## Co899

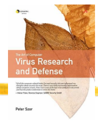
## Malware Recognition and Classification

"I think computer viruses should count as life. I think it says something about human nature that the only form of life we have created so far is purely destructive. We've created life in our own image"

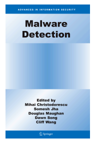
Stephen Hawking

## Resources

- Péter Ször, "The Art of Computer Virus Research and Defense", Symantec Press, 2005
  - Main Collection (QA 76.76.C68 szo)
- "A Survey of Automated Dynamic Malware Analysis Techniques and Tools", ACM Computing Surveys, 44(2), ACM Press, 2012
  - https://iseclab.org/papers/malware_survey.pdf
- Christodorescu, Jha, Maughan, Song and Wang, "Malware Detection", Springer, 2007
  - On-line access to e-book from Templeman








## Structure

- Malware self-protection strategies
- Hand-crafted signatures
- Signatures derived from binary code
- Signatures derived from traces

## Starting point

- Malicious software that deliberately fulfills the harmful intent of the attacker:
  - virus: requires a host to be run to be activated, and multiplies to form a new generation;
  - trojan: screen-saver, games, utility, but might download additional malware;
  - spyware: retrieves sensitive information such as contents of documents and e-mails;
  - rootkit: conceals processes, files or network connections (and its own presence) on an infected machine

## ps

- ps is a unix utility that lists active processes
- Suppose an open-source version of ps is modified to hide a particular process id (PID)

- Is this a trojan?
- Is this a rootkit?

## Malware ecosystem (community of organisms)

- A bot is machine that has been infected with malware so that it is remotely-controlled by a bot master
- Modern scenario:
  - Bot master rents botnet to a spammer
  - Spams contain link to infected webpage
  - Webpage installs spyware
  - Spyware collects online banking credentials
  - Credentials used to purchase goods on-line
- Torpig botnet consisted of 180K machines

## Anti-virus scanners (see list at virustotal.com)

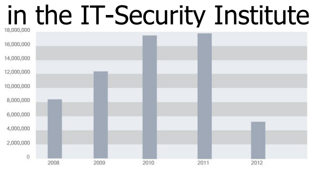
- Weapon of choice are the signature-based AV scanners
- Match pre-generated set of signatures against files of user
- Human analyst determines whether an unknown sample poses a threat
- If so, the security engineer attempts to find a signature which identifies the sample:
  - Generic enough to match variants
  - Specific enough not match legitimate content

## Limitations of signatures

- Signatures created by human analysts
  - Time-consuming and error-prone
- Not detect unknown threats (ground truth)
- Updated database needs to be deployed
- Vendors can release signatures that match legitimate executables (false positive)
  - Kaspersky released faulty signature that quarantined (or auto-deleted) Explorer

## Need for automation

- Growth in the IT-Security Institute collection:

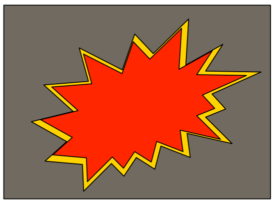


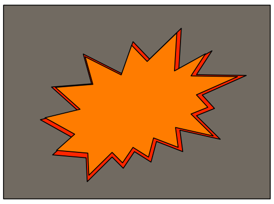

- Automation needed to quickly identify:
  - Samples that warrant manual analysis;
  - Samples that are variants of known threats.

## Different classification goals

- False positive = classified as malicious but benign
- False negative = classified as benign but malicious
- For triaging files for manual inspection:
  - False positive is annoying, but false negative is a disaster (miss unknown malicious file)
- For AV software:
  - False negative is annoying, but false positive is a disaster (quarantine or delete legitimate file)

## Malware classification schemes





- grey = space of all executables; red = malware
- yellow = malware over-approximation (no false negatives, ideal filter for security engineer)
- orange = malware under-approximation (no false positives, ideal AV engine for desktop)

## Dynamic (behavioral) analysis

- Information collected during execution:
  - system calls, network accesses,
  - file and memory modifications, etc
- Difficult to simulate conditions under which malicious activity is triggered
- Not clear how much time is required to observe malicious or key behavior
- Difficult to cover all paths through program

## Static analysis

- Do not need to execute file (no time issue)
- Potential for rapid classification
- Path coverage can be ensured by using compiler and testing techniques
- These techniques throw detailed information away to gain path coverage
- Difficult to derive detailed taints but call dependencies can be derived

## Part I
## Malware self-protection strategies

"If you know the enemy and know yourself, you need not fear the result of a hundred battles"
Sun Tzu

