Antti Niiranen

Machine learning based ISA detection for short shellcodes

Master's Thesis in Information Technology

June 20, 2021

University of Jyväskylä

Faculty of Information Technology

Author: Antti Niiranen

Contact information: antti.n.niiranen@student.jyu.fi

Supervisor: Andrei Costin

Title: Machine learning based ISA detection for short shellcodes

Työn nimi: Lyhyen hyökkäyskoodin käskykanta-arkkitehtuurin havaitseminen koneoppimiseen perustuvan sovelluksen avulla

Project: Master's Thesis

Study line: Computer Science, Cyber Security

Page count: 77+17

Abstract: Shellcodes are often used by cybercriminals in order to breach computer systems. Code injection is still a viable attack method because software vulnerabilities have not ceased to exist. Typically these codes are written in assembly language. Traditional method of analysis has been reverse engineering, but as it can be difficult and time-consuming, machine learning has been utilized to make the process easier. A literature review was performed to gain an understanding about shellcodes, artificial intelligence and machine learning. This thesis explores how accurately a state-of-the-art machine learning ISA detection tool can detect the instruction set architecture from short shellcodes. The used method was experimental research, and the research was conducted in a virtual environment mainly for safety reasons. Using three different sources which were Exploit Database, Shell-Storm and MSFvenom, approximately 20000 shellcodes for 15 different architectures were collected. Using these files, a smaller set of shellcodes was created in order to test the performance of a machine learning based ISA detection tool. When limitations were identified, it was noted that the test set may not be diverse or large enough. Nevertheless, with this set it was possible to gain an understanding on how the program currently handles shellcodes. The study found that with the current training, the program is not able to reliably detect ISA from the shellcodes of the database. Two different detection options were used and they both achieved the accuracy of approximately 30%. The different classifiers were tested as well and random forest had the

best performance.

Keywords: Artificial intelligence, machine learning, shellcode, code analysis

Suomenkielinen tiivistelmä: Hyökkäyskoodi (engl. shellcode) on usein käytössä kyberrikollisuudessa, kun tarkoituksena on tunkeutua erilaisiin tietoteknisiin järjestelmiin. Koodiinjektio on yhä toimiva hyökkäysmenetelmä, sillä ohjelmistohaavoittuvuudet eivät ole kadonneet mihinkään. Tyypillisesti tällainen koodi kirjoitetaan konekielellä. Perinteisesti näitä hyökkäyskoodeja on analysoitu takaisinmallintamalla, mutta menetelmän vaikeuden takia on ryhdytty turvautumaan koneoppimiseen, jotta prosessista tulisi helpompi. Tutkielmassa tehdyn kirjallisuuskatsauksen avulla hankittiin tietoa hyökkäyskoodeista, tekoälystä ja koneoppimisesta. Tässä tutkielmassa selvitettiin, kuinka tarkasti viimeisintä tekniikkaa edustava koneoppimispohjainen sovellus havaitsee hyökkäyskoodin käskykanta-arkkitehtuurin. Tutkimus oli kokeellinen ja se suoritettiin virtuaaliympäristössä muun muassa turvallisuuden takia. Työssä rakennettiin reaalimaailmaan perustuva hyökkäyskooditietokanta, joka sisältää noin 20000 hyökkäyskooditiedostoa 15 eri arkkitehtuurille. Koodit hankittiin kolmesta eri lähteestä, jotka ovat Exploit Database, Shell-Storm ja MSFvenom. Näistä koodeista koostettiin pienempi joukko testaamista varten. Tutkimuksen rajoituksia pohdittaessa todettiin, että testitietokanta saattaa olla liian suppea, mutta sen avulla kuitenkin pystyttiin kartoittamaan sovelluksen tämänhetkinen toiminta. Testeissä selvisi, että sovellus ei tällä hetkellä kykene havaitsemaan hyökkäyskoodin käskykanta-arkkitehtuuria riittävällä tarkkuudella. Kahta eri skannausasetusta testattiin, joista molemmat saavuttivat noin 30% tarkkuuden. Sovelluksen luokittelijat testattiin myös, niistä satunnaismetsä toimi parhaiten.

Avainsanat: Tekoäly, koneoppiminen, haittakoodi, hyökkäyskoodi, koodianalyysi

Glossary

Architecture	The unique encoding of a computer's instructions (Clemens 2015, S157). A processor architecture comprises an instruction set and knowledge about registers (Sweetman 2007, 29).
Endianness	Byte-order, i.e. the expected order of multi-byte data when it is in memory (Clemens 2015, S157). Endianness is divided into little-endian and big-endian. (Sweetman 2007, 280-281). Little-endian is an order where bytes are stored from least significant to most significant and big-endian means storing bytes using the opposite order ("Endianness" 2020).
ISA	Instruction Set Architecture. It provides an interface between hardware and software, and it defines the interface between the basic machine instruction set and the runtime and input/output control (Fox and Myreen 2010, 243; Abd-El-Barr and El-Rewini 2005, 1).
Word size	The maximum amount of bits that a processor can handle at a time, for example 32 or 64 bits (Clemens 2015, S158).
IoT	Internet of Things

List of Figures

List of Tables

Contents

1 Introduction

In cybercrime, attackers often use shellcode to compromise various computer systems and other such devices. Potentially an attack like this can have devastating effects because if successful, it can enable cybercriminals to obtain a shell connection, which is the highest level privilege available, to the targeted device (Brinda and George 2016, 310). Even the term shellcode originally refers to gaining a shell connection, although currently shellcodes are written for other purposes as well. These shellcodes are pieces of bytecode, typically written in assembler, which are used as the payload in the exploitation of software vulnerabilities (Anley et al. 2007, 41).

The ability to correctly analyze shellcode is crucial, and the two main methods for this are static and dynamic analysis (Sikorski and Honig 2012, 2). For shellcodes especially, manual reverse engineering is a typical method of analysis, though it is time-consuming and requires significant expertise (Borders, Prakash, and Zielinski 2007, 501). Regardless of the method, the first phase of analysis is to correctly detect the Instruction Set Architecture of the opcodes within the shellcode, but this information is not always readily available (Kairajärvi, Costin, and Hämäläinen 2020b). However, it is required and crucial information because without it, analysts are not able to correctly decode the instructions of the shellcode and find out what it does (Clemens 2015, S157). In addition, human error is a factor to be noted in the analysis process. Incorrectly identifying the Instruction Set Architecture of binary code caused approximately 10% of failures in the firmware analysis of IoT devices (Kairajärvi, Costin, and Hämäläinen 2020b).

