# Statistical Structures: Fingerprinting Malware for Classification and Analysis

Daniel Bilar Wellesley College (Wellesley, MA) Colby College (Waterville, ME)

bilar <at> alum dot dartmouth dot org



# Why Structural Fingerprinting?

Goal: Identifying and classifying malware

**Problem:** For any single fingerprint, balance between over-fitting (type II error) and underfitting (type I error) hard to achieve

**Approach:** View binaries simultaneously from different structural perspectives and perform statistical analysis on these 'structural fingerprints'



#### **Different Perspectives**

**Idea:** Multiple perspectives may increase likelihood of correct identification and classification

Structural Perspective	Description	Statistical Fingerprint	static / dynamic?
Assembly instruction	Count different instructions	Opcode frequency distribution	Primarily static
Win 32 API call	Observe API calls made	API call vector	Primarily dynamic
System Dependence Graph	Explore graph- modeled control and data dependencies	Graph structural properties	Primarily static



# Fingerprint: Opcode frequency distribution

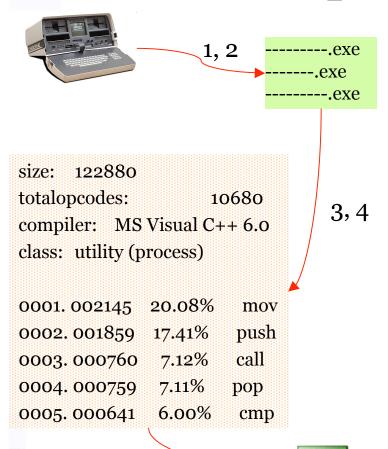
**Synopsis:** Statically disassemble the binary, tabulate the opcode frequencies and construct a statistical fingerprint with a subset of said opcodes.

**Goal:** Compare opcode fingerprint across non-malicious software and malware classes for quick identification and classification purposes.

Main result: 'Rare' opcodes explain more data variation then common ones



### Goodware: Opcode Distribution

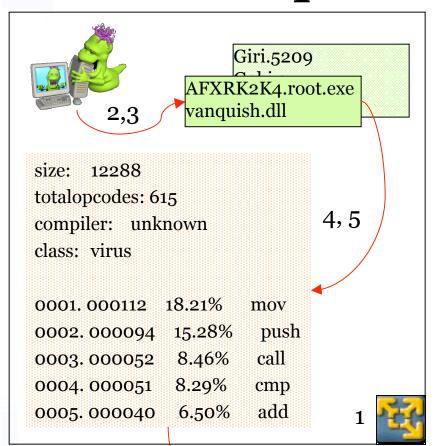


5

#### Procedure:

- 1. Inventoried PEs (EXE, DLL, etc) on XP box with Advanced Disk Catalog
- 2. Chose random EXE samples with MS Excel and Index your Files
- 3. Ran IDA with modified InstructionCounter plugin on sample PEs
- 4. Augmented IDA output files with PEID results (compiler) and general 'functionality class' (e.g. file utility, IDE, network utility, etc)
- 5. Wrote Java parser for raw data files and fed JAMA'ed matrix into Excel for analysis

## Malware: Opcode Distribution



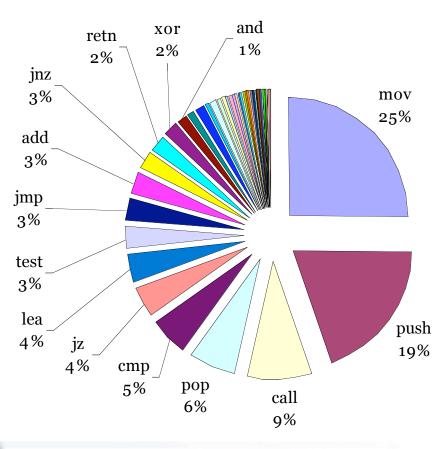
#### Procedure:

- 1. Booted VMPlayer with XP image
- 2. Inventoried PEs from C. Ries malware collection with Advanced Disk Catalog
- 3. Fixed 7 classes (e.g. virus,, rootkit, etc), chose random PEs samples with MS Excel and Index your Files
- 4. Ran IDA with modified InstructionCounter plugin on sample PEs
- 5. Augmented IDA output files with PEID results (compiler, packer) and 'class'
- 6. Wrote Java parser for raw data files and fed JAMA'ed matrix into Excel for analysis

6



### Aggregate (Goodware): Opcode Breakdown



#### **20 EXEs**

(size-blocked random samples from 538 inventoried EXEs)