## Cascade.1701 decryptor

```
         lea     si, start          ; position to decrypt
         mov     sp, 0682h          ; length of virus body

decrypt: xor     [si], si           ; decrypt key 1
         xor     [si], sp           ; decrypt key 2
         inc     si                 ; loop running forward
         dec     sp                 ; sp for anti-debugging
         jnz     decrypt            ; loop until all bytes decrypted

start:   ...
```

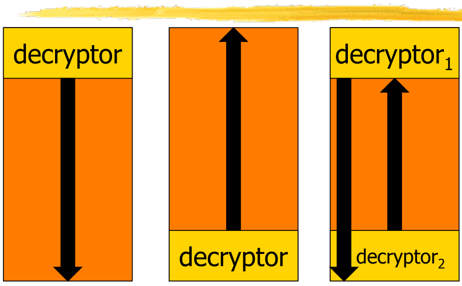
- Cryptographically weak
- Non-virus might use same encryptor

## xor encryption/decryption

Using repeating key 11110011:
- 11010011 10010101 10101111 10110111 $\rightarrow_e$
- 00100000 01100110 01011100 01000100 $\rightarrow_d$
- 11010011 10010101 10101111 10110111

- Using sliding key starting at 11110011:
  - 11010011 10010101 10101111 10110111 $\rightarrow_e$ (11110011 11110100 11110101 11110110)
  - 00100000 01100001 01011010 01000001 $\rightarrow_d$ (11110011 11110100 11110101 11110110)
  - 11010011 10010101 10101111 10110111

## Multi-layered decryption




W32/Harrier by TechnoRat

## O-ligo-morphic viruses

- Passing through a few changes of form
- Small set of decryptors
- Whale packaged with 12 predefined decryptors
- Memorial applied limited transformations:
  - Order mutations
  - Loop mutations
- Mutations delibrately rare (as in Badboy)

## Memorial (variant 1 of 96)

```
         mov    ebp, 00405000h      ; select base
         mov    ecx, 0550h          ; this many bytes
         lea    esi, [ebp+0000002E] ; offset of virus body
         add    ecx, [abp+00000029] ; plus this many bytes
         mov    al, [ebp+0000002D]  ; pick the first key

decrypt: nop              ; junk
         nop              ; junk
         xor    [esi],al  ; decrypt a byte
         inc    esi       ; next byte
         nop              ; junk
         inc    al        ; slide the key
         dec    ecx       ; any more bytes to decrypt?
         jnz    decrypt   ; until done
         jmp    start     ; execute body
```

## Memorial (variant 2 of 96)

```
         mov    ecx, 0550h          ; this many bytes
         mov    ebp, 013bc000h      ; select base
         lea    esi, [ebp+0000002E] ; offset of "start"
         add    ecx, [abp+00000029] ; plus this many bytes
         mov    al, [ebp+0000002D]  ; pick the first key

decrypt: nop              ; junk
         nop              ; junk
         xor    [esi],al  ; decrypt a byte
         inc    esi       ; next byte
         nop              ; junk
         inc    al        ; slide the key
         loop   decrypt   ; until done
                          ; mind the gap
         jmp    start     ; execute body
```

## Polymorphic viruses

- Generate many instances of mutators
- Mutators apply one or more of:
  - adding junk instructions (not necessarily nop)
  - adding random instructions after decryptor
  - permutating code blocks (skeleton changes)
  - Op code changes ie. exchanging xor eax, eax for
    - mov eax, 0 or
    - sub eax, eax
  - swapping registers
  - adding jump instructions

## 1260 (early example)

```
         inc    si       ; all junk marked in red
         mov    ax, 0e98 ; set key 1
         clc
         mov    di, 012a ; offset of start
         nop
         mov    cx, 0571 ; set key 2
decrypt: xor    [di], cx ; decrypt with key 2
         sub    bx, dx   ; junk
         xor    bx, cx   ; junk
         sub    bx, ax   ; junk
         sub    bx, cx   ; junk
         xor    dx, cx   ; junk
         xor    [di], ax ; decrypt with key 1
         inc    di       ; next byte
         clc             ; junk
         inc    ax       ; slide key 1
         loop   decrypt  ; slide key 2
```

- no repetitions in junk within a block

## W95/Marburg (later example)

```
routine6:        dec esi
                 ret
routine3:        esi, 439fe661h
                 ret
routine4:        xor [edi], 6f
                 ret
routine5:        add edi, 0001h
                 ret
decryptor_start: call routine1
                 call routine3
decrypt:         call routine4
                 call routine5
                 call routine6
                 cmp esi, 439fd271h
                 jnz decrypt
                 jmp start

routine1:        call routine2
routine2:        pop edi ; set edi to here
                 sub edi, 143ah
                 ret
```

- highly-structural
- decryptor split into islands of code
- islands reordered
- junk floating between islands
- BadBoy has 8 islands; Ghost has 10