This requirement of correctly identifying the Instruction Set Architecture of the shellcodes along with the fact that the amount of new processor architectures is on the rise because of the constantly growing number of various IoT devices has created a need for a state-of-the-art solution for performing the initial detection of Instruction Set Architecture (Kairajärvi, Costin, and Hämäläinen 2020b). Thus machine learning has been utilized in the attempt to solve this problem. Researchers in this field seem to agree (Borders, Prakash, and Zielinski 2007; Clemens 2015; Kairajärvi, Costin, and Hämäläinen 2020b) that these machine learning based tools are promising and will help analysts to be more efficient and accurate in their

work.

To the best knowledge, the machine learning based state-of-the-art tool that is under examination in this thesis has not previously been tested with short shellcodes in this scope. Therefore, this thesis will provide new research data and shellcode database might be useful to other researchers as well. These are the two main contributions of this thesis. In addition, the research topic is current and relevant. As Chen et al. (2016, 107) state, a code injection attack is an old method, but still viable and popular because software vulnerabilities have not ceased to exist.

1.1 Research questions

The goal of this thesis is to answer these two main research questions:
RQ1: How to create a significant and representative real-world database of shellcodes?
RQ2: How accurately can a machine learning based ISA identification system detect the correct CPU architecture and endianness from short shellcodes?
And the following sub-questions:
SQ1: How to automate the creation of the shellcode database?
If the accuracy of the detection is not satisfactory:
SQ2: How can machine learning based ISA identification systems be improved?
SQ3: Which different one byte combinations MSFvenom accepts or rejects as bad characters, and what are the most problematic bytes?

1.2 Organization

This section explains how this thesis is organized after chapter 1. Chapter 2 gives some insight into shellcodes. The chapter describes what shellcodes are, how they are written, and gives some examples on how shellcodes are used in cyber attacks. Finally, the topic of shellcode analysis is explored.

Chapter 3 briefly discusses the broad topic of artificial intelligence. First, the chapter de-

scribes what artificial intelligence is and then it discusses the various subfields of artificial intelligence. The purpose of this chapter is not to dive very deep into this topic, but rather explain how machine learning fits into this context and that it is just one subfield of artificial intelligence.

Chapter 4 explores machine learning. The chapter begins by describing what machine learning is and then moves on to discuss the different machine learning methods. Afterwards, the machine learning algorithms that are relevant to this thesis are described and finally, the topic of machine learning based code analysis is explored. This final section of the chapter concentrates on studying previous research in this field.

Chapter 5 explains how the research was conducted, what methods were used and how the research data was gathered. Chapter 6 presents the results of the research, chapter 7 discusses them, identifies limitations and explores opportunities for further research. Finally, chapter 8 concludes this thesis.

2 Overview of shellcode

Shellcode is a piece of bytecode which is used as the payload when attempting to exploit software vulnerabilities. A piece of shellcode can also be seen as an instruction set, which is injected and executed by the targeted program. Originally, the purpose of shellcode was to execute a shell, hence the name shellcode, but recently the definition has changed or broadened. Currently shellcodes are written for other purposes as well, such as creating files, proxying system calls, directly manipulating registers, or changing how a program functions. Shellcodes are typically written in assembler and then translated into hexadecimal opcodes, which are actual machine instructions. If a piece of shellcode has been written with a high-level language, injecting it will most likely fail, because such shellcode may not execute cleanly (Anley et al. 2007, 41-42; Foster and Price 2005, 631).

Assembly is a low-level programming language and therefore it enables creating programs that are fast and very small. However, assembly has some drawbacks as well. For example, assembly code is dependent on the processor, so it is difficult to port it to other processors as well as other operating systems even if they are running on the same processor (Foster and Price 2005, 336).

2.1 Writing shellcode

When writing shellcode, the most important thing to take into consideration is that the code should be as short and simple as possible. As shellcode will be injected into vulnerable input areas which are n bytes long, the code must always be shorter than n. Otherwise the process will fail, because the shellcode does not fit into the input area. The usual target for injection is a buffer which is set aside for user input and most often the type of this buffer is a character array (Anley et al. 2007, 44, 48).

For the shellcode to be injectable and successful, it must not contain any nulls which looks like this:

$$0x00$$

The purpose of these null characters is to terminate string and thus, if they are present in the

injectable shellcode, it will fail. When writing shellcode, it is essential to figure out ways to remove any nulls that might exist in the opcodes or use non-null opcodes to replace the ones that contain nulls. There are two common methods which can be used to achieve null removal. In the first method, the assembly instructions whose purpose is to create nulls are replaced with instructions that do not create them. In the second method, nulls are added at runtime with instructions that do not create them. Because of this,the second if more difficult. Another thing that makes the second method more difficult is the requirement of knowing the exactly where the shellcode is located in memory. In addition to making shellcode injectable, in best case scenarios removing null opcodes also makes the code significantly shorter (Anley et al. 2007, 26, 48-50). Other examples of bad bytes are:

$$0x0a\ 0x0d\ 0xff$$

As stated before, 0x00 is a null byte (Anley et al. 2007, 48), 0x0a is a new-line character, 0x0d is a carriage-return character and 0xff is a form-feed character (Foster and Liu 2006, 397).

The main objective for writing shellcodes is to alter the original functionality of the target program. One example of this kind of manipulation is to make the program perform a system call, or syscall in short. System calls are the interface which lies between user mode and protected kernel mode. The function of system calls is to enable direct access to kernel and operating system-specific functions. The most basic system call is exit() which ends the current process. In order to create a piece of shellcode which executes the exit() system call, one could write this system call in C, compile it into binary and then disassemble it. The actual assembly instructions can be examined from this disassembly, and also it is important to understand what these instructions do as they are necessary pieces when creating the shellcode for the exit() system call. After this, if possible, the shellcode can be cleaned up in order to make it smaller and the next step is to acquire the hexadecimal opcodes from the assembly. These opcodes can then be placed into a character array and then a C program can be created which executes the string. Below is an example of how this process looks like in the code:

From instructions:

$$mov\ \$0x1,\ \%eax$$

To opcodes:

$$b8\ 01\ 00\ 00\ 00$$

To character array in C:

$$char\ shellcode[\,] = "\backslash xb8\backslash x01\backslash x00\backslash x00\backslash x00";$$

In any case, now the exit() shellcode should be finished and ready for testing (Anley et al. 2007, 42-48).

