String Signature based Malware Detection using Machine Learning, Spring 2011

Sandipan Dey,
Independent Study,
CMSC 699,
UMBC CSEE

March 9, 2011

1 Problem Definition

Given a set of n binary (executable) files $\{F_i\}_{1\ldots n}$, each one as a set of instructions (as hex op-code/mnemonic) $\{I_{ij}\}_{j=1\ldots n_i}, I_{ij} \in F_i, |F_i| = n_i$, we have to find (using machine learning string signature based techniques) which of the files are malwares or malware affected.

1.1 Generative Model

In this (supervised) case, we already have a set of training files with known labels (as malware or not). We have to learn a model from the training data set. The result of the learning process may be in terms of

1. A classifier (a function $\hat{f} : \{F_i\} \rightarrow \{\text{"}Y\text{"}, \text{"}N\text{"}\}$) with output "Y" (malware) or "N"
2. A set of Grammar rules learnt from the training data that defines a malware.

Like any standard classification/regression problem, we may attempt to express the classifier function $\hat{f}$ as a weight matrix W. But, the challenge here is to define the attributes or the features to which the weights are to be applied to.

2 State of the Art

One of the definitions of "Malware", as discussed in [7], is "any code added, changed, or removed from a software system in order to intentionally cause harm or subvert the intended function of the system". Examples of malwares being viruses, worms and trojan horses/spywares. There are 3 different detection techniques: 1) Signature based 2) Specification (behavior) based 3) Anomaly based.

A signature-based detection technique [7]

1. Attempts to model the malicious behavior of malware, stores the model (as signature, typically represented by sequences of code) in its repository and uses this model in the detection of malware.
2. Drawback: cannot detect zero-day attacks (an attack for which there is no corresponding signature stored in the repository), the repository of signatures is a weak approximation to the set of possible malicious behaviors.
3. Can be static (automaton, regex, LCS are used to match a new binary with the model learnt), dynamic or hybrid.

As discussed in [1], CNG (common n gram analysis) can be used for detection of new malicious code, with the following supervised steps:

1. In the training phase, the data for the classes MC (malicious) and BC (benign) are collected and n-grams (window of n consecutive words)

with their normalized frequencies are counted. The L most frequent n-grams with their normalized frequencies represent features of a class profile.

2. In the testing phase, a new instance is classified by choosing the class with the closest class profile using the relative distance measure $\sum_{s \in features} \left(\frac{f_1(s) - f_2(s)}{(f_1(s) + f_2(s))/2} \right)^2$.

3. Parameters n (n-gram size), L (profile length) are varied and 5-fold cross validation is used, obtaining high accuracies.

Alternatively, in a variant of [1], as discussed in [8, 9],

1. In the training phase, the features are chosen by finding the most relevant n-grams (maximizing the information gain (average mutual information) $I_G(j) = \sum_{v_g \in \{0,1\}} \sum_{C_i \in \{MC,BC\}} P(v_j, C_i) log \frac{P(v_j, C_i)}{P(v_j)P(C_i)}$, where each n-gram is viewed as a boolean attribute that is either present in (i.e., T) or absent from (i.e., F) in each of the training example executable.

2. Many different models are learnt (k-nearest neighbors, Naive Bayes, Decision Trees, Boosted Classifiers [8], SVM [9] in the training phase.

3. In the testing phase, the learnt classifiers are used to classify the new instance, results indicating that Boosted ensemble classifiers work best, with 95% area-under-ROC [8] and SVM gives excellent results with high true positive or recall $\left(\frac{TP}{TP+FN}\right)$, low false alarm rate $\left(\frac{FP}{FP+TN}\right)$ and high overall accuracy $\left(\frac{TP+TN}{TP+FP+TN+FN}\right)$ [9].

4. In [9], value of the parameter n is chosen by upper-bounding the information loss and another parameter s (shift length) is accordingly chosen. Real huge executable are considered as datasets, instead of skewed ones.

In [6],

1. A novel automatic generation of string signature is proposed instead of hash signatures (although with very low FP rate, malware samples covered are very low).

2. It uses 5-gram (5 byte sequences) Markov model, which is a Markov chain of order 4. Uses relative information gain again to prune away the insignificant ones and also uses a fixed memory constraint.

3 Approach, Implementation and Questions

1. Why relative information gain is assumed to give the most relevant features regarding the malwares? I understand the n-grams with more I_G have more information contained in them, but does it necessarily have to be about the malicious portion of the binary? Why can't it happen that the portions of code that are not that relevant in terms of information gain can not contain malicious code?