## Metamorphic viruses

- Body-polymorphic rather than decryptor-polymorphic
- Virus body itself is mutated/not encrypted
- W32/Apparition which:
  - Carries its source
  - Drops the source when it finds a C compiler
  - Inserts into and removes junk code from source
  - Dangerous for Linux, where C compilers are commonly installed, even if not used

## Two generations of W95/Regswap

```
5a                   pop   edx
bf 04 00 00 00       mov   edi, 0004h
8b f5                mov   esi, ebp
b8 0c 00 00 00       mov   eax, 000ch
81 c2 88 00 00 00    add   edx, 0088h
8b 1a                mov   ebx, [edx]
89 9c 86 18 11 00 00 mov   [esi+eax*4+00001118], ebx

58                   pop   eax
bb 04 00 00 00       mov   ebx, 0004h
8b d5                mov   edx, ebp
bf 0c 00 00 00       mov   edi, 000ch
81 c0 88 00 00 00    add   eax, 0088h
8b 30                mov   esi, [eax]
89 b4 ba 18 11 00 00 mov   [edx+edi*4+00001118], esi
```

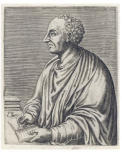
- edx→eax
- edi→ebx
- esi→edx
- eax→edi
- ebx→esi

## Mutation engines

- Hard to write polymorphic mutator, but MtE (Mutation Engine) can be linked against given:
  - registers not in use
  - desired size
  - length of virus body etc
- RPME (Real Permuting Mututor Engine) has been applied to create metamorphic viruses
- If a pattern of polymorphism can be detected, then any new virus is covered by detector
- Now hundreds of engines are known

## Part II

## Hand-crafted Signatures



"Greater is our terror of the unknown"
Titus Livius (59 BC – 17 AD)

## String scanning

- Signature = sequence of bytes (string) that is characteristic of the virus but not likely to be found in a benign executable
- Virus scanning engine searches predefined areas of files and the system, searching strings against its database of signatures
- Challenge is how to do this efficiently, say, no more than a second per file

## Your PC is now stoned!

- Early boot virus but with political variants
- Modifies boot sector, which is a region of a hard disk or floppy, that contains machine code to be loaded into RAM by boot process
- Machine code is usually, but not necessarily, the OS which is stored on the same device
- Angelina variant discovered on factory-sealed Seagate Technology 5850 hand drives
- A, B and C variants detected with signature

## Stoned signature (0400 b801 020e 07bb 0002 33c9 8bd1 419c)

```
be 04 00        mov si, 4
b8 01 02        mov ax, 201h
0e 07           push cs
bb 00 02        mov bx, 200h
33 c9           xor cx, cx
8b d1           mov dx, cx
41              inc cx
9c              pushf
2e ff 1e 09 00  call cs:9
73 0e           jnb fine
33 c0           xor ax, ax
9c              pushf
2e ff 1e 09 00  call cs:9
4e              dec si
75 e0           jnz next
eb 35           jmp giveup
```

## Searching for a signature

- First-generation scanners used Boyer-Moore
- Boyer-Moore algorithm is (counter-intuitively) faster with a longer signature (input pattern)
- Matches the tail of pattern first
- Skip across sections of file so sub-linear
- Scanning is typically I/O bound
- Many viruses prefix or postfix host files, so scanners use top-and-tail scanning on first 4K and last 4K

## Boyer-Moore

```
THIS IS A SIMPLE EXAMPLE        Match E against S
EXAMPLE                         Fail and S not in pattern
THIS IS A SIMPLE EXAMPLE        Thus shift pattern beyond S
       EXAMPLE                  Match E against P
THIS IS A SIMPLE EXAMPLE        Fail so shift to align with
         EXAMPLE                last P of pattern
THIS IS A SIMPLE EXAMPLE        Match E against E
         EXAMPLE                Success so match again
THIS IS A SIMPLE EXAMPLE        Match L against L
         EXAMPLE                Success so match again
THIS IS A SIMPLE EXAMPLE        Match P against P
         EXAMPLE                Success so match again
THIS IS A SIMPLE EXAMPLE        Match M against M
         EXAMPLE                Success so match again
THIS IS A SIMPLE EXAMPLE        Match A against I
         EXAMPLE                Fail and I not in pattern
```

## Boyer-Moore (cont')

```
THIS IS A SIMPLE EXAMPLE        Match E against X
            EXAMPLE             Fail so shift to align with
THIS IS A SIMPLE EXAMPLE        last X of pattern
                 EXAMPLE        Match E against E
THIS IS A SIMPLE EXAMPLE        Success so match again
                 EXAMPLE        Match L against L
THIS IS A SIMPLE EXAMPLE        Success so match again
                 EXAMPLE        Match P against P
THIS IS A SIMPLE EXAMPLE        Success so match again
                 EXAMPLE        Match M against M
THIS IS A SIMPLE EXAMPLE        Success so match again
                 EXAMPLE        Match A against A
THIS IS A SIMPLE EXAMPLE        Success so match again
                 EXAMPLE        Match X against X
THIS IS A SIMPLE EXAMPLE        Success so match again
                 EXAMPLE        Match E against E
```