Intrusion detection systems are of course equipped to deal with shellcodes. However, by using obfuscation and end-to-end encryption for shellcodes to encrypt data communication, it is possible to bypass intrusion detection (Anley et al. 2007, 299-311).

2.2 Shellcode-based attacks

Shellcode-based attacks are common, and it is possible to execute them in various ways. The purpose of this section is not to discuss them all, but rather give some examples of perhaps some of the most typical ways to use shellcode for malicious purposes.

2.2.1 Buffer overflow

One method to inject shellcode is to successfully perform a buffer overflow attack. A buffer overflow is probably the most widely known software security vulnerability. It happens when more data is copied to a buffer, which has been allocated a specific amount of storage space, than it can handle. These overflows can be divided into two classes: heap overflows and stack overflows. Buffer overflows are a consequence of badly developed software programs. The programming mistakes that result in buffer overflows can be either small or complex, and these mistakes can be found in both local and remote programs. (Foster et al. 2005, 3-4, 18).

There are several ways to detect and prevent buffer overflows. Analyzing source code is one of these methods. Source code analysis can reveal the size of the copied data and the size of the buffer where the data will be copied. An overflow will occur, if the size of the data is

larger than that of the buffer. However, source code analysis can take a lot of time and it is not easy as it requires a lot of programming skills and knowledge about software security. There are also some applications that scan the code and perform security analysis which can produce reliable results (Foster et al. 2005, 404-405, 450-452). Some programming languages such as C and C++ are more vulnerable to buffer overflows, so using them should be avoided if possible. This is because several functions in these languages that have access to memory and can manipulate it do not perform bounds checking. Therefore, these functions can overwrite the allocated buffers they are used on. These languages have bounded functions as well, but even with those incorrect usage can result in vulnerabilities (”Buffer Overflow | OWASP” 2020). As buffer overflows are a direct consequence from programming mistakes, allowing programmers to concentrate on using their core skills instead of countless and sometimes pointless software development methodologies might help to reduce programming mistakes and thus, reduce overflows as well (Shaw 2020).

2.2.2 Shellcode embedded documents

One, quite often used attack vector is to embed shellcode in documents. Formats such as portable document format, or PDF, Word, and PowerPoint are popular among attackers because they are considered easy to exploit. In addition, the malicious components in documents such as these are not always easy to detect, and many security mechanisms can have difficulties with them (Brinda and George 2016, 310). The PDF format describes the text formatting, graphics hypertext, bookmarks, and multimedia elements. In addition to text, PDF documents can contain standalone scripts, images, and other multimedia elements. The PDF format is appealing to attackers because of the dynamic data which allows them to embed shellcode in these documents. The objective of shellcodes in these cases could be, for example, to open a backdoor or download malware to the victim's system (Brinda and George 2016, 310-311). Microsoft Office uses and holds the title for the Object Linking and Embedding Technology which enables embedding and linking to the documents and using different sources to add various data components to the documents. It creates a compound binary file, or CBF in short, which stores data in multiple streams and these streams in turn are held in different storages. The storages resemble subdirectories and the streams resem-

ble files. These data streams can also contain malicious shellcode (Brinda and George 2016, 310-312).

2.3 Shellcode analysis

Static and dynamic analysis are the two cardinal methods for discovering vulnerabilities and analyzing shellcodes and malware. In static analysis malicious objects are observed without executing them, and in dynamic analysis they are analyzed after execution, in a running state (Sikorski and Honig 2012, 2). For shellcodes, manual reverse engineering is a common method of analysis. Successful reverse engineering can reveal important information, such as the purpose of the exploit payload, about the functionality of shellcodes. This information can be essential in creating and implementing defense mechanisms for the exploit. However, the drawback is that manual reverse engineering can be cumbersome, time-consuming, and challenging as it requires serious expertise (Borders, Prakash, and Zielinski 2007, 501).

The execution of shellcodes is not similar to that of normal executables as shellcodes are often only binary chunks of data. This means that loading and running shellcodes in a debugger can cause problems because the user might have to provide input during the loading process and select the correct processor architecture as well (Sikorski and Honig 2012, 408). Selecting the correct architecture is crucial. For example, in the IoT firmware analysis approximately 10% of analysis failures were caused by incorrect identification of the binary code's instruction set architecture. If an incorrect architecture is selected, opcodes will be misread and this leads to errors in the analysis process (Kairajärvi, Costin, and Hämäläinen 2019).

3 Overview of artificial intelligence

According to Garnham (1987, 2) artificial intelligence is the study of intelligent behavior, but Kaplan (2016, 1) points out that artificial intelligence has several proposed definitions and while there is not one single clear definition for this concept, the general consensus is that artificial intelligence means creating computer programs or machines that are able to perform in a way which humans perceive as intelligent. Garnham (1987, 2) agrees and adds that another purpose for artificial intelligence is to understand human intelligence. Artificial intelligence has many subfields, though they all aim to address similar problems. In addition to machine learning, some of the more notable subfields are robotics, computer vision, speech recognition and natural language processing (Kaplan 2016, 49).

Robotics aims to build machines that perform various physical tasks. Usually the focus in robotics is to build machines that can perform specialized and complex tasks instead of general ones. One clear advantage of machines is that they can work in conditions and perform tasks that are too dangerous for humans (Kaplan 2016, 49-54).

Computer vision aims to equip computers with the ability to interpret visual images, or in human terms, to see. Early work in this field concentrated on creating algorithms that used specialized knowledge of visual images and descriptions of objects to search meaningful elements. In the modern work of this field machine learning is used in order to build models of objects from large collections of examples. Mainly computer vision technology is used to solve real-world problems that are visual by nature to gather information. One major application of this technology are numerous real-world problems which involve identifying and locating objects of interest in a specified setting. Another major application is related to information. Currently data is mostly in digital form and has become more visual which enables computer vision technology to begin managing this data automatically (Kaplan 2016, 54-57).