2. Malwares can be injected into any files (not necessarily only binary files), do we need to consider that too?

3. Can some discriminative pre-processing be done prior to training? For instance, here no domain knowledge about the instructions (e.g., which instrcutions can group together, which can not) are used to find the n-grams (it may be possible to find some portion of code that is totally benign, e.g., xor eax, eax, that can be eliminated before starting).

4. How can we learn CFGs (e.g, of the form $MC \rightarrow$ and $BC \rightarrow$) out of this? can the most relevant n-grams be chosen as the terminal symbols?

5. Virus Collections from Vx Heven are used for malware samples. System binaries are used as benign executables.

6. NASM dissassembler is used to dissassemble the code into mnemonics and hex dump.

7. Program bin2tmp converts the binary code to temporal data. It first obtains the hex dump and then hashes an opcode to Y_t, where $t = 1, 2, \ldots$ is index (sequence) of an instruction.

8. Program freq calculates the frequency (probability) distribution of instructions in a binary.

9. Since pentium has a huge set of instructions, to learn mnemonics of the corresponding opcode a program oplearnt is written, that basically learn new mnemonics not yet learnt and stores.

10. Some preprocessing using SAX and other clustering techniques [4] are done.

11. Can the eigen analysis of $tf - idf$ matrix (laten semantic analysis) give any useful concept abut malwares? what about modeling it in terms of Hidden Markov Model / Expectation Maximization / ICA?

String Kernel and SVM

SVM and Kernel

The input space is separated by a hyperplane

$$f(x) = sign(wx + b)$$

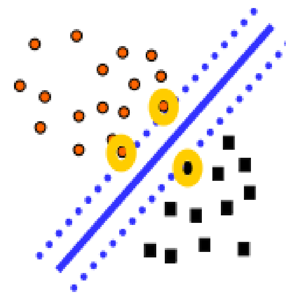
It turns out that the solution is a linear combination of training points [3]

$$w = \sum_i \alpha_i y_i x_i$$
$$\alpha_i \geq 0$$

and the (primal) optimization problem (maximizing the margin between the support vector planes) becomes [2]

$$\min \frac{1}{2}||w||^2$$
$$s.t.\ y_i(wx_i - b) \geq 1$$

as shown in the following figure from [2]. The decision



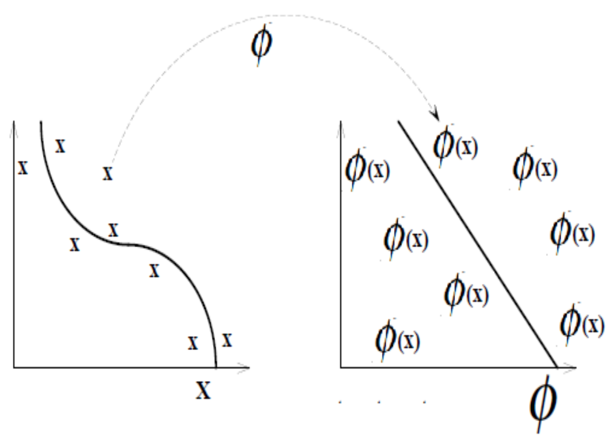
function can be re-written as

$$f(x) = sign\left(\sum_i \alpha_i y_i \langle x_i, x\rangle + b\right)$$

When the features are not linearly separable, they are mapped onto a high dimensional space ($x \to \phi(x)$) where they become linearly separable and then linear classifier can be used (shown in the following figure) and the decision function becomes

$$f(x) = sign\left(\sum_i \alpha_i y_i \langle \phi(x_i), \phi(x)\rangle + b\right)$$
$$= sign\left(\sum_i \alpha_i y_i K(x_i, x) + b\right)$$

where K is the Kernel. As described in [3], Kernel is



a function that returns the value of the dot product

C++ code

```cpp
#include <cstdio>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

Asm code (g++)

```
___gnu_compiled_cplusplus:
    .def    ___main;    .scl    2;  .type   32; .endef
.text
LC0:
    .ascii "Hello World\12\0"
    .align 4
.globl _main
    .def    _main;  .scl    2;  .type   32; .endef
_main:
    pushl %ebp
    movl %esp,%ebp
    call ___main
    pushl $LC0
    call _printf
    addl $4,%esp
    xorl %eax,%eax
    jmp L1
    xorl %eax,%eax
    jmp L1
    .p2align 4,,7
L1:
    movl %ebp,%esp
    popl %ebp
    ret
    .def    _printf;    .scl    3;  .type   32; .endef
```

Dissassembled code (nasm)