## Second-generation scanners (type I)

- Virus mututor kits add:
  - junk,
  - NOPs,
  - reorder branches,
  - all of which change offsets
- So-called smart scanners strip out NOPs
- Construct signature from code that excludes relative branches and data offsets

## Second-generation scanners (type II)

- Kas-per-sky used two cryptographic checksums (normally used in transmission)
- Mathematical operations translates two sections of the file into two checksums
- First checksum over small range
- Second checksum over large range
- If first checksum recognised but not second, then a warning issued for a possible variant of known malicious code

## W95/Mad signature (896b 0a00 008a 8544 3040 0030 0747 e2fb eb09)

- Before generic decryptors, the code of the decryptors was often unique to malware
- Detect virus without decrypting it

```
89 00 40 30 45               mov edi, 00403045h
03 fd                        add edi, ebp                ; location of virus body
89 6b 0a 00 00               mov ecx, 0a6bh              ; length of virus body
8a 85 4 30 40 00             mov al, [ebp+40304000h]     ; read key
30 07             decrypt:   xor [edi], al
47                           inc edi                ; increment counter position
e2 fb                        loop decrypt           ; decrement ecx and loop until zero
eb 09                        jmp start              ; jump over one byte
78                key:       db 78h
...               start:     ...
```

## X-ray scanning

- Decrypts to n buffers using n algorithms
- Each buffer is searched for a signature
- SMEG (Simulated Metamorphic Encryption Generator) which decrypts using one of:
  - $d_i := e_i$ xor $k_i$ where $k_{i+1} = k_i + q$
  - $d_i := e_i - k_i$ and then $k_{i+1} = k_i + q$
  - $d_i := e_i$ xor $k_i$ and then $k_{i+1} = e_i + q$
  - $d_i := (-e_i)$ xor $k_i$ and then $k_{i+1} = k_i + q$
  - $d_i :=$ neg($e_i$ xor $k_i$) and then $k_{i+1} = k_i + q$
  - Where $k_i$ is a sliding key is q is the key shifter

## X-ray scanning

- Attempts n decryptions to n buffers
- Each buffer is searched for a signature
- SMEG (Simulated Metamorphic Encryption Generator) which applies one of:
  - $e_i := c_i$ xor $k_i$ and then $k_{i+1} = k_i + q$
  - $e_i :=$ neg($c_i$ xor $k_i$) and then $k_{i+1} = k_i + q$
  - $e_i := c_i$ add $k_i$ and then $k_{i+1} = k_i + q$
  - $e_i := c_i$ xor $k_i$ and then $k_{i+1} = e_i + q$
  - Where $k_1$ is the initial key is q is the key shifter

## Kaspersky versus Black Baron

- SMEG used by Pathogen and Queeg
- But decryptors are parametric in $k_1$ and q
- Yet $k_1$ can be derived from $e_1$
- Yet q can be derived from $e_2$ and $k_1$
- However, X-ray scanning cannot handle multiple layers of encryption
- X-ray scanning can categorise malware that have a bogus decryptor, provided that the virus body was correctly encrypted

## Code emulation

- Virtual machine is implemented to simulate CPU, flags, memory management, APIs, etc.
- Malicious code is simulated in VM; no malware is executed on CPU itself
- Emulator must mimic real system otherwise malware might detect simulator
- Viruses will eventually present themselves in VM's memory if emulator left for long enough

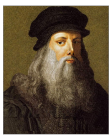
## Active memory and dirty pages (IBM AntiVirus)

- A decryptor will modify a page of memory and then execute instructions on that page
- Mark all pages clean
- Execute up to, say, $10^6$ instructions
  - Mark page as dirty if modified
  - if execute a dirty page then restart
- Will terminate when a dirty page within within $10^6$ instructions
- Scan dirty page for signatures

## Part III

## Signatures derived from binary code

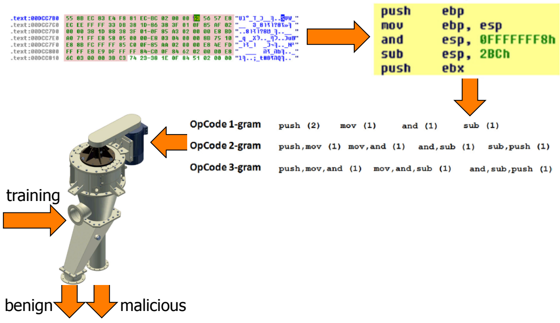
"Practice should always be based upon a sound knowledge of theory"
Leonardo da Vinci
1452-1519