Speech recognition is probably one of the most challenging subfields because processing speech is much more complex a task than processing visual images or written language. There are many factors which make speech recognition difficult for computers. For exam-

ple, speech must be separated from any background noise and the meaning of spoken words is affected by elements such as volume, tone, and pitch. In addition, some words sound the same when spoken out loud. In order to recognize speech and figure out its' meaning, machines must correctly interpret all these elements and handle possible distractions as well. However, recently modern machine learning techniques have enhanced the precision and utility of speech recognition systems because it is possible to collect and analyze large quantities of speech samples with these techniques. Currently state-of-the-art speech recognition systems are not nearly as capable as human speakers, but they have real utility in limited domains (Kaplan 2016, 57-60).

Natural language processing observes the interactions between natural human languages and computer languages. The old approach to natural language processing was to codify natural human language to word categories and sentence structure. The aim was to imitate the generally accepted view of languages obeying syntactic rules. However, this approach proved to be too inflexible because human languages and their usage is complex, and formal grammatical analysis is not enough to capture what is really going on. More recently the approach to natural language processing has changed. Now machine learning, especially statistical machine learning methods are used to analyze human languages. This analysis enables computers to solve practical language-related problems such as translating from one language to another, answering question from databases of facts and generating summaries of documents. With large amounts of examples, it is possible for computers to work with languages reasonably well even without knowing the meaning of the texts (Kaplan 2016, 60-64). Areas and applications of artificial intelligence can be viewed from figure 1 below.

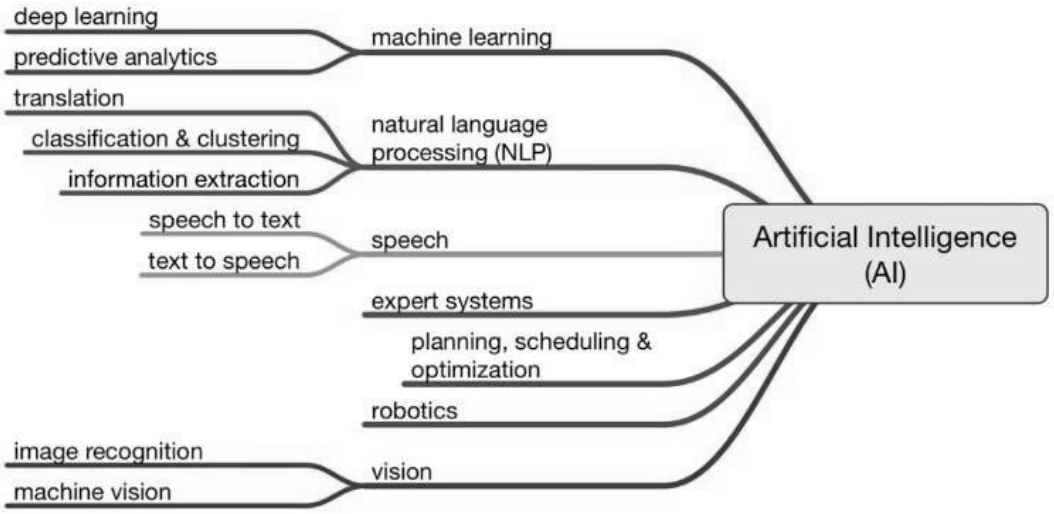

Figure 1. Areas and applications of artificial intelligence (Atlam, Walters, and Wills 2018)

4 Overview of machine learning

Machine learning is another major subfield of artificial intelligence. The objective of machine learning is to enable machines to skillfully perform and complete the tasks assigned to them by using intelligent software (Mohammed, Khan, and Bashier 2017, 4). This field focuses on developing computer systems that have the ability learn from provided data. These systems may then automatically learn and improve, and with enough time and experience they might develop models which can be used to predict outcomes of problems and give answers to questions based on previous learning (Bell 2014, 2). In other words, in machine learning the aim is to answer how computers can learn specific tasks such as recognition, categorization and even helping specialists of different fields to make decisions (Fernandes de Mello and Antonelli Ponti 2018, 1).

There are many different learning algorithms that can be used in machine learning, and the required output defines which one should be used. These algorithms can be placed in one of these two learning types: unsupervised learning or supervised learning (Bell 2014, 2-3). However, the performance of machine learning models and algorithms severely depend on the representation of the data provided to them. This also means that the choice of representation significantly impacts the performance of the algorithms (Goodfellow, Bengio, and Courville 2016, 3). According to Mohammed, Khan, and Bashier (2017, 7), in total there are four different learning types which can be seen in the figure 2 below along with their required data.

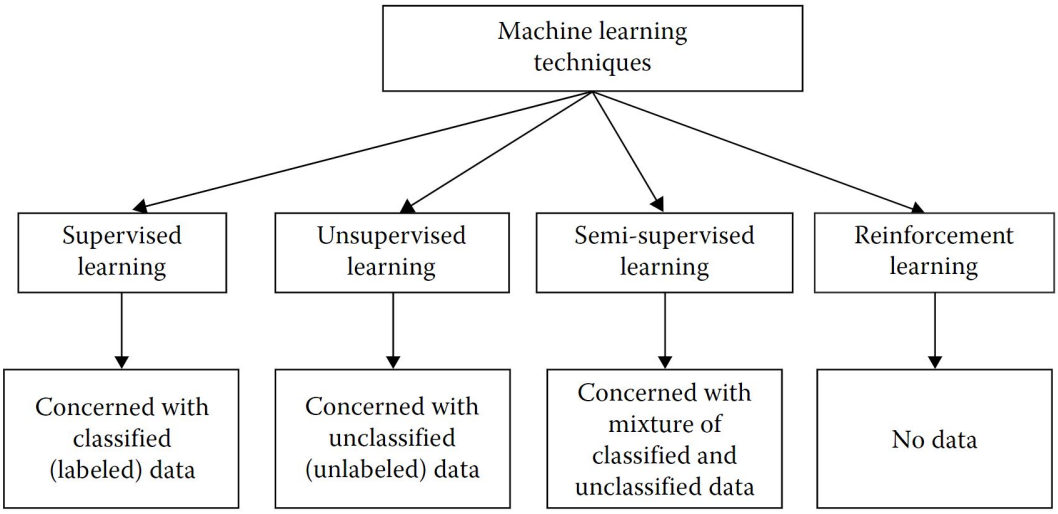

Figure 2. Different machine learning techniques and the type of data they require (Mohammed, Khan, and Bashier 2017, 7)

There is also a machine learning method known as deep learning which is not to be confused with the four methods described in figure 2. Deep learning will be discussed further in section 4.5, but for now, it is a subfield of machine learning which uses many layers of information-processing stages in hierarchical architectures to perform pattern classification and representation learning (Deng 2014).