```
00000000  4D         dec bp
00000001  5A         pop dx
00000002  90         nop
00000003  0003       add [bp+di],al
00000005  0000       add [bx+si],al
00000007  0004       add [si],al
00000009  0000       add [bx+si],al
0000000B  00FF       add bh,bh
0000000D  FF00       inc word [bx+si]
0000000F  00B80000   add [bx+si+0x0],bh
00000013  0000       add [bx+si],al
00000015  0000       add [bx+si],al
00000017  004000     add [bx+si+0x0],al
0000001A  0000       add [bx+si],al
0000001C  0000       add [bx+si],al
0000001E  0000       add [bx+si],al
00000020  0000       add [bx+si],al
00000022  0000       add [bx+si],al
00000024  0000       add [bx+si],al
00000026  0000       add [bx+si],al
00000028  0000       add [bx+si],al
0000002A  0000       add [bx+si],al
0000002C  0000       add [bx+si],al
0000002E  0000       add [bx+si],al
...       ...        ...
```

total 158475 instructions !

Figure 1: Dissassembling HelloWorld

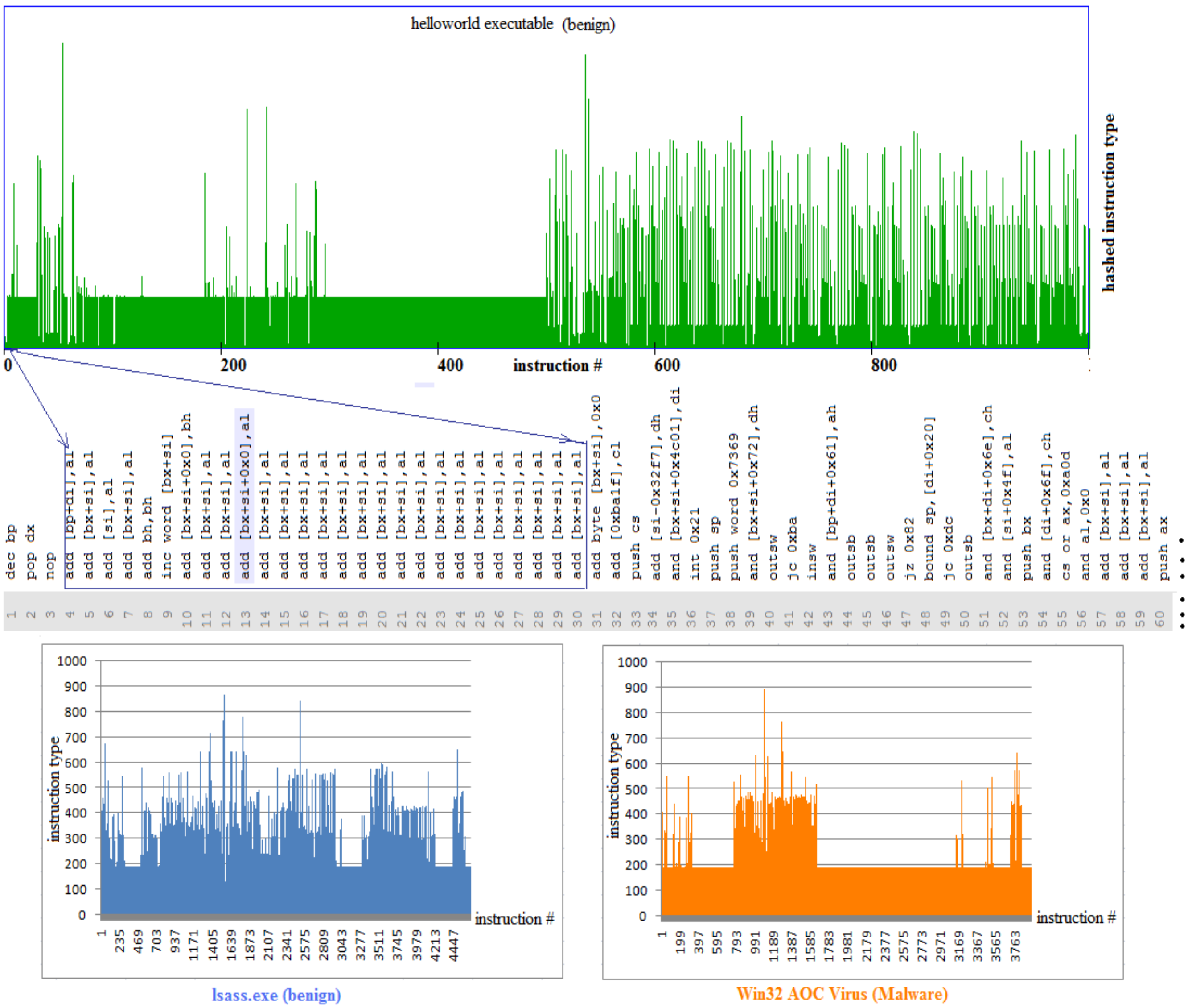
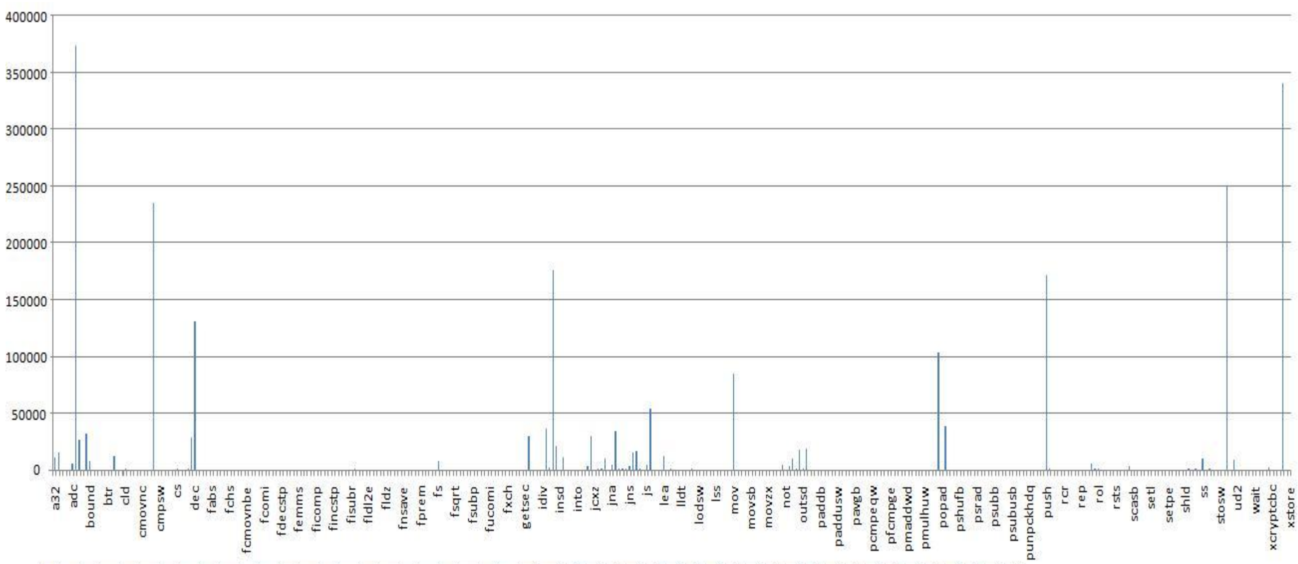


Figure 2: Binary to Temporal Conversion

Figure 3: Temporal Conversion of Malwares

Mnemonic	OpCode	Mnemonic	OpCode	Mnemonic	OpCode	Mnemonic	OpCode	Mnemonic	OpCode
a32	676E	cmc	F5	fadd	D802	fdivrp	DEF1	fldlg2	D9EC