## n-grams

- In computational linguistics, an n-gram is a contiguous sequence of n items from a given sequence of text or speech
- Example "to be or not to be"
  - uni-grams: to, be, or, not, to, be
  - bi-grams: to be, be or, or not, not to, to be
  - tri-grams: to be or, be or not, or not to, not to be

## Big picture




## Rationale for n-grams

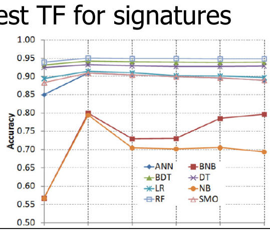
- Families of malware share common engine
- Common engine can be hidden by placing it in different locations in executable
- Locations may change between variants
- Disregard parameters (operands) of opcodes
- Gives more robust classifier

## Classification

- Signature is vector of n-grams recording relative frequency of each n-gram:
  - normalised term frequency (TF) = n-gram frequency / frequency of most frequent n-gram
- In training phase, a set of benign and malicious binaries is presented to system
- Use Kaspersky AV program as oracle for whether or not binaries are malicious
- Apply learning algorithm (SVMs, decision trees, neural networks etc)

## Best results

- Vocabularies of 515, 39K, 443K, 1769K and 5033K for 1-, 2-, 3-, 4- and 5-grams
- Need to reduce number of features
- Choose 1000 with highest TF for signatures
  - ANN = neural network
  - NB = naïve Bayes
  - BDT = decision tree
  - BNB = non-naïve Bayes
  - RF = random forest
  - LR =logistic regression




## Program-size/Kolmogorov complexity

- Kolmogorov complexity $C_L(s)$ of a finite string is equal to the length of the shortest program in language L that generates s
- The measure is parameterised by the underlying programming language
- If s1 = abcabcabcabc…abc where length(s1) = 196,609 then $C_{Java}(s1) \leq 170$
- If s2 = 139278124372921876…275 where length(s2) = 2412 then $C_{Java}(s2) \leq 272$

## 170 characters

```
public class ab
{   public static void main(String[] args)
    {   String s = "abc";
        for (int i = 0; i < 16; i++)
        {   s = s + s;   }
        System.out.println(s);
    }
}
```

## 275 characters

```
import java.math.*;
public class three
{   public static void main(String[] args)
    {   String s = "";
        BigInteger p = BigInteger.valueOf(1);
        for (int i = 0; i < 100; i++)
        {   s = s + p;
            p = p.multiply(BigInteger.valueOf(3));
        }
        System.out.println(s);
    }
}
```

## Kolmogorov complexity

- $C_L(s)$ is a measure of "complexity" of s:
  - simple string = short program
  - complicated string = long program
- $C_L(s) \leq |s| + c$
- $C_{JVM}(s) > C_{Java}(s)$ but $C_{JVM}(s) = C_{Java}(s) + d$ where d is a positive constant
- Invariant theorem for L1 and L2. There exists some constant e such that for all strings s:

$$C_{L1}(s) - e \leq C_{L2}(s) \leq C_{L1}(s) + e$$

## Normalised compression distance

- Given s1 and s2 NCD(s1, s2) = N/D where
  - $N = C_L(\text{append}(s1, s2)) - \min(C_L(s1), C_L(s2))$
  - $D = \max(C_L(s1), C_L(s2)$
- NCD(s1, s2) is a ratio with no units
- NCD(s1, s2) = NCD(s2, s1)

## Everything in common case

- If s1 = s2 then:
- $C_L(\text{append}(s1, s2)) = C_L(s1) + c$ for small c
- $\min(C_L(s1), C_L(s2)) = C_L(s1)$
- $\max(C_L(s1), C_L(s2)) = C_L(s1)$
- So N = c and $D = C_L(s1)$
- Therefore NCD(s1, s2) ≈ 0

## Nothing in common case

- If s1 and s2 have no common structure (analogous to no common subroutines)
- Suppose $C_L(s1) \leq C_L(s2)$ otherwise swap
- Then $C_L(\text{append}(s1, s2)) \approx C_L(s1) + C_L(s2)$
- $\min(C_L(s1), C_L(s2)) = C_L(s1)$
- $\max(C_L(s1), C_L(s2)) = C_L(s2)$
- $N \approx (C_L(s1) + C_L(s2)) - C_L(s1) = C_L(s2)$
- $D = C_L(s2)$
- NCD(s1, s2) ≈ 1

