From NLP (Natural Language Processing) to MLP (Machine Language Processing)

Peter Teufl (peter.teufl@iaik.tugraz.at)[1] and
Udo Payer (udo.payer@campus02.at)[2] and
Guenter Lackner (guenther.lackner@studio78.at)[3]

[1] Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
[2] CAMPUS02, Graz University of Applied Science
[3] Studio78, Graz

Abstract. Natural Language Processing (NLP) in combination with Machine Learning techniques plays an important role in the field of automatic text analysis. Motivated by the successful use of NLP in solving text classification problems in the area of e-Participation and inspired by our prior work in the field of polymorphic shellcode detection we gave classical NLP-processes a trial in the special case of malicious code analysis. Any malicious program is based on some kind of machine language, ranging from manually crafted assembler code that exploits a buffer overflow to high level languages such as Javascript used in web-based attacks. We argue that well known NLP analysis processes can be modified and applied to the malware analysis domain. Similar to the NLP process we call this process Machine Language Processing (MLP). In this paper, we use our e-Participation analysis architecture, extract the various NLP techniques and adopt them for the malware analysis process. As proof-of-concept we apply the adopted framework to malicious code examples from Metasploit.

Key words: Natural Language Processing, Malware Analysis, Semantic Networks, Machine Language Processing, Machine Learning, Knowledge Mining

1 Introduction

Natural Language Processing (NLP) involves a wide range of techniques that enable the automated parsing and processing of natural language. In the case of written text, this automated processing ranges from the lexical parsing of sentences to applying sophisticated methods from machine learning and artificial intelligence in order to gain insight on the covered topics. Although NLP is a complex and computationally intensive task, it gains more and more importance due to the need to automatically analyze large amounts of information stored within arbitrary text sources on the Internet. For such large text corpora it is not feasible for human experts to read, to understand and to draw conclusions in a complete manual way.

An example for such a domain is the electronic participation (further denoted as e-Participation) of citizens within a governmental decision process. Typically, this process involves citizens that express their opinion on certain topics and domain experts that analyze these opinions and extract important concepts and ideas. In order to speed up the process and improve the results it makes sense to apply NLP techniques that support the domain experts. Therefore, we have implemented and employed an e-Participation analysis framework [1].

Due to previous work in the field of malicious code detection—*especially in the field of polymorphic shellcode detection* [2], [3] — we realized that the analysis of natural languages is somewhat similar to the analysis of machine languages. Malware, regardless of its nature, is always based on some kind of programming language used to encode the commands that an attacker wants to execute on a victim's machine. This can be raw assembler code or a high level scripting language such as Javascript. The process of detecting malware is to identify malicious code within large amounts of regular code. There are a wide range of malware detection methods ranging from simple signature detection methods to highly sophisticated methods based on machine learning. However, before such methods can be deployed for malware detection we need to analyze and understand the underlying code itself. Due to self mutating code, encryption, metamorphic and polymorphic engines, and other methods designed to camouflage the malware itself, it is not possible to create simple signatures anymore. Therefore, we need to extract other more complex relations within the machine language that allow us to devise more robust detection methods.

In this paper, we argue that the same NLP processes and techniques used for the analysis of natural language can be mapped and applied to machine language. Analog to the NLP process we introduce the concept of Machine Language Processing (MLP). In order to find relevant MLP processes, we extract the various analysis steps of our e-Participation analysis framework and define corresponding MLP steps. In order to test the implementability of this approach we finally apply the modified framework to real assembler code extracted from various decoding engines generated by the Metasploit framework.

Although the proof-of-concept and the NLP-to-MLP transformations focus on assembler code, the discussed techniques could easily be extended to arbitrary machine languages.