aaa	37	cmovc	0F4200	faddp	1-Dec	femms	0F0E	fldln2	D9ED
aad	D562	cmovnc	0F4300	fbld	DFA40000	ffree	DDC1	fldpi	D9EB
aam	D400	cmovpo	0F4B00	fbstp	DF34	ffreep	DFC1	fldz	D9EE
aas	3F	cmovs	0F4801	fchs	D9E0	fiadd	36DE02	fmul	DC8D0100
adc	1000	cmp	3C00	fcmovb	DAC1	ficom	DE5400	fmulp	9-Dec
add	005F5F	cmpsb	A6	fcmovbe	DAD5	ficomp	DA1A	fninit	DBE3
and	2000	cmpsw	A7	fcmove	DAC9	fidiv	DEB40000	fnop	D9D0
andnps	0F554800	cmpxchg	0FB04100	fcmovnb	DBC7	fidivr	DA7901	fnsave	DDB14300
arpl	637274	cpu_read	0F3D	fcmovnbe	DBD0	fild	DF060400	fnstcw	D97DFE
bound	627379	cpu_write	0F3C	fcmovne	DBC9	fimul	DA0D	fnstenv	D9B24300
bsf	0FBC4800	cpuid	0FA2	fcmovnu	DBDA	fincstp	D9F7	fnstsw	DD3A
bsr	0FBD4800	cs	2E7465	fcmovu	DADF	fist	DF160100	fpatan	D9F3
bt	0FA34100	cwd	99	fcom	DC910100	fistp	DB980000	fprem	D9F8
btc	0FBB4100	daa	27	fcomi	DBF1	fisttp	DF8B0000	fprem1	D9F5
btr	0FB34100	das	2F	fcomip	DFF1	fisub	DA24	fptan	D9F2
bts	0FAB4100	db	0	fcomp	DC5B01	fisubr	DEAE0000	frndint	D9FC
call	FF1A	dec	49	fcompp	DED9	fld	D903	frstor	DDA00000
cbw	98	div	F632	fcos	D9FF	fld1	D9E8	fs	645F
clc	F8	ds	3.00E+99	fdecstp	D9F6	fldcw	D96DFE	fscale	D9FD
cld	FC	enter	C8030080	fdiv	DC7601	fldenv	D9A10100	fsetpm	DBE4
cli	FA	es	26B80200	fdivp	DEFF	fldl2e	D9EA	fsin	D9FE