## Big problem?

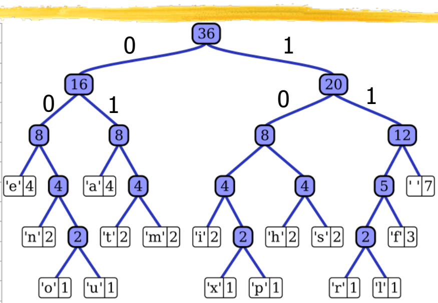
- But $C_L(s)$ is not a computable function
- For given L, there will *never* be any program which takes *s* as input and outputs $C_L(s)$
- Recall that zip is a lossless data compression program (based on Huffman encoding)
- Then $C_{Java}(s) \leq$|Huffman.java| where Huffman.java decompresses s from a binary sequence encoding s using a Huffman tree
- Huffman gives the optimal symbol-by-symbol (unrelated) coding and is in O(|s| log |s|)

## Huffman algorithm

- Create a leaf node for each character and add it to a priority queue
- While more than one node in the queue:
  - Remove the two nodes of highest priority (lowest frequency) from the queue
  - Create a new internal node whose priority is the sum of the two nodes' frequences
  - Add the new node to the queue
- Read off each path through tree in binary

## "this is an example of a huffman tree"

| | | |
|---|---|---|
| l | 1 | 11001 |
| o | 1 | 00110 |
| p | 1 | 10011 |
| r | 1 | 11000 |
| u | 1 | 00111 |
| x | 1 | 10010 |
| h | 2 | 1010 |
| i | 2 | 1000 |
| m | 2 | 0111 |
| n | 2 | 0010 |
| s | 2 | 1011 |
| t | 2 | 0110 |
| f | 3 | 1101 |
| a | 4 | 010 |
| e | 4 | 000 |
| space | 7 | 111 |



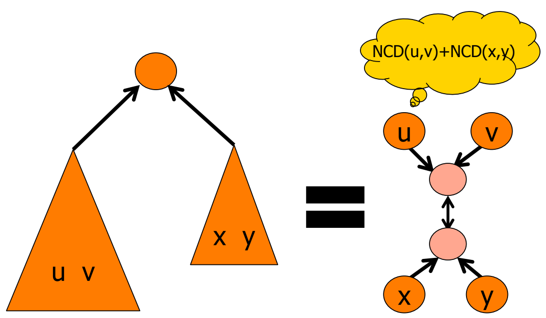

Gives string "0110,1010,1000,1011,..."

## NCD classification

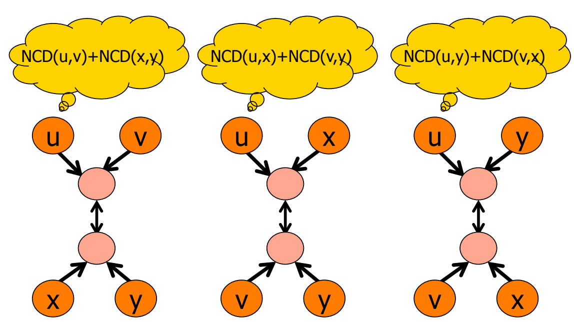
- $C_{Java}(s) \leq$|Huffman.java| ≈ |s.zip| for large s as then binary sequence dominates
- For clustering, compute

  $NCD_{zip}(s1, s2) = N/D$ where
  - N = |s1s2.zip| – min(|s1.zip|, |s2.zip|)
  - D = max(|s1.zip|, |s2.zip|)
- Compute $NCD_{zip}(s1, s2)$ to give a distance (adjacency) matrix for each pair of samples
- Fit a binary tree to the matrix

## Cost of a quartet (u,v,x and y)




## Cost of a quartet (u,v,x and y)

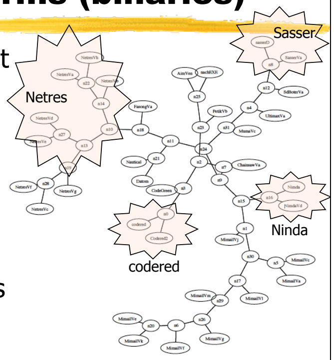


## Hill-climbing using quartets

- Let S denote set of malware sample
- Consider a binary tree T where each node is labeled with a sample from S
- Compute cost(u,v,x,y) for each {u,v,x,y} ⊆ S
- Define cost(T) as the sum of cost(u,v,x,y) for all the quartets {u,v,x,y} ⊆ S
- Want to find tree T with least cost(T)
- Mutate T (ie. swap two labels) to get T′ and check whether cost(T′) < cost(T)

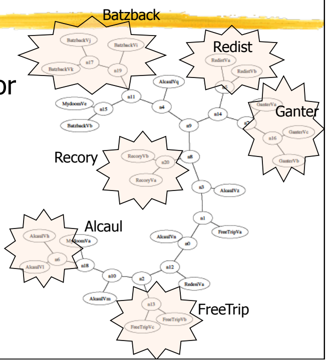
## Windows worms (binaries)