2 Related Work

Malware is defined as some piece of software with the only intention to perform some harmful actions on a device, which is already under control or is intended to be under control of an attacker. Malware analysis—*on the contrary*—is the process of re-engineering these pieces of software or to analyze the behavior for the only purpose to identify or demonstrate the harmfulness of these pieces of software (such as a virus, worm, or Trojan horses). Actually, malware analysis can be divided into

- behavior analysis (*dynamic analysis*) and

– code analysis (*static analysis*)

Since no generic tool exists to perform this analysis automatically, the process of malware analysis is a manual one, which can fortunately fall back on a rich set of efficient but simple tools. A tricky part in malware analysis is to detect pieces of code, which are only triggered under some specific conditions (day, time, etc. ...). In such cases, it is essential to disassemble the whole executable and to analyze all possible execution pathes. Finding and watching such execution pathes (e.g. by the help of a disassembler) is forming the core mechanism of a sophisticated code analysis process.

As "dynamic" approach to detect execution chains within a piece of software is to execute and analyze its behavior in a restricted environment. Such an environment can be a debugger, which is controlled by a human analyst, to step through each single line of code to see the code-execution happening and to understand the "meaning" of the code. Examples of such "sandbox"- techniques are CWSandbox [4], the Norman SandBox [5], TTAnalyze and Cobra [6]. Common to all these examples is that code is automatically loaded and analyzed in a virtual machine environment to find out the basic behavior and execution pathes. A special dynamic sandbox-method is the so called black box analysis. In this case, the system is studied without any knowledge about its internal construction. Observable during the analysis are only external in- and outputs as well as their timing relationships. After a successful simulation, a post mortem analysis will show effects of the malware execution. This post mortem analysis can be done by standard computer forensic tool chains.

In the case of malicious code analysis, the common idea is to use *analysis archtitectures* to make use of the huge number of useful tools in a controlled way. BitBlaze [7] for instance even tries to combine static- and dynamic analysis tools. The BitBlaze framework actually consists of three components: Vine, the static analysis tool, TEMU, the dynamic analysis component, and Rudder, a separate tool to combine dynamic and static analysis results.

NLP is a huge field in computer science about language- based interactions between computers and humans. It can basically be divided in the following two major areas:

– *Natural language generation systems* (LGS), which convert information from computer databases into readable human language and
– *Natural language understanding systems* (LUS), which are designed to convert samples of human language into a formal representation. Such a representation can be used to find out what concepts a word or phrase stands for and how these concepts fit together in a meaningful way.

Related to the content of this paper, we always think about NLP as an application that can deal with text in the sense of classification, automatic translations, knowledge acquisition or the extraction of useful information. In this paper, we will not link NLP to the generation of natural languages. Especially in the case of LUS, a lot of prior work exists, which was carried out by many different research groups (e.g. [8],[9]). Machine learning techniques have been

applied to the natural language problem, statistical analysis has been performed and large text corpora have been generated and have been used successfully in the field of NLP. Thus, several projects—*about innovative ways to run and improve NLP-methods*—have already been finished or are still ongoing - and we are quite sure that there will be many more.

3 Methods

3.1 NLP:

All NLP components of the platform are based on the lingpipe NLP API [10]. It is a Java API that covers a wide range of algorithms and techniques important for NLP: Examples are Part-of-Speech (POS) tagging, the detection of sentences, spelling correction, handling of text corpora, language identification, word sense disambiguation (e.g. [11]), etc. The techniques that are relevant for our text-analysis architecture will be shortly discussed in the subsequent sections. For a good overview of all these techniques we refer to the tutorials that come with the lingpipe package[4].

3.2 Semantic/Associative Networks and Spreading Activation (SA)

Associative networks [12] are directed or undirected graphs that store information in the network nodes and use edges (links) to present the relation between these nodes. Typically, these links are weighted according to a weighting scheme. Spreading activation (SA) algorithms [13] can be used to extract information from associative networks. Associative networks and SA algorithms play an important role within Information Retrieval (IR) systems such as [14], [15] and [11]. By applying SA algorithms we are able to extract *Activation Patterns* from trained associative networks. These *Activation Patterns* can then be analyzed by arbitrary supervised and unsupervised machine learning algorithms.