4.1 Supervised learning

In supervised learning the work is done with a set of labeled training data. Each piece of training data there contains two objects, one for input and one for output (Bell 2014, 3). These objects contain labels or tags and together they form a training example. Thus, training data consists of training examples. A label or a tag from an output object is the explanation of its' respective input example from an input object. If labeling does not exist for an input object, it is unlabeled data. The output object consists of labels for every training example which are present in the training data. A supervisor provides these labels and usually the supervisor is a human being, but labeling can be done by machines as well. Human labeling costs more and machine labeling has higher error rates, so currently human labeling is considered superior. In addition, manually labeled data is a reliable source for supervised

learning (Mohammed, Khan, and Bashier 2017, 7-8).

When using supervised learning, there are some issues that need to be taken into consideration. One example of these issues is the bias-variance dilemma, or bias-variance tradeoff. High bias models contain restricted learning sets while high variance models tend to be more complex, and they learn with complexity against noisy training data (Bell 2014, 3). The optimal approach would be to have both low bias and low variance, but it is not possible and hence a tradeoff has to be made between them: if one is improved, most likely the other will worsen. High bias can be fixed by getting more features, making the model more complex or by changing the model. High variance can be fixed by getting more data or by decreasing the complexity (Richert and Coelho 2013, 102-3).

Another thing to take into consideration is the class imbalance problem. It is a problem where one class is heavily represented and another one is much smaller in proportion. The class imbalance problem is a significant issue because in some cases it severely hinders the performance of any learning method which assumes a balanced class distribution (Japkowicz 2000).

4.2 Unsupervised learning

In unsupervised learning, training data or supervisors are not present. Therefore, there is only unlabeled data and there can be many reasons for this. For example, it may be a lack of funds to pay for manual labeling, or the data itself can be inherent (Mohammed, Khan, and Bashier 2017, 9-10). It is up to the machine learning algorithms to find hidden patterns in this data. There are no correct or false outcomes in this type of machine learning, the algorithms are just run in order to see what results and patterns occur in the data (Bell 2014, 3-4). Nowadays, when data is collected at an unprecedented rate, it is important to get something from this massive amount of data without supervisors (Mohammed, Khan, and Bashier 2017, 9).

Clustering is a common unsupervised machine learning method. This method aims to segment data into specific groups that share similar characteristics. In other words, this is how the classification is done in unsupervised learning. Clustering has wide applications and it

is used, for example, in social media analysis, market research, law enforcement and IoT related analysis. Generally clustering is useful when it is necessary to group multivariate data into distinctive groups (Bell 2014, 161-64).

4.3 Semi-supervised learning

In semi-supervised learning, the used data is a combination of classified and unclassified data. Using this mixture, a suitable model for the classification of data is generated. In most cases the amount of unlabeled data is abundant, and the amount of labeled data is much scarcer. Thus, generally the approach here is to combine these large quantities of unlabeled data with the much smaller amounts of labeled data. The goal is to generate a model that can make more accurate predictions than a model which has been created by only using labeled data. It can be said that human learning resembles semi-supervised learning. In human terms, the environment provides unlabeled data and a supervisor, a teacher for example, provides labeled data by pointing out objects in the environment and giving them names, i.e. labels (Mohammed, Khan, and Bashier 2017, 10-11).

4.4 Reinforcement learning

In reinforcement learning the approach is to use observations gathered from interacting with the environment to take actions which aim at maximizing rewards or minimizing risks. When producing intelligent programs, also known as agents, reinforcement learning goes through a process which can be divided into four stages. In the first stage, the program observes the input state. Then, in the second stage the program performs an action by using a decision-making function. In the third stage, after the action of the second stage is completed, the program receives feedback from the environment in the form of reward or reinforcement. And finally, in the fourth stage, the state-action pair information about the feedback received during the third stage is saved. After this process is completed, the saved information can be used to adjust the action of any stored state-action pair. This procedure can enhance the program's decision-making capabilities. (Mohammed, Khan, and Bashier 2017, 11).

4.5 Deep learning

It is possible to solve various artificial intelligence tasks by designing the correct set of features to extract for the task at hand, and then giving these features to a machine learning algorithm. But in some cases, it is not easy to figure out which features could be useful in completing the given task. There is an approach known as representation learning, which is used to solve these kinds of problems. Representation learning is a technique in which machine learning is used to discover both the mapping from representation to output and the representation itself (Goodfellow, Bengio, and Courville 2016, 4).

However, variations in raw data can cause problems for many real-world artificial intelligence applications, because for computers it is difficult to understand the meaning of raw sensory input data. For example, a red object can appear darker, almost black at night, and in most cases the shape of the silhouette of objects depend on the viewing angle. Usually artificial intelligence applications require human touch to examine these variations and discard those that are not needed. In addition, extracting abstract features such as the ones discussed above from raw data can be a challenging task, and identifying these kinds of variations require sophisticated, close to human-level understanding of the data. Representation learning does not seem to be effective when obtaining representations and solving the original problem are almost equally difficult (Goodfellow, Bengio, and Courville 2016, 5-6).

Deep learning attempts to solve this pivotal problem in representation learning using multiple other representations that are much more simple by nature. When applying deep learning methods, computers can use simple concepts to build more complex ones. One demonstration which clarifies this process is how deep learning can be used to construct a complete image from multiple simpler pieces. This is achieved by combining simple concepts into more complex ones until the full image is constructed (Goodfellow, Bengio, and Courville 2016, 5-6).

4.6 Machine learning algorithms

Some of the most common machine learning algorithms, or those relevant to this thesis, will be introduced in the following subchapters. These algorithms are either supervised or

unsupervised learning algorithms.

The supervised algorithms that will be discussed are rule-based classifiers, decision trees, naïve Bayesian classifiers, *k*-nearest neighbor classifiers, neural networks, linear discriminant analysis, and support vector machine. Supervised learning algorithms can be placed in either one of these two categories: regression or classification (Mohammed, Khan, and Bashier 2017, 8, 35). In classification, the aim is to assign an unknown pattern to known classes, and in regression, the goal is to solve a curve fitting problem, or to attempt to predict continuous values such as stock market prices (Theodoridis 2015, 2-4).