clts	0F06	fabs	D9E1	fdivr	DC7D02	fldl2t	D9E9	fsincos	D9FB

Figure 4: Opcodes learnt



	tf
a32	11332
aaa	15555
aad	244
aam	433
aas	203
adc	6020
add	373494
and	26366
andnps	1
arpl	32168
bound	8145
bsf	6
bsr	10
bt	3
btc	3
btr	3
bts	3
call	12040
cbw	363
clc	405
cld	861
cli	202
clts	1
cmc	476
cmovc	3
cmovnc	4
cmovpo	5
cmovs	18
cmp	234799
cmpsb	224
cmpsw	192
cmpxchg	5
cpu_read	1
cpu_write	1
cpuid	4
cs	1336
cwd	195
daa	75
das	1466
db	28531
dec	130924
div	116
ds	73
enter	503
es	95
fabs	1
fadd	417
faddp	48
fbld	19
• • •	• • •

Figure 5: Probability distribution of instructions (HelloWorld)

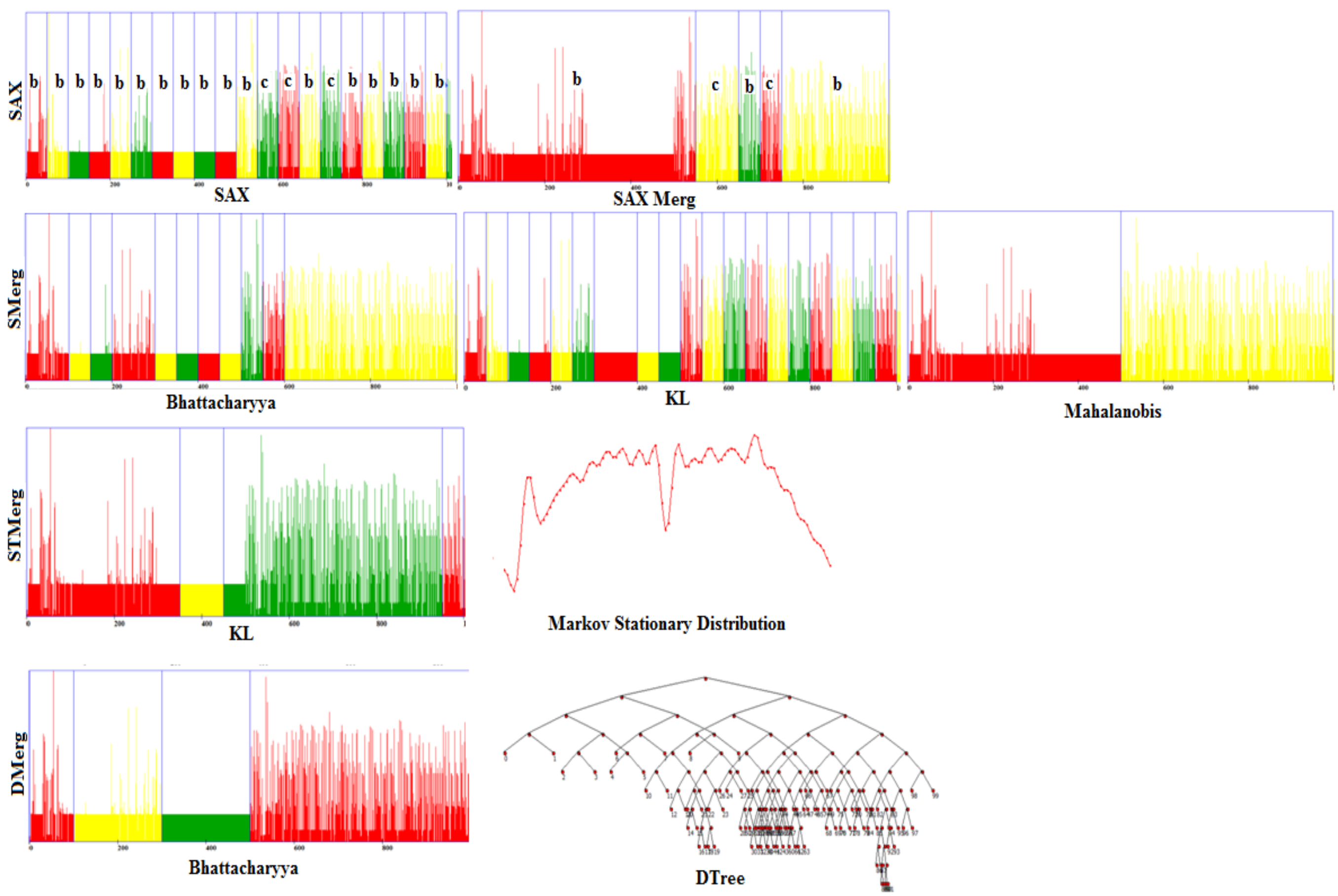


Figure 6: Different Compression Techniques

between the images of the two arguments. By Mercers Theorem, the kernel matrix is Symmetric Positive Definite. Some popular kernels include Radial Basis Function (RBF) kernel, polynomial kernel etc.

$$K(x,z) = e^{\frac{-||x-z||^2}{2\sigma}} \text{ (RBF)}$$
$$K(x,z) = \langle x,z \rangle^d \text{ (Polynomial)}$$

String Kernel

The main idea of string kernels [5] is to compare the hexdumps of the binary files not by words, but by the substrings they contain.

$$\Phi_u(s) = \sum_{\mathbf{i}:u=s[\mathbf{i}]} \lambda^{l(\mathbf{i})}$$
$$K_n(s,t) = \sum_{u\in\Sigma^n} \Phi_u(s)\Phi_u(t)$$