~1,520,000 opcodes read

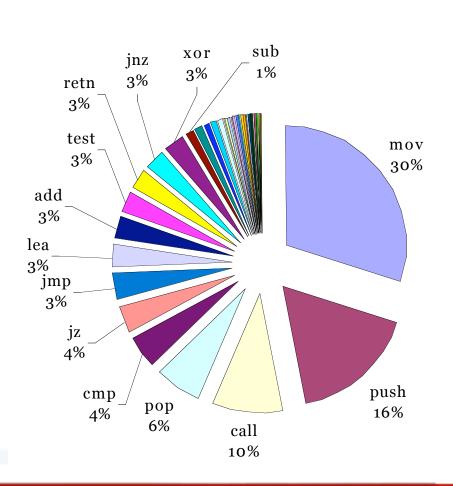
**192 out of 398** possible opcodes **found** 

**72 opcodes** in pie chart account for >99.8%

14 opcodes labelled account for ~90%

**Top 5 opcodes** account for ~**64** %

# Aggregate (Malware): Opcode Breakdown



67 67 PEs

(cl (class-blocked random samples from 250 inventoried PEs)

~665,000 opcodes read

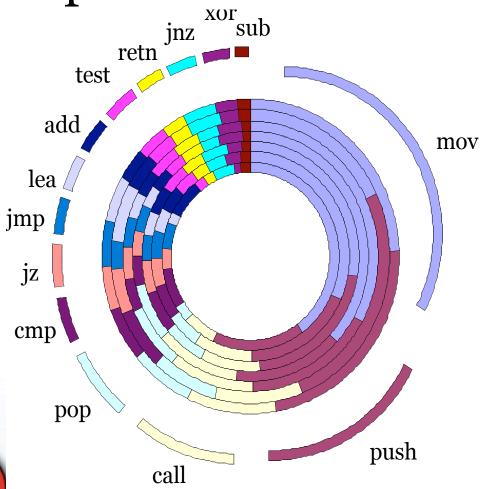
14 141 out of 398 possible op opcodes found (two undocumented)

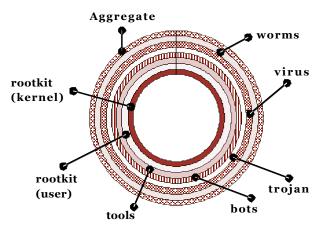
60 opcodes in pie chart ac account for >99.8%

14 opcodes labelled account for >92%

Top 5 opcodes account for ~65%

# Class-blocked (Malware): Opcode Breakdown Comparison





Aggre	egate	breakd	<u>own</u>
mov	30 %	lea	3%
push	16 %	add	3%
call	10 %	test	3%
pop	6 %	retn	3%
cmp	4 %	jnz	2%
jz	4 %	xor	2%
jmp	4 %	sub	1%

### Top 14 Opcodes: Frequency

Opcode	Goodware	Kernel RK	User RK	Tools	Bot	Trojan	Virus V	Worms
mov	25.3%	37.0%	29.0%	25.4%	34.6%	30.5%	16.1%	22.2%
push	19.5%	15.6%	16.6%	19.0%	14.1%	15.4%	22.7%	20.7%
call	8.7%	5.5%	8.9%	8.2%	11.0%	10.0%	9.1%	8.7%
pop	6.3%	2.7%	5.1%	5.9%	6.8%	7.3%	7.0%	6.2%
cmp	5.1%	6.4%	4.9%	5.3%	3.6%	3.6%	5.9%	5.0%
jz	4.3%	3.3%	3.9%	4.3%	3.3%	3.5%	4.4%	4.0%
lea	3.9%	1.8%	3.3%	3.1%	2.6%	2.7%	5.5%	4.2%
test	3.2%	1.8%	3.2%	3.7%	2.6%	3.4%	3.1%	3.0%
jmp	3.0%	4.1%	3.8%	3.4%	3.0%	3.4%	2.7%	4.5%
add	3.0%	5.8%	3.7%	3.4%	2.5%	3.0%	3.5%	3.0%
jnz	2.6%	3.7%	3.1%	3.4%	2.2%	2.6%	3.2%	3.2%
retn	2.2%	1.7%	2.3%	2.9%	3.0%	3.2%	2.0%	2.3%
xor	1.9%	1.1%	2.3%	2.1%	3.2%	2.7%	2.1%	2.3%
and	1.3%	1.5%	1.0%	1.3%	0.5%	0.6%	1.5%	1.6%

### Comparison Opcode Frequencies

Opcode	Goodware
mov	25.3%
push	19.5%
call	8.7%
pop	6.3%
cmp	5.1%
jz	4.3%
lea	3.9%
test	3.2%
jmp	3.0%
add	3.0%
jnz	2.6%
retn	2.2%
xor	1.9%
and	1.3%

Perform distribution tests for top 14 opcodes on 7 classes of malware:

Rootkit (kernel + user)
Virus and Worms
Trojan and Tools
Bots

Investigate: Which, if any, opcode frequency is significantly different for malware?

