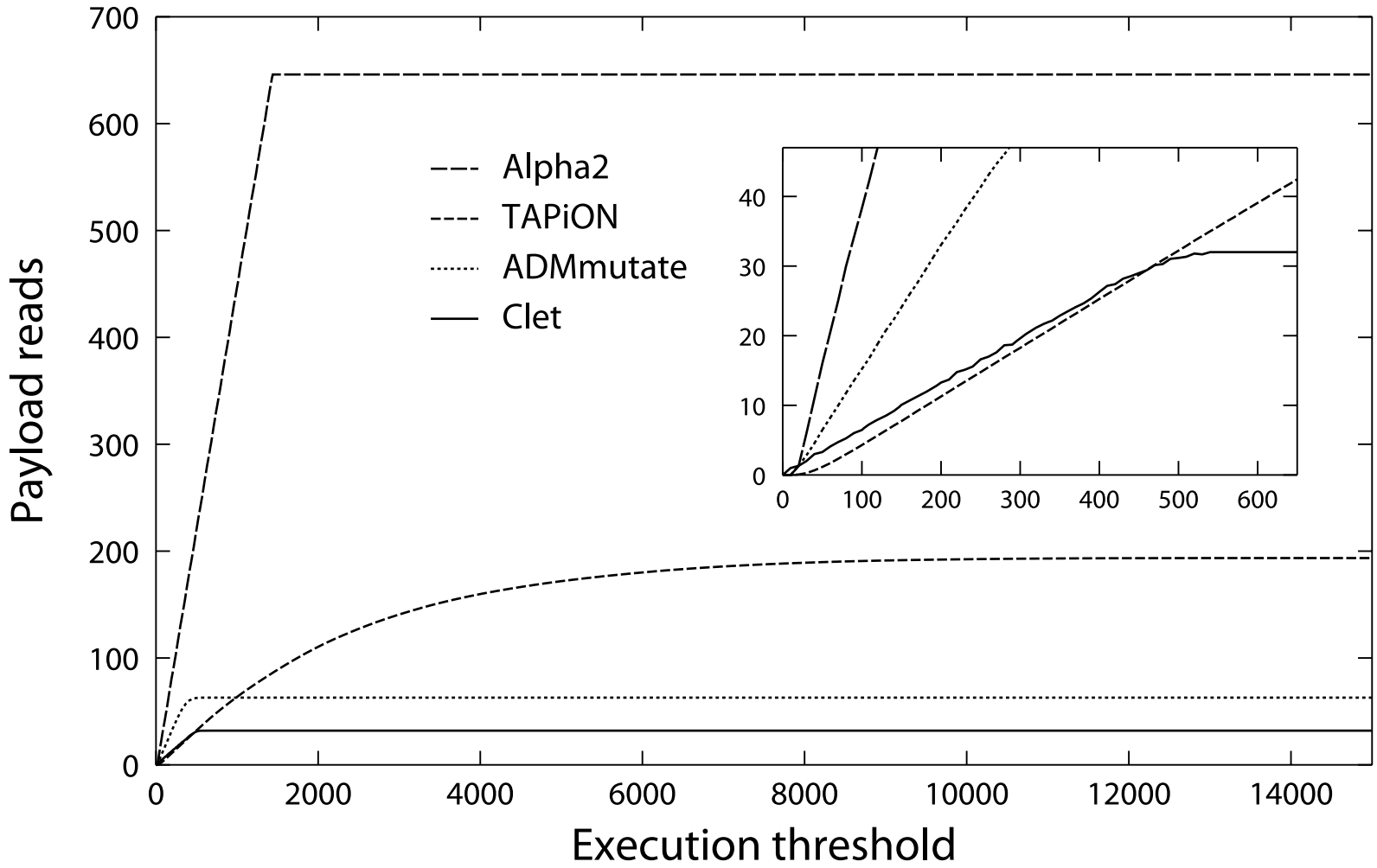
# Network-Level Emulation



# Network-Level Emulation



# # Payload Reads vs Execution Threshold



# Example Snort Signatures

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
(msg:"SHELLCODE Linux shellcode"; content:"|90 90 90 E8 C0 FF
FF FF|/bin/sh"; classtype:shellcode-detect; sid:652; rev:9;)
```

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
(msg:"SHELLCODE x86 setuid 0"; content:"|B0 17 CD 80|";
classtype:system-call-detect; sid:650; rev:8;)
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 10202:10203 (msg:"CA
license GCR overflow attempt"; flow:to_server,established;
content:"GCR NETWORK<"; depth:12; offset:3; nocase;
pcre:"/^\S{65}|\S+\s+\S{65}|\S+\s+\S+\s+\S{65}/Ri"; sid:3520;)
```