3.3 Machine Learning (ML)

For the supervised or unsupervised analysis of the *activation patterns* – the patterns generated by applying SA to the semantic/associative network – standard machine learning algorithms can be applied. Examples for supervised algorithms are the widely used Support Vector Machines (SVM), Neural Networks and Bayesian Networks. The family of unsupervised algorithms has an important role, since such techniques allow us to extract relations between features, to detect anomalies and to find similarities between patterns without having an a-priori knowledge about the analyzed data. Examples for such algorithms are Neural Gas based algorithms [16], Self Organizing Maps (SOM), Hierachical Agglomerative Clustering (HAC), or Expectation Maximation (EM). In this work we employ the Robust Growing Neural Gas algorithm (RGNG) [16].

[4] http://alias-i.com/lingpipe/demos/tutorial/read-me.html

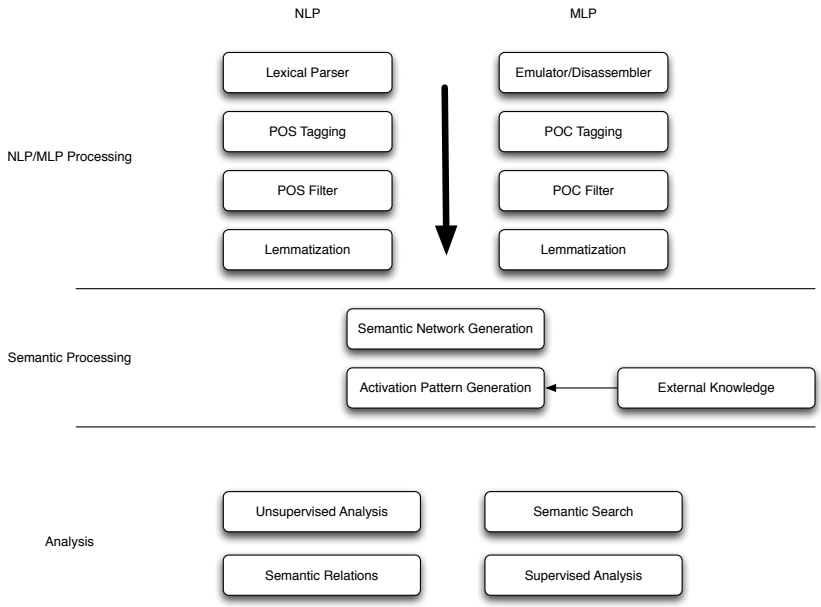

Fig. 1: MLP vs. NLP Processing

4 From NLP to MLP

In [1] we present an automated text-analysis architecture that is used for the analysis of various e-Participation related data-sets. The basic modules of this architecture are depicted in Figure 1. The remaining part of this section describes the various NLP and ML related submodules of this architecture and how they can be applied or transformed to MLP modules for malware analysis.

4.1 Lexical Parser/Emulator/Disassembler

NLP: For NLP, we need to convert a sequence of characters into a sequence of tokens. These tokens represent the terms of the underlying text. The conversion process is called lexical analysis. By using lexical parsers such as the Stanford Parser [17], we are able to extract the roles of terms within a sentence and the relations between these terms. Depending on the subsequent processing steps, this could range from a superficial analysis identifying some key grammatical concepts to a deep analysis that is able to extract fine details.

MLP: Raw machine code is a byte sequence that contains instructions that are executed by the processor. In addition, most of the available instructions have

parameters that are also encoded in the byte sequence. In order to extract information for further analysis, we need to process this byte sequence and extract the instructions and the parameters. In a simple scenario this could be done with a disassembler that extracts instructions from a given byte sequence. However, due to branch operations such as **jmp** or **call** these byte sequence is not processed by the CPU in a linear way. Thus, in order to extract the instruction chain the way it is executed on a CPU, we need to employ emulators or execute the code directly on the CPU. For the example presented later in this work, we utilize the PTRACE system call[5] on linux to execute code directly on the CPU (see Section 5.1 for a more detailed description). By applying such methods to the raw byte sequence, we are able to extract and inspect the instructions chains executed on the CPU. In analogy to NLP these instruction chains represent the written text, which needs to be analyzed. Similar to NLP the deepness of the analysis depends on the applied method. These methods range from extracting the instructions and their execution order to more complicated methods capable of identifying more complex structures: constructs such as loops, the necessary preparation for executing interrupts, branching etc.