- Same families put together
- Also shown to separate shell scripts from executables
- Linux from Windows binaries




## UPX compression

- UPX uses fast in-place decompressor of < 200 bytes
- A compressed string is hard to compress further
- But still group into families as UCL algorithm not as strong as Huffman




## Binary differencing

- Distance is a norm (number) versus a difference that indicates structural/ behavioral distinctions
- In a security update:
  - the security analyst must rapidly identify what is old from what is new (different)
- In malware analysis:
  - The engineer must find which parts of a sample are different from known malware

## Sources of differences

- Different compilers/compiler versions/ optimisation levels
- Addition/removal of functions/inlining
- NOP blocks for functional alignment
- Instruction reordering/register allocation
- Diversifying transforms (copyright protection)
- Willful obfuscation in order to bypass detection by signature engines:
  - rewriting/junk code/call-stack tampering

## BinDiff

- Developed at Zynamics by:
  - Thomas Dullien/Halvar Flake
  - Rolf Rolles
- Performs structural matching:
  - Call graph
  - Control-flow graph
- Compares functions/blocks using:
  - attributes
  - selectors

## BinDiff attributes

- By default, three attributes $\varepsilon$, $\rho$, $\beta$ are used for matching functions:
  - $\varepsilon$ is the number of edges between blocks in the function
  - $\rho$ is the number of returns in the function
  - $\beta$ is the number of basic blocks that make up the function
- Other attributes can also be used:
  - checksum of instructions that encode function
  - function symbol names (if available)

## BinDiff signatures

- The tuple, s = $(\varepsilon, \rho, \beta)$, is used as a signature for a function f
- Different functions can same the same signature
- Two (unmatched) functions, $f_1$ and $f_2$, in binaries, $b_1$ and $b_2$, are matched if:
  - $f_1$ has a unique signature s in $b_1$
  - $f_2$ has a unique signature s in $b_2$
- Matching is improved by (unmatched) parents and children of matched functions (selectors)

## BinDiff algorithm

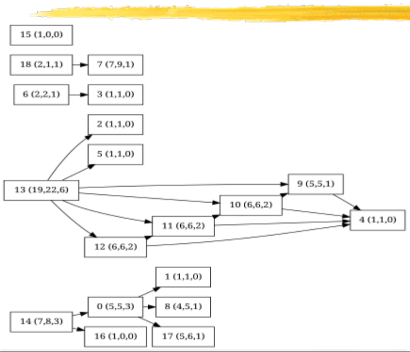
function bindiff($G_1$, $G_2$);
    M := Ø;    // $M \subseteq \{ (f1, f2) \mid f_1 \in G_1$ and $f_2 \in G_2 \}$
    $S_1 := G_1$;  // unmatched functions
    $S_2 := G_2$;  // unmatched functions
    $(M', S_1', S_2')$ := firstMatches($S_1$, $S_2$); // $S_1' \subseteq S_1$ and $S_2' \subseteq S_2$
    while $M \neq M'$ do
        $(M, S_1, S_2) := (M', S_1', S_2')$
        $(M', S_1', S_2')$ := furtherMatches(M, $S_1$, $S_2$); // $M \subseteq M'$
    end
return M;

## BinDiff example

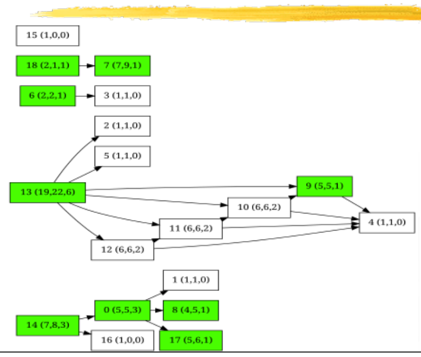
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void D() {
        printf("\n");
        return;
}