#### Top 14 Opcode Testing (z-scores)

Opcode	Kernel RK	User RK	Tools	Bot	Trojan	Virus	Worms
mov	36.8	20.6	2.0	70.1	28.7	-27.9	-20.1
push	-15.5	-21.0	4.6	-59.9	-31.2	12.1	6.9
call	-17.0	1.2	5.2	26.0	10.6	2.6	-0.3
pop	-22.0	-13.5	4.9	5.1	9.8	4.8	-1.1
cmp	7.4	-3.5	-0.6	-30.8	-21.2	4.7	-1.8
jz	-7.4	-6.1	0.9	-20.9	-11.0	1.4	-4.4
lea	-16.2	-8.4	10.9	-29.2	-18.3	11.5	4.2
test	-12.2	0.0	-6.6	-14.6	1.8	-0.2	-3.4
jmp	8.5	11.7	-5.0	-2.2	5.0	-2.3	20.4
add	22.9	10.8	-6.4	-13.5	-0.1	4.3	0.5
jnz	8.7	7.4	-11.7	-12.2	-0.9	5.3	8.0
retn	-5.5	2.5	-12.3	18.4	17.8	-1.4	2.6
xor	-8.9	6.7	-2.6	29.5	15.3	2.7	7.7
and	1.9	-7.3	-0.7	-33.6	-17.0	2.4	5.9

Higher
High
Similar
Low
Lower

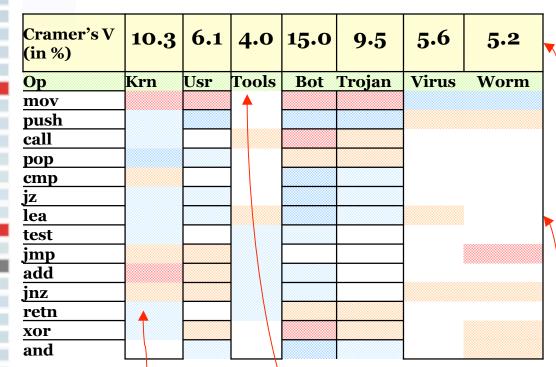
Opcode Frequency

Tests
suggests
opcode
frequency
roughly

1/3 same 1/3 lower 1/3 higher

vs goodware

#### Top 14 Opcodes Results Interpretation



**Kernel-mode Rootkit:** 

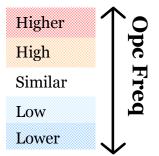
most # of deviations

→ handcoded assembly; 'evasive' opcodes?

**Tools:** (almost) no deviation in top 5 opcodes → more 'benign' (i.e. similar to goodware)?

Most frequent 14 opcodes **weak predictor** 

Explains just 5-15% of variation!



#### Virus + Worms:

few # of deviations; more jumps → smaller size, simpler malicious function, more control flow?

#### Rare 14 Opcodes (parts per million)

Opcode	Goodware	Kernel RK	User RK	Tools	Bot	Trojan	Virus	Worms
bt	30	0	34	47	70	83	0	118
fdivp	37	o	o	35	<b>52</b>	52	o	59
fild	357	О	45	O	133	115	0	438
fstcw	11	O	0	O	22	21	O	12
imul	1182	1629	1849	708	726	406	755	1126
int	25	4028	981	921	O	O	108	O
nop	216	136	101	71	7	42	647	83
pushf	116	O	11	<b>59</b>	0	O	54	12
rdtsc	12	o	O	O	11	o	108	0
sbb	1078	588	1330	1523	431	458	1133	782
setb	6	o	68	12	22	52	0	24
setle	20	o	0	0	0	21	0	0
shld	22	o	45	35	4	0	54	24
std	20	272	56	35	48	31	0	95