4.2 POS (Part-of-Speech) Tagging, POC (Part-of-Code) Tagging

NLP: POS tagging can also be seen as part of the lexical analysis described in the previous section. However, since it plays an important role for text analysis, we describe it as separate process. In NLP, Part-of-Speech tagging is the process of identifying the role of each term in a sentence. The following example shows the POS tags for a given sentence: *Hello_RB I_PRP am_VBP a_DT little_JJ sentence_NN trying_VBG to_TO find_VB my_PRP place_NN within_IN this_DT text_NN ._.* The tags were obtained by using the online interface of the Stanford parser[6], where for example *NN* indicates nouns and *VB** identifies verbs and their different modes. POS tags are used for subsequent processing steps, which include the filtering of terms according to their tags and establishing relations between terms in a semantic network according to these tags.

MLP: Obviously, there are no nouns, verbs or related concepts in machine code, but there are similar concepts that could be used to tag single instructions. We call these tags Part-Of-Code (POC) tags. For the example presented in Section 5 we tag the instructions according to their functionality which results in the following categories: control flow, arithmetic, logic, stack, comparison, move, string, bit manipulation, flag manipulation, floating point unit instructions, other.

4.3 POS/POC Filtering

NLP: Depending on the subsequent analysis, it makes sense to keep only terms with certain POS tags. For the e-Participation related text analysis, we only

[5] http://linux.die.net/man/2/ptrace
[6] http://nlp.stanford.edu:8080/parser/

keep nouns, verbs and adjectives since the already convey a large part of the information within the text.

MLP: According to the determined POC tags, we can easily define filters that allow us to focus on branching behavior, arithmetic operations, logical operations etc.

4.4 Lemmatization

NLP: Before proceeding with the NLP analysis of POS tagged text, it makes sense to derive the lemmas of the remaining terms. By doing so we avoid the ambiguity of different forms such as inflected terms or plural forms. For example the term **bought** would be mapped to its lemma **buy** for further analysis.

MLP: When applying this process to machine code, we need to ask "What is the lemma of an assembler instruction?". There is not a single answer to this question, but there are several concepts that could be used for lemmatization:

- **Instruction without parameters:** In this case we strip away the parameters of an instruction and use the instruction as lemma.
- **Mapping of instructions**: Instructions that belong to the same family could be mapped to one instruction. An example would be the mapping of all **mov** derivates to one instruction.
- **High level interpretation:** In this case we focus on the operations performed by the instructions and not the instructions themselves. E.g. the instructions and their parameters **xor eax,eax** or **mov eax,0** or the chain **mov eax,5; sub eax,5** all have the same effect – the **eax** register contains the value 0. As we see, this effect can be achieved by using various instructions or instruction chains. Such techniques are typically employed by polymorphic and metamorphic engines trying to camouflage their real purpose by changing the signature of each generated instance.

4.5 Creation of the Associative/Semantic Network

In this step we create the semantic or associative network that stores the information on how different features are related. In case of NLP, the terms of a text are the features and the relations are defined by the co-occurence of terms within sentences. For MLP, the features are represented by instructions and the relations between instructions are based on the co-occurence of these instructions within chains. We note that although these relations are rather simple they already convey important information for further analysis (see Section 6 for possible improvements). The semantic network is generated in the following way:

NLP: For each sentence, we apply the following procedure: For each different term (sense) within the analyzed text corpus we create a node within the associative network. The edges between nodes and their weights are determined in the following way: All senses within a sentence are linked within the associative network. Newly generated edges get an initial weight of 1. Every time senses co-occur together, we increase the weight of their edges by 1. In addition, we store the type of connection for each edge. Examples for these types are noun-to-noun links, noun-to-verb links or adjective-to-adverb links. By using this information when applying SA algorithms, we are able to constrain the spreading of activation values to certain types of relations.