Regarding unsupervised learning, there exists a wide variety of algorithms especially for clustering, so sometimes the most sensible method to choose the correct algorithm is experimentation (Bell 2014, 162). Some examples of unsupervised learning algorithms according to Mohammed, Khan, and Bashier (2017, 129) are *k*-means clustering, Gaussian mixture model, hidden Markov model and principal component analysis in context of dimensionality reduction. However, only the *k*-means clustering algorithm will be discussed in this thesis.

4.6.1 Decision trees

Decision trees are tree structures which can resemble flowcharts. They consist of non-leaf nodes, leaf nodes, branches, and a root node. A non-leaf node stands for a test on an attribute, a leaf node holds a class label, a branch represents a result of the test and finally. The topmost node is the root node and it represents the attribute which has the main role in classification. Decision trees classify data in datasets, and they do this by flowing through a query structure from root node to leaf node. Below, in figure 3 a classic decision tree model is shown, and rectangles represent non-leaf nodes and ovals represent leaf nodes. The point of this decision tree is to indicate how likely it is for a customer to purchase a certain product (Han and Kamber 2011, 330-31; Mohammed, Khan, and Bashier 2017, 37).

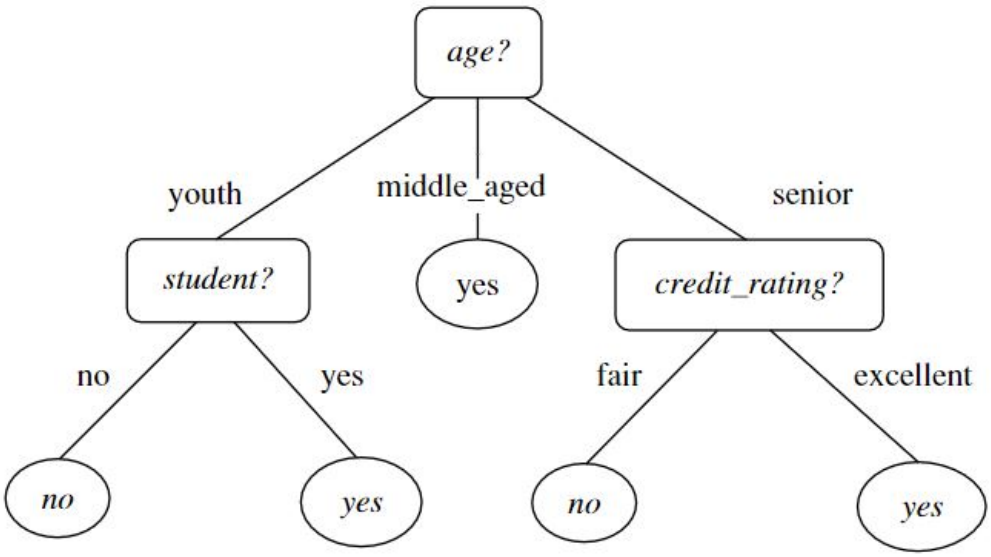

Figure 3. An example of a decision tree (Han and Kamber 2011, 331)

4.6.2 Random forest

Random forest is an ensemble method which consists of many individual decision trees. Every classifier in this ensemble is a decision tree. When generating individual decision trees, randomly selected attributes are used at every node in order to determine the split. In addition, by performing the random feature selection, the decision tree models are diversified. Once the ensemble is created, a voting system is used to combine the predictions of the trees and the returned class is the one that is the most popular. Random forest has the ability to work efficiently with very large datasets, the kind of with which other models may fail. The reason for this ability is that the ensemble does not utilize the full feature set, instead it only uses a small, randomly generated part of it (Han and Kamber 2011, 382-83; Lantz 2013, 344-45). Using these ensemble methods can yield more accurate results when compared to using just their base classifiers, and based on this it can be assumed that random forests might enhance the overall accuracy of decision trees (Han and Kamber 2011, 378, 386).

Random forest is a strong model because of its' reliable performance and versatility. In addition, random forest can deal with noisy or missing data as well as massive datasets, and it also is able to select the most important features from these massive datasets. On the

downside, random forest is not as easy to interpret as, for example, a decision tree. Also, tuning this model to the data might require some effort (Lantz 2013, 345).

4.6.3 Rule-based classifiers

In rule-based classifiers sets of IF-THEN rules are used for classification. Here, IF represents the rule antecedent or precondition, and it is composed of at least one attribute test such as age or occupation, or both. If there are more than one rule antecedent, they are combined with the logical AND operator. THEN, the latter part of this rule pair, is the rule consequent and it comprises a class prediction. For example, the purchase behavior of customers can be predicted here. A rule antecedent is satisfied when the conditions in it hold true. Rules can also be assessed by their coverage and accuracy. One method to create these rules is to extract them from decision trees so that every path between a root node and a leaf node becomes a rule. This can be a useful approach if the decision tree is very large (Han and Kamber 2011, 355-57; Mohammed, Khan, and Bashier 2017, 53). An example rule R, according to Han and Kamber (2011, 355), can be:

$$R: IF\ age = middle-aged\ AND\ working = yes\ THEN\ purchases\_product = yes$$

4.6.4 Naïve Bayes classifiers

Bayesian classifiers are statistical classifiers that have the ability to predict class membership probabilities. Bayesian classification is based on Bayes' theorem:

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)}$$

In this equation, H represents a hypothesis and X represents data. For example, H can be a hypothesis about X belonging to a certain class.

P(H|X) represents the posterior probability of H conditioned on X. For example, X could be a customer who belongs to certain age and income groups, and H could be that this customer buys a certain product. In this case P(H|X) would represent the probability of customer X purchasing a product given that this customer's age and income are known.

P(X|H) represents the posterior probability of X conditioned on H. Using the previous example, this is the probability of customer X, given that his age and income are known, purchas-

ing a product.
P(H) stands for prior probability of H. For example, this value represents the likelihood of a customer, regardless of any personal information such as age or income, purchasing a product.
P(X) represents the prior probability of X. For example, this represents how likely it is that a person from the entire dataset belongs to certain age and income groups (Han and Kamber 2011, 350-51).