void C() {
        D();
        printf("\n");
}

void B() {
        C();
        printf("\n");
}
```

```
void B() {
        C();
        printf("\n");
}

void A() {
        B();
        printf("\n");
}

int main() {
        srand(45);
        A();
        B();
        C();
        D();
        rand();
}
```

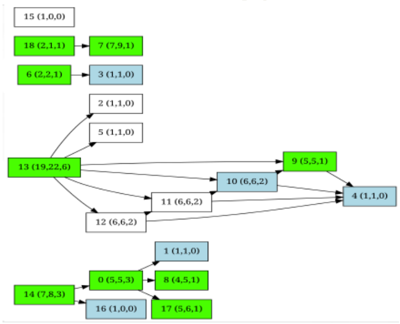
## Recovered call graph



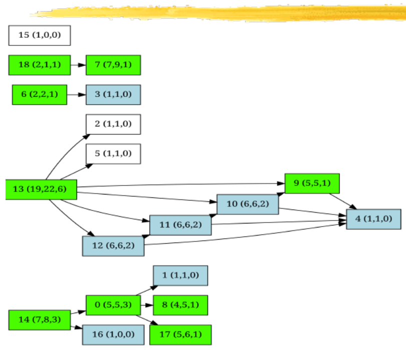

## Initial matches (against itself)



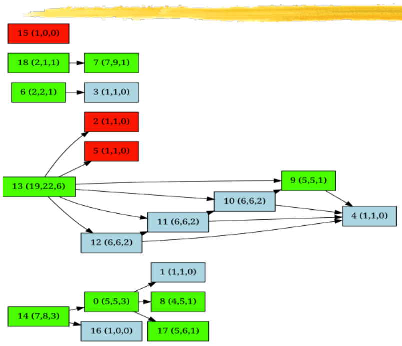
## Using selectors (1st iteration of loop)




## Using selectors (2nd and 3rd iterations of loop)




## Unresolved matches




## Postscript: BinDiff for distance

- Can transform $G_1$ into $G_2$ by series of edit operations where an edit operation is one of:
  - Vertex/edge insertion
  - Vertex/edge deletion
  - Vertex/edge substitution
- A sequence of operations form an edit path
- There are many edit paths from $G_1$ into $G_2$
- Graph edit distance, GED($G_1$, $G_2$), is the length of the shortest path between $G_1$ and $G_2$
- Finding GED($G_1$, $G_2$) is NP-hard

## Obfuscation I: opaque constants

- In a binary/executable constants arise as:
  - Targets of conditional and unconditional jumps
  - Locations in memory which store values
  - Numbers in numeric expressions
- Obfuscate control-flow with:

```
char c = 57, d = 57; // assignment obfuscated
if (c == d)          // opaque predicate
      normal code
else
      dead code which spurious calls
```

## Opaque constants [Moser et al, ACSAC, 2007]

- Recall 57 = 0b00111001
- Partition the 8 bits into 2 groups, say LMLLMLML, where L = loop and M = mask
- Loop code section generates 0b0*11*0*1
- Mask code section generates 0b*0**1*0*
- Put char a[7] = {0b01111001,...,0b00111011};
- Put char b[7] = {0b00111011,...,0b01110001};
- By replacing * in 0b0*11*0*1 with random bits

## Opaque constant calculation for 0b00111001

```
char a[7] = {0b01111001, ..., 0b00111011};
char b[7] = {0b00111011, ..., 0b01110001};
char unknown = load_from_wierd_address;
char c = 0;
for (i = 0; i < 7; i++)
{
    if (bit_set(unknown, i)) c = c ^ a[i];
    else c = c ^ b[i];
}
c = c  | 0b00001000      // set the single 1
c = c & 0b10111101      // set the two 0's
```

## Part IV

## Signatures derived from traces

## System calls

- Only way a user-mode program (application) can interact with its environment:
  - system call used to create a file
  - All subsequent interaction through a handle
  - calls execute in kernel (privileged O/S) mode
  - malware must likewise communicate with environment through system call interface
- System calls are behavioral abstractions for:
  - networking, system and file management, etc

## Hooking

- The process of intercepting function calls is called hooking
- Hook function is responsible for recording invocation in log and analysing parameters
- -finstrument-functions used in GCC
- Binary rewriting can modify all call sites
- Dynamic shared libraries can be renamed and replaced with stubs libraries containing hooks
- Call traces converted to graphs for analysis
- Signature is set of sub-graph of calls

## Taint Analysis

- Monitoring sensitive data to refine graphs
- Taint source introduces taints:
  - system call to time to trigger bomb (Michelangelo)
  - Different taints used for different sources
- Sink reacts when stimulated with tainted data:
  - warn when taint transmitted over network in log
- Policies need to propagate taints over:
  - assignments;
  - address dependencies;
  - control-flow dependencies

## Propagating taints

- Data dependencies where x is tainted:
  - mov eax, x          // a = x
- Address dependencies:
  - mov eax, [x + 10]          // a = x[10]
- Control-flow dependencies:
  - if (x == 0) v = 0 else v = 1
  - all assignments in both branched tainted until confluence point when branches recombine
- Combining taints or precedence resolution:
  - add eax, ebx          // both registers tainted

## Obfuscation II: logic bombs

- A trigger that hides malicious intent ie:
  - until k keys have been pressed;
  - activated by a command from a bot;
  - classically annually on the $6^{th}$ March
- A (blind) trace is unlikely to trigger the bomb
- In the case of a remotely activated bomb:
  - bot command might even be an encryption key that hides the malicious code
  - hash of key used in trigger condition to hide key