MLP: In machine code, sentences as we know them from text, do not exist. However we can find other techniques that separate instruction chains in a meaningful way:

- **Using branch operations to limit instruction chains:** For this method, we use branch operations such as **jmp**, **call** to identify the start/end of an instruction chain. We have already sucessfully applied this method in prior work ([3]).
- **Number of instructions:** We could simply define a window with size n that take n instructions from the instruction chains.

Regardless of the method for the extraction of instruction chains, the network is generated in the same way as for the text data.

4.6 Generation of Activation Patterns

Information about the relations between terms/instructions can be extracted by applying the SA-algorithm to the network. For each sentence/instruction chain, we can determine the corresponding nodes in the network representing the values stored in the data vector. By activating these nodes and applying SA, we can spread the activation according to the links and their associated weights for a predefined number of iterations. After this process, we can determine the activation value for each node in the network and represent this information in a vector - the *Activation Pattern*. The areas of the associative network that are activated and the strength of the activation gives information about which terms/instructions occurred and which nodes are strongly related.

4.7 Analysis of Activation Patterns

The *activation patterns* generated in the previous layers are the basis for applying supervised and unsupervised Machine Learning algorithms. Furthermore, we can implement semantic aware search algorithms based on SA.

Unsupervised Analysis: Unsupervised analysis plays an important role for the analysis of text, since it allows us to automatically cluster documents or instruction chains according to their similarity.

Search with Spreading Activation (SA): In order to search for related concepts within the analyzed text sources/instruction chains, we apply the following procedures:

1. The user enters the search query, which could be a combination of terms or instructions, a complete sentence or instruction chain or even a document containing multiple sentences or instruction chains.
2. We determine the POS/POC tags for every term/instruction within the search query.
3. Optionally, we now make use of an external knowledge source to find related terms/instructions and concepts for the terms/instructions in the query. For NLP such an external source could be WordNet [14] or Wikipedia. For MLP we could use reference documentation that describes all available instructions, their parameters and how these are related. An example for such a source is the Instruction Set Reference for Intel CPUs[7].
4. We activate the nodes corresponding to the terms/instructions of the search query and use the SA algorithm to spread the activation over the associative network.
5. We extract the *activation pattern* of the associative network and compare it to the document, sentence or instruction chain patterns that were extracted during the training process. The patterns are sorted according to their similarity with the search pattern.

External knowledges sources such as Wordnet can be quite useful for improving the quality of the search results. In order to highlight some of the benefits, we have the following example for text-analysis. Assuming, we execute a search query that contains the term **fruit**. After applying SA, we get the relations that were generated during the analysis of the text. However, these relations only represent the information stored within the text. The text itself does not explain that apples, bananas and oranges are instances of the term **fruit**. Therefore when searching for **fruit** we will not find a sentence that contains the term **apple** if the relation between these two terms is not established within the text. Thus, it makes sense to include external knowledge sources that contain such information. For NLP we can simply use Wordnet to find the instances of **fruit** and activate these instances in the associative network before applying SA. For MLP, such information could also provide vital information about the relations between instructions. In a similar way we could issue a search query that extends the search to all branch or arithmetic instructions.

Relations between Terms/Instructions: The trained associative network contains information about relations between terms/instructions that co-occur within sentences/instruction chains. By activating one or more nodes within this network and applying the SA algorithm, we are able to retrieve related terms/instructions.

[7] http://www.intel.com/products/processor/manuals/

5 The Real World – Example

In order to show the benefits of a possible malware analysis architecture based on MLP, we transform the existing NLP framework and apply it to payloads and shellcode encoders generated by the Metasploit framework. The Metasploit project is described in this way on the project website[8]: *Metasploit provides useful information to people who perform penetration testing, IDS signature development, and exploit research. This project was created to provide information on exploit techniques and to create a useful resource for exploit developers and security professionals. The tools and information on this site are provided for legal security research and testing purposes only.*

5.1 PTRACE Utility

For the lexical analysis of an arbitrary byte sequence we have developed a simple tool based on the PTRACE system call[9] on Linux.