#### Rare 14 Opcode Testing (z-scores)

Opcode	Kernel RK	User RK	Tools	Bot	Trojan	Virus	Worms
bt	-1.2	-0.4	0.7	6.6	5.9	-0.7	4.8
fdivp	-1.3	-2,2	-0.3	3.8	2.8	-0.8	1.3
fild	-4.3	-6.5	-6.1	-1.5	-0.8	-2.6	2.1
fstcw	-0.7	-1.2	-1.0	3.3	2.2	-0.4	0.2
imul	-3.3	1.3	-5.9	4.4	-1.4	-1.7	0.9
int	45.0	26.2	28.7	-1.8	-1.0	2.4	-1.4
nop	-2.3	-3.6	-3.2	-5.0	-1.6	4.5	-2.3
pushf	-2.4	-3.7	-1.8	-3.9	-2.2	-0.7	-2.6
rdtsc	-0.7	-1.2	-1.1	1.1	-0.7	3.8	-0.9
sbb	-6.5	-2.0	3.4	-2.2	0.3	0.8	-2.0
setb	-0.5	4.7	0.6	4.6	7.9	-0.3	2.1
setle	-1.0	-1.6	-1.4	-1.6	1.3	-0.6	-1,2
shld	-1.0	0.6	0.6	-1.1	-0.9	1.0	0.2
std	4.8	1.4	0.8	0.3	2.4	-0.6	4.8

Higher
High
Similar
Low
Lower

Opcode Frequency

Tests
suggests
opcode
frequency
roughly

1/10 lower 1/5 higher 7/10 same vs

goodware

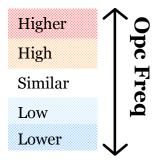
#### Rare 14 Opcodes: Interpretation

Cramer's V (in %)	63	36	42	17	16	10	12
Op	Krn	Usr	Tools	Bot	Trojan	Virus	Worm
bt							
fdivp							
fild							
fstcw							
imul							
int		▼.					
nop							
pushf							
rdtsc							
sbb							
setb							
setle							
shl <mark>á</mark>							
std							

**INT:** Rooktkits (and tools) make heavy use of software interrupts → tell-tale sign of RK?

Infrequent 14 opcodes much better predictor!

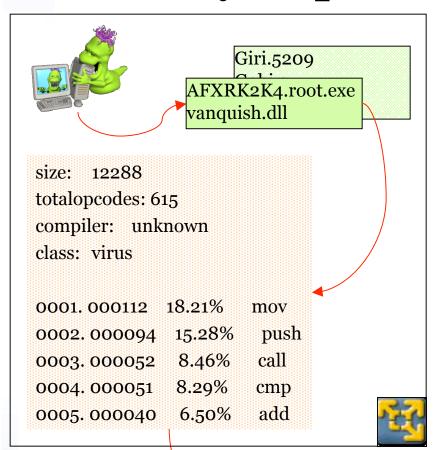
Explains 12-63% of variation



#### NOP:

Virus makes use → NOP sled, padding?

#### Summary: Opcode Distribution



Compare opcode fingerprints against various software classes for quick identification and classification

Malware opcode frequency distribution seems to deviate significantly from nonmalicious software

'Rare' opcodes explain more frequency variation then common ones

# Opcodes: Further directions

Acquire more samples and software class differentiation

Investigate sophisticated tests for stronger control of false discovery rate and type I error

Study n-way association with more factors (compiler, type of opcodes, size)

Go beyond isolated opcodes to semantic 'nuggets' (sizewise between isolated opcodes and basic blocks)

Investigate equivalent opcode substitution effects

#### Related Work

M. Weber (2002): *PEAT – Toolkit for Detecting* and Analyzing Malicious Software

R. Chinchani (2005): A Fast Static Analysis
 Approach to Detect Exploit Code Inside
 Network Flows

S. Stolfo (2005): Fileprint Analysis for Malware Detection

# Fingerprint: Win 32 API calls

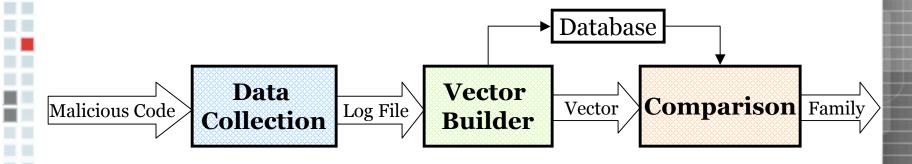
Joint work with Chris Ries

**Synopsis:** Observe and record Win32 API calls made by malicious code during execution, then compare them to calls made by other malicious code to find similarities

Goal: Classify malware quickly into a family (set of variants make up a family)