Naïve Bayesian classifiers are probabilistic by nature and they are based on the Bayes' theorem. These classifiers strongly, or naïvely, assume that an attribute value's effect on a given class is indepentent of the values of other attributes (Han and Kamber 2011, 385). According to Lantz (2013, 95), these assumptions typically do not work or are not true when applied to most of the real-world scenarios. However, the performance of Naïve Bayes is decent even when these assumptions are mistaken. There are other machine learning algorithms that use Bayesian methods, but naïve Bayes is the most common of them and in addition it is the standard method in text classification (Lantz 2013, 95). The Naïve Bayesian algorithm is simple, fast and effective, it can handle noisy and missing data well, the estimated probability for a prediction is easy to get, and the requirement for training examples is not very high, and it works well with high amount of training examples as well. The drawbacks are that The Naïve Bayesian algorithm is not optimal for datasets with large numbers of numeric features, it relies on an assumption that features are independent and equally important, and estimated probabilities are not as reliable as the predicted classes (Lantz 2013, 95).

4.6.5 *K*-nearest neighbor classifiers

The *k*-nearest neighbor algorithm, *k*-NN in short, is a simple but effective method that is used for regression and classification. (Hastie, Tibshirani, and Friedman 2009, 11-18). The input comprises the *k* closest training examples in both cases, and the usage determines the output. When this algorithm is used for classification, a class membership will be the the output. Classification is made by a majority vote of the object's neighbors, and the object is assigned to the most common class among its' *k*-nearest neighbor. Typically, *k* is a small positive integer and for example, if $k = 1$, the object will be assigned to the class of this single nearest

neighbor. When this algorithm is used for regression, be the property value for the object will be the output. This property value is the average of the values of its' k-nearest neighbor (Mohammed, Khan, and Bashier 2017, 83). The k-NN algorithm is simple and effective, it has a fast training phase, and it does not assume anything about the underlying distribution of data. On the downside, the k-NN algorithm's classification phase is slow, it has a high memory requirement, missing data and nominal features require additional processing. In addition, the k-NN enables the creation of such models that do not limit the ability to discover new insights in the features' relationships (Lantz 2013, 67).

Nearest neighbor methods are so-called lazy learning algorithms because they do not execute the processes of abstraction and generalization. Lazy learners do not actually learn anything, instead they only store the training data verbatim, and because of this the training phase is very fast. The potential drawback here is that the prediction-making process can be relatively slow (Lantz 2013, 74-75).

4.6.6 Neural networks

Artificial neural networks are models that draw their inspiration from the biological neural networks such as animal brains. The basis of these networks are simple forms of inputs and outputs. Brains contain neurons, and in biological terms these neutrons are cells that can transmit and process electrical or chemical signals. Neurons are connected and together they form a network that resembles the notion of graph theory with nodes and edges. Artificial neural networks are used in real-time or very near real-time scenarios because they are fast and can efficiently handle large volumes of data. Currently artificial neural networks are one of the leading computational intelligence tools, but their performance is still far from the human brain (Bell 2014, 91-92; Mohammed, Khan, and Bashier 2017, 89-30).

A perceptron is the foundation of a neural network. The perceptron is able to receive many inputs, but it generates a single output. This process can be broken down to few stages. First, an input signal is delivered to the perceptron. After the perceptron has received the input signal is received, it passes this input value through a function and outputs the result of this function (Bell 2014, 94). A perceptron is the simplest kind of artificial neural network, and

it contains a single neuron which can receive multiple inputs and produce a single output (Mohammed, Khan, and Bashier 2017, 91-92). A visual demonstration is offered in figure 4.

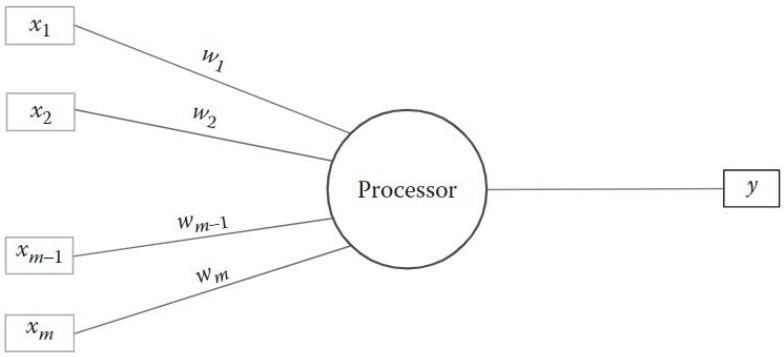

Figure 4. An example of a perceptron with *m* input features (Mohammed, Khan, and Bashier 2017, 91)

4.6.7 Linear discriminant analysis

Linear discriminant analysis is a generalization of Ronald Fisher's linear discriminant. Maximizing the class discrimination is the objective in linear discriminant analysis and the number of linear functions it produces is based on the classes. The highest value class for its' linear function will be the predicted class for an instance. For example, linear discriminant analysis can be used to predict what smartphone brand a customer belonging to certain age and income groups could be interested in (Mohammed, Khan, and Bashier 2017, 107-8).

4.6.8 Support vector machine

Support vector machine is a classifier that is based on a linear discriminant function. It is the most popular such classifier and it has a robust performance especially in binary classification. (Murty and Raghava 2016, 41). Support vector machines are supervised learning models with associated learning algorithms that are used for data analysis and pattern recognition. Support vector machines are flexible and computationally efficient. Thus, they are versatile and can be applied to various kinds of classification problems (Zolotukhin and Hämäläinen 2013, 212).

In the training phase a support vector machine takes a set of input points. Then, each input point will be assigned to one of two categories, and finally the support vector machine builds a model which represents the input points. In this representation a clear gap, that is as wide as possible, divides the input points of different categories. Afterwards new data points will be mapped into the same space and prediction takes place. A data point is predicted to belong to a category based on which side of the gap it fell. Support vector machines can perform both linear and non-linear classification efficiently (Zolotukhin and Hämäläinen 2013, 212).

4.6.9 *K*-means clustering

The *k*-means is possibly the most often used method for clustering. Although it is occasionally confused with the similarly named *k*-nearest neighbor, the *k*-means is a different algorithm, albeit it has some similarities with the *k*-NN method. The *k*-means algorithm assigns every n example into k clusters, where k is a predefined number. Maximizing the differences between clusters and minimizing the differences within each cluster are the objectives of *k*-means clustering (Lantz 2013, 271-72). For example, there could be 1000 objects and the objective could be to find 4 clusters. In this example, $n = 1000$ and $k = 4$. Every cluster has a point where the distance of the objects will be calculated. This point is called the centroid, which is also known as the mean, and this is where the name *k*-means originates (Bell 2014, 164).