- **Single stepping:** By utilizing PTRACE we are able to instruct the processor to perform single stepping. This enables us to inspect each executed instruction, its parameters and the CPU registers.
- **Execution of arbitrary byte sequences:** The utility follows each instruction chain until the bounds of the byte sequence are reached, the maximum number of loops is reached or a fault occurs. Whenever one of these conditions is fulfilled, the tool searches for a new entry point that has not already been executed. By applying these technique we are able to find executable instruction chains even if they are embedded in other data (e.g. images, network traffic).
- **Blocking of interrupts:** The analysis of the payloads and encoders generated by Metasploit is rather simple. In order to keep payloads from writing on the harddrive, we simple block all interrupts encountered by the tool.
- **Detection of self modifying code:** Such behavior is typical for a wide range of encoders/decoders that encode the actual payload in order to hide it from IDS systems. Typically the actual payload is decoded (or decrypted) by a small decoder. After this process the plain payload is executed. Since this decoding process changes the byte sequence, it is easy to detect when the decoder has finished and jumps into the decoded payload.
- **Dumping of instructions:** The tool makes use of the libdisasm library [10] to disassemble instructions. For each CPU step, we dump the instruction, its parameters and the category it belongs to.

5.2 Metasploit Data

Metasploit offers a command line interface to generate and encode payloads. We have used this interface to extract various payloads. Furthermore, we have

[8] http://www.metasploit.com/
[9] http://linux.die.net/man/2/ptrace
[10] http://bastard.sourceforge.net/libdisasm.html

encoded a payload with different shellcode encoders including the polymorphic shellcode encoder shikata-ga-nai. As dump format we have used the unsigned char buffer format. In order to apply MLP techniques we use the existing NLP architecture as basis and add or modify existing plugins for MLP processing:

- **Lexical Analysis:** For the extraction of the instruction chain we use our ptrace utility. The extracted chains contain the executed instructions, their parameters and the instruction category. We do not consider the parameters for further processing. The instruction chains are seperated into smaller chains by using control flow instructions (e.g. **jmp, call, loop**) as separator. In analogy to NLP, these sub instruction chains are considered as "sentences" whereas the whole payload/encoded payload is considered as "document".
- **Tagging:** Similar to a POS tagger, we can use a POC tagger for MLP. In this case this tagger uses the instruction category as tag. We consider all tags for further analysis and do not apply a filter.
- **Lemmatization:** Except for dropping the parameters, we do not employ further lemmatization operations.
- **Semantic network generation:** We apply the same semantic network generation process as used in the NLP architecture.
- **Activation pattern generation:** This is also based on the same process that is used for the NLP architecture. For each sub instruction chain (sentence), we activate the nodes corresponding to the instructions within the chain and spread the activation over the semantic network. We do not make use of any external knowledge source.
- **Analysis:** We show some examples for the analysis of the extracted/encoded payloads: Unsupervised clustering, finding relations between instructions and semantic search.

5.3 Relations

For text-analysis we often need to find terms that are closely related to a given term. An example from the e-Participation data analysis is shown in Figure 2(a). We use the term **vehicle** and extract the related terms from the the semantic network. Some examples for related terms are: **pollution**, **climate change**, **car**, **pedestrian** and **pedestrian crossing**. These relations are stored in the semantic network that was generated during the analysis of the text data. In MLP, we can apply exactly the same procedure. For the following example we want to find instructions that are related to **XOR** within the dataset consisting of subchains. In this case relation means that the instructions co-occur within the same chain. By issuing the query for **xor**, we get the following related instructions: **push, pop, inc, add, dec, loop**. These results can be explained by having a closer look on the decoding loops of various decoders (shikata-ga-nai, countdown, alpha-mixed) shown in Table 1. The utilitzation of these other instructions is necessary for reading the encoded/encoded shellcode, performing the actual decoding and writing the decoded shellcode back onto the stack. Due to the unsupervised analysis and the semantic network we are able to find these relations without knowing details about the underlying concepts.