**Main result**: Simple model yields > 80% correct classification, call vectors seem robust towards different packer

### Win 32 API call: System overview

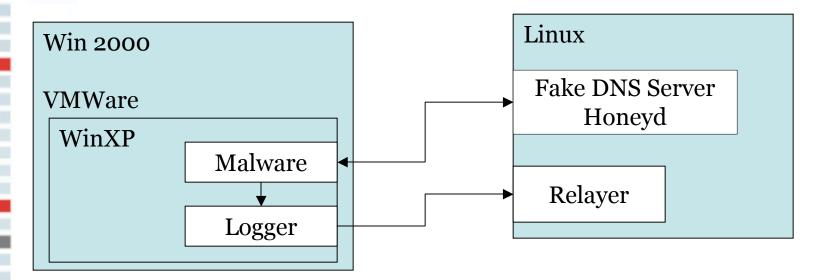


**Data Collection:** Run malicious code, recording Win32 API calls it makes

**Vector Builder:** Build count vector from collected API call data and store in database

Comparison: Compare vector to all other vectors in the database to see if its related to any of them

#### Win 32 API Call: Data Collection



Malware runs for short period of time on VMWare machine, can interact with fake network

API calls recorded by logger, passed on to Relayer Relayer forwards logs to file, console

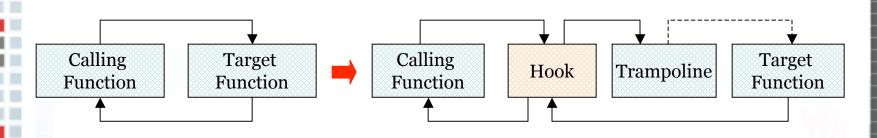
### Win 32 API Call: Call Recording

Malicious process is started in suspended state

DLL is injected into process's address space

When DLL's **DllMain()** function is executed, it hooks the Win32 API function

Hook records the call's time and arguments, calls the target, records the return value, and then returns the target's return value to the calling function.



Function call before hooking

Function call after hooking

## Win 32 API call: Call Vector

Function Name	FindClose	FindFirstFileA	CloseHandle	EndPath	
Number of Calls	62	12	156	0	•••

Column of the vector represents a hooked function and # of times called

1200+ different functions recorded during execution

For each malware specimen, vector values recorded to database

# Win 32 API call: Comparison

Computes cosine similarity measure *csm* between vector and each vector in the database

$$csm(\vec{v}_1, \vec{v}_2) = \frac{\vec{v}_1 \cdot \vec{v}_2}{|\vec{v}_1||\vec{v}_2|} = \longleftarrow$$

If csm(vector, most similar vector in the database) > threshold → vector is classified as member of family<sub>most-similar-vector</sub>

Otherwise vector classified as member of family<sub>no-variants-yet</sub>

### Win 32 API call: Results

**Misclassifications** 

27

Discrepancies

Collected 77 malware samples

Classification made by 17 major AV scanners produced 21 families (some aliases)

0.62

48

~80 % correct with *csm* threshold 0.8

0.99

Threshold	☑ %	☑ #	false fam.	both	miss. fam.
0.7	8.0	62	5	8	2
0.75	8.0	62	5	6	4
8.0	0.82	63	3	6	5
0.85	0.82	63	2	4	8
0.9	0.79	61	1	4	10
0.95	0.79	61	2	3	11

Family	# of	# correct
	members	
Apost	1	0
Banker	4	4
Nibu	1	1
Tarno	2	2
Beagle	15	14
Blaster	1	1
Frethem	3	2
Gibe	1	1
Inor	2	0
Klez	1	1
Mitgleider	2	2
MyDoom	10	8
MyLife	5	5
Netsky	8	8
Sasser	3	2
SDBot	8	5
Moega	3	3
Randex	2	1
Spybot	1	0
Pestlogger	1	1
Welchia	6	6

### Win 32 API call: Packers

Wide variety of different packers used within same families

Dynamic Win 32 API call fingerprint seems robust towards packer

8 Netsky variants in sample, 7 identified

Variant	Packer	Identified
Netsky.AB	PECompact	V
Netsky.B	UPX	
Netsky.C	PEtite	$\checkmark$
Netsky.D	PEtite	$\overline{\checkmark}$
Netsky.K	tElock	$\checkmark$
Netsky.P	FSG	$\checkmark$
Netsky.S	PE-Patch, UPX	$\overline{\mathbf{Q}}$
Netsky.Y	PE Pack	$\overline{\checkmark}$