$$= \sum_{u\in\Sigma^n} \sum_{\mathbf{i}:u=s[\mathbf{i}]} \sum_{\mathbf{j}:u=t[\mathbf{j}]} \lambda^{l(\mathbf{i})+l(\mathbf{j})}.$$
$$\lambda \leq 1$$

Structure Learning and Markov Random Field

References

[1] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. Detection of new malicious code using n-grams signatures. In *PST*, pages 193–196, 2004.

[2] K. P. Bennett and C. Campbell. Support vector machines: Hype or hallelujah? *SIGKDD Explorations*, 2(2):1–13, 2000.

[3] N. Cristianini. Support vector and kernel machine. In *ICML Tutorial*, 2001.

[4] S. Dey, V. P. Janeja, and A. Gangopadhyay. Temporal neighborhood discovery using markov models. In *ICDM*, pages 110–119, 2009.

[5] B. Fortuna. String kernels.

[6] K. Griffin, S. Schneider, X. Hu, and T. cker Chiueh. Automatic generation of string signatures for malware detection. In *RAID*, pages 101–120, 2009.

[7] N. Idika and A. P. Mathur. A survey of malware detection techniques. 2007.

[8] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research 7*, pages 2721–2744, 2006.

[9] T. Stibor. A study of detecting computer viruses in real-infected files in the -gram representation with machine learning methods. In *IEA/AIE (1)*, pages 509–519, 2010.