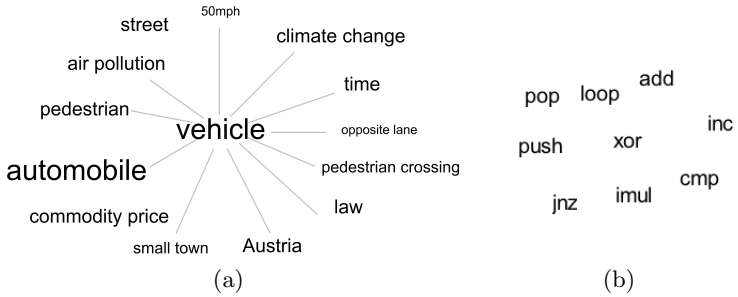

Fig. 2: NLP - Relation between terms (a) and MLP - relations between instructions (b)

5.4 Semantic Search

The previous example shows that due to the semantic network and the links within this network we are able to find relations between terms/instructions. These relations can also be used for executing semantic aware search queries. In order to highlight the benefits, we first present a simple example taken from text-analysis. Assuming we have two sentences[11] **A** and **B**: **A**: "Evidence suggests flowing **water** formed the rivers and gullies on the **Mars** surface, even though surface temperatures were below freezing" and **B:** "Dissolved minerals in liquid **water** may be the reason". When we search for the term **Mars** we obviously are able to retrieve sentence **A**. However, since sentence **A** talks about water on Mars, we also want to find sentence **B** that adds further details concerning the term **water**. Since the term **Mars** is not in sentence **B** we need to make use of the relations stored in the semantic network in order to include sentence **B** in the search results. The same procedure can be applied to MLP. For the following example we search for instruction chains that are related to the instruction **add**, which plays a role in various shellcode decoders. The results are shown in Table 1. Obviously, the algorithm returns decoders with an **add** instruction first, since these have the best matching pattern. However, at position 4 and 5 we also retrieve decoding loops of other decoders that do not make use of the **add** instruction. We are able to find these decoding loops since they use other instructions that are typical for such loops: **xor, sub, loop**. Due to the relations created by the decoding loops of shikata-ga-nai, **add** is linked with those and similar instructions. Thus, we are able to retrieve these other decoder loops that do not contain the **add** instruction, but have similar tasks.

5.5 Clustering

By clustering whole execution chains or sub chains (e.g. loops) into clusters, we are able to categorize different execution chains automatically. For unsupervised

[11] Take from the article: NASA Scientists Find Evidence for Liquid Water on a Frozen Early Mars, May 28th, http://spacefellowship.com

Result	Decoder	Instruction chain	Description
1	shikata-ga-nai	xor add add loop	Decoder
2	shikata-ga-nai	xor mov fnstenv pop mov xor add add loop	Decoder setup
3	nonalpha	pop mov add mov cmp jge	Decoder setup
4	fnstenv-mov	xor sub loop	Decoder
5	countdown	xor loop	Decoder

Table 1: Semantic search results for instruction **add**

clustering we apply the RGNG [16] cluster algorithm to the *activation patterns* of the subchain dataset. By choosing a rather simple model complexity, we retrieve 4 clusters: Cluster 1 primarily consists of the decoding loops of **alpha-upper** and **alpha-mixed**. Since both decoders have similar tasks (but not the same instruction chains), they are categorized within the same cluster. Cluster 2 and Cluster 4 contain the polymorphic decoding engines of **shikata-ga-nai**. By observing the instruction chains of those both clusters we see that Cluster 2 has chains based on **add** instructions whereas Cluster 4 consists of those chains that employ **sub** instructions. This is a perfect example why it could make sense to employ external knowledge to gain additional information about the analyzed instructions. In this case, **add** and **sub** could be mapped to arithmetic instructions which would result in the categorization within the same cluster. Cluster 3 contains chains related to decoding engine setup and the necessary preparations for calling an interrupt (typically the payload itself).