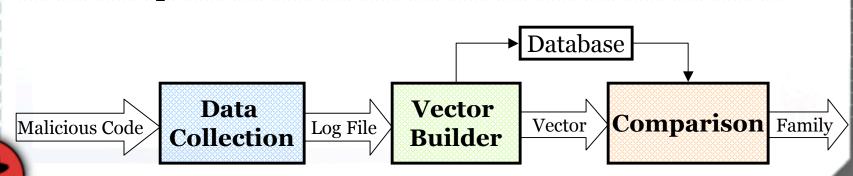
# Summary: Win 32 API calls

Allows researchers and analysts to quickly identify variants reasonably well, without manual analysis

Simple model yields > 80% correct classification

Resolved discrepancies between some AV scanners

Dynamical API call vectors seem robust towards different packer



#### API call: Further directions

Acquire more malware samples for better variant classification

Explore resiliency to obfuscation techniques (substitutions of Win 32 API calls, call spamming)

Investigate patterns of 'call bundles' instead of just isolated calls for richer identification

Replace VSM with finite state automaton that captures rich set of call relations

#### Related Work

R. Sekar et al (2001): A Fast Automaton-Based Method for Detecting Anamalous Program Behaviour

J. Rabek et al (2003): DOME – Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code

K. Rozinov (2005): Efficient Static Analysis of Executables for Detecting Malicous Behaviour

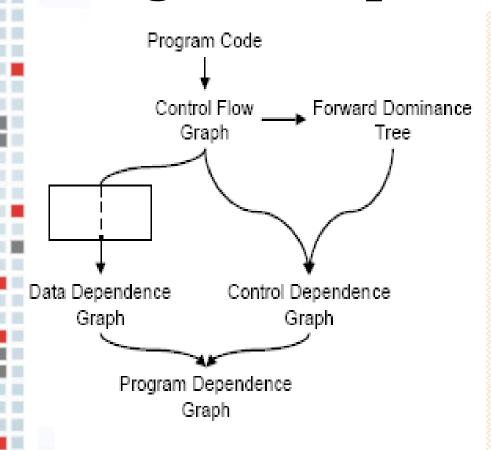
# Fingerprint: PDG measures

**Synopsis:** Represent binaries as a System Dependence Graph, extract graph features to construct 'graph-structural' fingerprints for particular software classes

**Goal:** Compare 'graph structure' fingerprint of unknown binaries across non-malicious software and malware classes for identification, classification and prediction purposes

Main result: Work in progress

## Program Dependence Graph



Picture from J. Stafford (Colorado, Boulder)

A PDG models intraprocedural

#### **Data Dependence:**

Program statements compute data that are used by other statements.

#### **Control Dependence:**

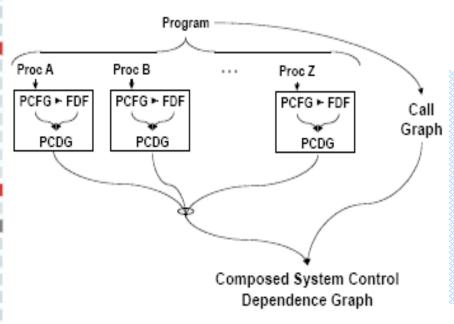
Arise from the ordered flow of control in a program.

# Program Dependence Graph

```
int main() {
      int sum = 0;
                                   Control dependence
      int i = 1;
      while (i < 11) {
                                   Flow dependence
            sum = sum + i;
            i = i + 1;
      printf("%d\n",sum);
      printf("%d\n",i);
                              Enter
                              T
                                        printf(sum)
                                                       printf(i)
                       while (i < 11)
  sum = 0
        sum = sum + i
                              i = i + i
```

Material from Codesurfer

# System Dependence Graph



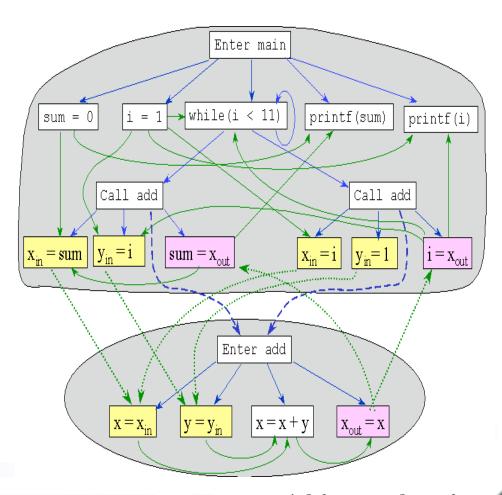
A SDG of a program are the aggregated PDGs augmented with the calls between functions

Picture from J. Stafford (Colorado, Boulder)

A SDG models control, data, and call dependences in a program

# System Dependence Graph

```
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
          sum = add(sum, i);
          i = add(i,1);
    printf("%d\n",sum);
    printf("%d\n",i);
 int add(int x, int y) {
      return x + y;
```