6 Conclusions and Outlook

In this paper we present a MLP architecture for malware analysis. This architecture is the result of adopting an existing NLP architecture to the analysis of machine code. We map existing NLP modules to MLP modules and describe how established NLP processes can be transferred to malware analysis. In order to show some of the possible applications for such an MLP architecture, we analyze different shellcode engines and payloads from the Metasploit framework. The presented malware architecture can be seen as the first step in this direction. There are further promising techniques, which would increase the capabilities and the quality of the analysis process:

- Improved lexical parsing in order to allow the identification of more complex structures such as loops, preparations for interrupts, etc.
- Due to improved lexical parsing, more relations could be stored in the semantic network, which would enable more detailed or focused analysis processes.
- High level interpretation of the underlying machine code.
- Extending the MLP framework to high level languages such as Javascript.

All of these suggested improvements have corresponding elements within NLP and are partly already solved there. This means, that we might be able to apply some of these techniques directly in MLP or adapt them for MLP. As next step we will identify more suitable NLP techniques and adopt them to MLP

modules. Finally, we especially want to thank P. N. Suganthan for providing the Matlab sources of RGNG [16].

References

1. Teufl, P., Payer, U., Parycek, P.: Automated analysis of e-participation data by utilizing associative networks, spreading activation and unsupervised learning. (2009) 139–150
2. Payer, U., Teufl, P., Kraxberger, S., Lamberger, M.: Massive data mining for polymorphic code detection. In Vladimir Gorodetsky, Igor Kotenko, V.S., ed.: Computer Network Security. Volume 3685 of Lecture notes im computer science., Springer (2005) 448 – 453
3. Payer, U., Teufl, P., Lamberger, M.: Hybrid engine for polymorphic code detection. In Klaus Julisch, C.K., ed.: Detection of intrusions and malware, and vulnerability assessment. Volume 3548 of Lecture notes im computer science., Springer (2005) 19 – 31
4. SunbeltSoftware: (Cwsandbox - automatic behavior analysis of malware)
5. Norman: (Norman sandbox: A virtual environment where programs may perform in safe surroundings)
6. Vasudevan, A., Yerraballi, R.: Cobra: Fine-grained malware analysis using stealth localized-executions. Security and Privacy, IEEE Symposium on **0** (2006) 264–279
7. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: ICISS '08: Proceedings of the 4th International Conference on Information Systems Security, Berlin, Heidelberg, Springer-Verlag (2008) 1–25
8. Microsoft: (Natural language processing group: Redmond-based natural language processing group)
9. Stanford: (Natural language processing group: Natural language processing group at stanford university.)
10. Alias-i: (Lingpipe: A suite of java libraries for the linguistic analysis of human language)
11. Tsatsaronis, G., Vazirgiannis, M., Androutsopoulos, I.: Word sense disambiguation with spreading activation networks generated from thesauri. In Veloso, M.M., ed.: IJCAI 2007. (2007)
12. Quillian, M.R.: Semantic memory. MIT Press, Cambridge, MA (1968)
13. Crestani, F.: Application of spreading activation techniques in information retrieval. Artificial Intelligence Review **11** (1997) 453–482
14. Fellbaum, C.: WordNet: An Electronic Lexical Database (Language, Speech, and Communication). The MIT Press (1998)
15. Kozima, H.: Similarity between words computed by spreading activation on an english dictionary. In: EACL. (1993) 232–239
16. Qin, A.K., Suganthan, P.N.: Robust growing neural gas algorithm with application in cluster analysis. Neural Netw. **17** (2004) 1135–1148
17. Klein, D., Manning, C.D.: Fast exact inference with a factored model for natural language parsing. In: In Advances in Neural Information Processing Systems 15 (NIPS, MIT Press (2002) 3–10