Material from Codesurfer

### Graph measures as a fingerprint

Binaries represented as a System Dependence Graph (SDG)

Tally distributions on graph measures:

Edge weights (weight is jump distance)

Node weight (weight is number of statements in basic block)

**Centrality** ("How important is the node")

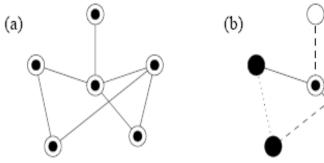
Clustering Coefficient ("probability of connected neighbours")

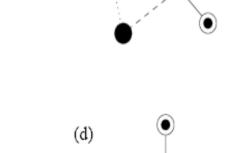
**Motifs** ("recurring patterns")

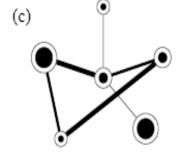
→ Statistical structural fingerprint

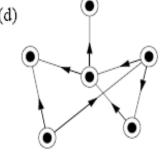
Example: H. Flake (BH 05) Graph-structural measure

# Primer: Graphs









Picture from Mark Newman (2003)

Graphs (Networks) are made up of vertices (nodes) and edges

An edge connects two vertices

Nodes and edges can have different weights (b,c)

Edges can have directions (d)

## Modeling with Graphs

C	at	e	g	o	ry	7
			$\boldsymbol{\mathcal{O}}$		•	

#### **Social**

Set of people/groups with some interaction between them

#### **Information**

Information linked together

#### **Technology** (Transmission)

resource or commodity distribution/transmission

#### **Biological**

biological 'entities' interacting

#### **Physical Science**

physical 'entities' interacting

#### Example (nodes, edge)

Friendship (people, friendship bond)

Business (companies, business dealings)

Movies (actors, collaboration)

Phone calls (number, call)

Citation (paper, cited)

Thesaurus (words, synonym)

www (html pages, URL links)

Power grid (power station, lines)

Telephone

Internet (routers, physical links)

Genetic regulatory network (proteins, dependence)

Cardiovascular (organs, veins) [also transmission]

Food (predator, prey)

Neural (neurons, axons)

Chemistry (conformation of polymers, transitions)

# Measures: Centrality

Centrality tries to measure the 'importance' of a vertex

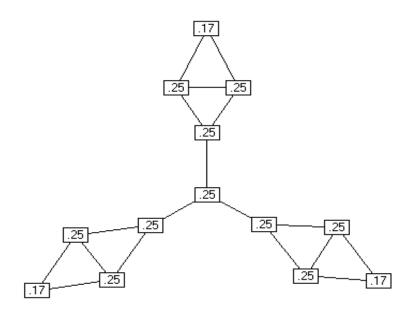
**Degree centrality:** "How many nodes are connected to me?"

Closeness centrality: "How close am I from all other nodes?"

**Betweenness centrality:** "How important am I for any two nodes?"

Freeman metric computes centralization for entire graph

## Measure: Degree centrality



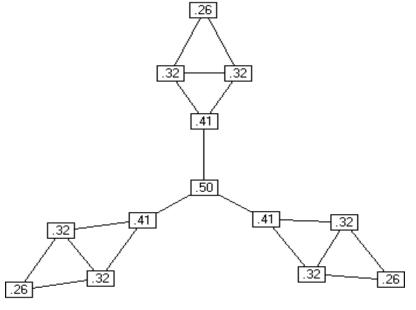
$$M_d(n_i) = \sum_{j=1, i \neq j}^{N} edge(i, j)$$

Degree centrality: "How many nodes are connected to me?"

Normalized:

$$M_d(n_i) = \frac{\sum_{j=1, i \neq j}^{N} edge(n_i, n_j)}{N-1}$$

## Measure: Closeness centrality



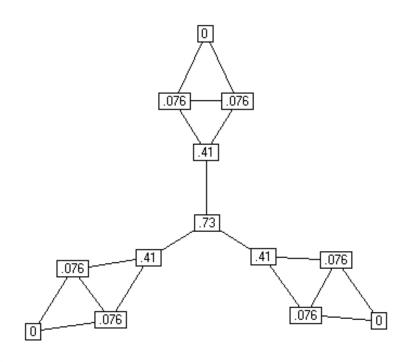
$$M_{c}(n_{i}) = \left[\sum_{j=1}^{N} d(n_{i}, n_{j})\right]^{-1} \qquad M_{c}'(n_{i}) = M_{c}(n_{i})(N-1)$$

**Closeness** centrality: "How close am I from all other nodes?"

Normalized:

$$M'_{C}(n_{i}) = M_{C}(n_{i})(N-1)$$

## Measure: Betweenness centrality



Betweenness centrality: "How important am I for any two nodes?"

Normalized:

$$C_B(n_i) = \sum_{j \neq k \neq i} g_{jk}(n_i) / g_{jk}$$
  $C_B(n_i) = \frac{C_B(n_i)}{(N-1)(N-2)/2}$ 

# Local structure: Clusters and Motifs

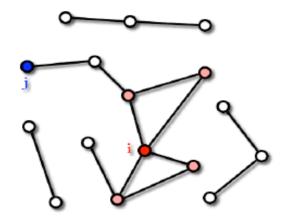
Graphs can be decomposed into constitutive subgraphs

**Subgraph:** A subset of nodes of the original graph and of edges connecting them (does not have to contain all the edges of a node)

Cluster: Connected subgraph

**Motif:** Recurring subgraph in networks at higher frequencies than expected by random chance

# Measure: Clustering Coefficient



Cliques (completely connected subgraphs)

$$k = N - 1, \ n = \frac{N(N - 1)}{2}$$



How close the neighborhood of a node is to a

clique? Edges among first neighbors of node *i* 

$$C_i \equiv \frac{n_i}{k_i(k_i-1)/2}$$
,  $k \neq 0,1$  or

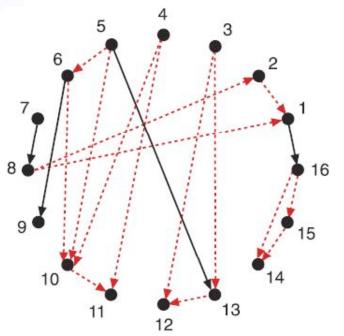
Clustering coefficient

$$C_i \equiv \frac{nr. \ of \ triangles \ connected \ to \ i}{nr. \ of \ triples \ centered \ on \ i}$$

Average clustering coefficient

$$C \equiv \frac{1}{N} \sum_{i=1}^{N} C_i$$

### Measure: Network motifs



A motif in a network is a subgraph 'pattern'

Recurs in networks at higher frequencies than expected by random chance

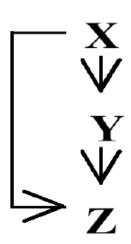


Motif may reflect underlying generative processes, design principles and constraints and driving dynamics of the network

# Motif example:

#### **Feed-forward loop:**

X regulates Y and Z Y regulates Z



Found in

Biochemistry (Transcriptional regulation)

Neurobiology (Neuron connectivity)

Ecology (food web)

Engineering (electronic circuits) ..

and maybe Computer Science (PDGs) ??

## Graph measures as a fingerprint

Binaries represented as a System Dependence Graph (SDG)

Tally distributions on graph measures:

Edge weights (weight is jump distance)

**Node weight** (weight is number of statements in basic block)

**Centrality** ("How important is the node")

Clustering Coefficient ("probability of connected neighbours")

**Motifs** ("recurring patterns")

→ Statistical structural fingerprint

# Summary: SDG measures

**Synopsis:** Represent binaries as a System Dependence Graph, extract graph features to construct 'graph-structural' fingerprints for particular software classes

**Goal:** Compare 'graph structure' fingerprint of unknown binaries across non-malicious software and malware classes for identification, classification and prediction purposes

Main result: Work in progress

#### Related Work

H. Flake (2005): Compare, Port, Navigate

M. Christodorescu (2003): Static Analysis of Executables to Detect Malicious Patterns

A. Kiss (2005): Using Dynamic Information in the Interprocedural Static Slicing of Binary Executables

### References

#### **Statistical testing:**

- S. Haberman (1973): *The Analysis of Residuals in Cross-Classified Tables*, pp. 205-220
- B.S. Everitt (1992): *The Analysis of Contingency Tables (2<sup>nd</sup> Ed.)*

#### Network Graph Measures and Network Motifs:

- L. Amaral et al (2000): Classes of Small-World Networks
- R Milo, Alon U. et al (2002): *Network Motifs: Simple Building Blocks of Complex Networks*
- M. Newman (2003): The structure and function of complex networks
- D. Bilar (2006): Science of Networks. http://cs.colby.edu/courses/cs298

#### **System Dependence Graphs:**

- GrammaTech Inc.: Static Program Dependence Analysis via Dependence Graphs. http://www.codesurfer.com/papers/
- Á. Kiss et al (2003). Interprocedural Static Slicing of Binary Executables