The *k*-means algorithm has two phases. First, examples are assigned to an initial set of k clusters. Next, the assignments are updated by adjusting the boundaries of the cluster according to the examples that currently fall into it. This two-step process of assigning and updating happens multiple times, up to the point where making changes does not improve the cluster fit. At this stage, the clusters are finalized, and the process stops (Lantz 2013, 272). One thing to take into consideration is that each time the *k*-means algorithm is used, it gives different means and clusters due to the random selection of the initial *k*-means (Mohammed, Khan, and Bashier 2017, 133).

What is good about the *k*-means algorithm is that it uses simple principles for cluster identification, it is efficient and very flexible, and with simple adjustments the *k*-means can be ad-

justed to deal with almost every drawback it has. In addition, the *k*-means algorithm divides the data into useful cluster well. The drawbacks are that this algorithm is not as sophisticated as some of the more recent clustering algorithms. Also, as the *k*-means uses an element of random chance, it is not able to find the optimal set of clusters every time. Another drawback is that the number of clusters that exist in the data naturally must be guessed reasonably well when using the *k*-means algorithm (Lantz 2013, 271).

4.7 Machine learning based code analysis

According to other research, machine learning can be used to effectively analyze and evaluate code, shellcode included. Borders, Prakash, and Zielinski (2007) introduce Spector in their research paper. Spector automatically inspects shellcodes which an intrusion detection system has already deemed as malicious. In addition, Spector can at least deal with obfuscation and polymorphism that were current when the research paper was released. After the inspection Spector produces an output which tells the user what the shellcode does, and this means that the process of manual reverse engineering can be completely skipped. In this research, Spector was used to analyze approximately 23000 payloads, and it analyzed these payloads in about three hours. Spector found 11 different classes of shellcodes based on how they functioned. For example, these classes included code which connects back to the attacker or to a malicious web server (Borders, Prakash, and Zielinski 2007, 501-502, 513).

Clemens (2015) demonstrates that machine learning can be used to automatically and accurately analyze object code. This research proves that it is possible to discover the target architecture and endianness of object code as well as other important information by using machine learning techniques. This kind of automatic analysis allows analysts to entirely skip the process of identifying the code's endianness and target architecture. In this research, four features were derived and the purpose of these features was to assist in identifying the architecture and endianness of the target code. Clemens' theory was that these features are enough to classify this information. A dataset of over 16000 code samples from 20 different architectures was created in order to conduct experiments and test this theory. The results show that different machine learning algorithms can accurately identify the target architecture and endianness of object code even when only a small sample is available. Ten different

algorithms were tested and from those the support vector machine and nearest neighbor performed the best with approximately 90% accuracy when only 16 bytes of sample data was available. In addition, by 8 kilobytes of sample data almost every classifier reached 90% accuracy. (Clemens 2015, S156-S162).

Kairajärvi, Costin, and Hämäläinen (2020b) introduce ISAdetect, an automatic state-of-the-art tool for detecting CPU architecture and endianness from binary files and object code. This research notes that these state-of-the-art tools have potential, but they don't have the support of public datasets and toolsets. Therefore evaluating, comparing and improving any of these datasets, machine learning models and techniques is difficult. In this research the missing toolset and datasets are developed from the beginning. The dataset contains ISO files, DEB files, ELF files and ELF code sections and it covers multiple architectures. Several different classifiers were trained and tested, and then the performance of these classifiers was compared against other research such as (Clemens 2015). In addition, the effect of the sample size on the performance of the classification was studied. In this case, the classifiers were tested against a set of code sections of increasingly varying size. Finally, the performance of the classifiers was tested against complete binaries. The results show that when the classifiers are trained and tested using only sections of code from compiled binary files, they can reach over 98% accuracy. (Kairajärvi, Costin, and Hämäläinen 2020b).

5 Methodology and research data

Data in this thesis is acquired by observing, analyzing, and measuring the functionality of a machine learning based ISA detection application, and the collected data is the cornerstone of the research. Therefore, this thesis falls under the umbrella of empirical research (University of Jyväskylä 2010b). As this thesis will observe and measure the detection accuracy of the application, results will be presented in numeric variables, and final discussion and conclusion will be based on these variables, the research method should be quantitative (University of Jyväskylä 2010d). Of all the quantitative research methods (University of Jyväskylä 2010e), the best choice for this study seems to be experimental research. Experimental research allows the observation and analysis of the application in a controlled environment which will be created for this thesis. The results will be accurate, because experimental research enables controlled and systematic observation of the detection application (University of Jyväskylä 2010c).

Experimental research methods are known as hypothetico-deductive and quasi-experiment. Hypothetico-deductive research focuses on hypotheses or behavior predictions, and the majority of the work comprises collecting evidence which either supports the hypotheses or prove them wrong. It can be said that this kind of research is the traditional scientific method. In a quasi-experiment, the researcher has hypotheses, but cannot control some elements outside the research environment, and therefore is not capable of isolating, identifying, and controlling the variables. This means that conducting a legitimate hypothetico-deductive is not possible. Therefore, the researcher runs a quasi-experiment instead, and operates with the variables that can be controlled and acknowledges those that cannot be controlled. In addition, quasi-experiment can be a viable choice, if there is an incentive to conduct the research without hypotheses (Edgar and Manz 2017, 73-74). For this thesis, the hypothetico-deductive method is the most viable method. The hypotheses are:

- The detection accuracy is satisfactory.
- The detection accuracy is not satisfactory.

These hypotheses will drive the experimentation, and the satisfactory level of accuracy will

be derived from similar research. These hypotheses also follow the guidelines of a good hypothesis: they can be observed and tested, and they are clearly defined, single concept and predictive (Edgar and Manz 2017, 219).

Other research methods do not feel as viable as experimental research does. For example, case study is an observational method, but it is a method for studying events or situations that have already passed. Potentially case study could be a solid choice for this kind of research, but for this thesis there most likely would not be enough cases to study (Edgar and Manz 2017, 133-134).

5.1 Reliability and validity

When evaluating research, reliability and validity are probably the most essential factors to take into consideration. Reliability means how consistent the analysis is and how repeatable the measuring results are. For example, in a study like this the tests can be run multiple times to see whether or not the output is the same on each run. If the output is the same each time, reliability is high and if not, reliability is low. Validity means how accurately the intended factors are measured (University of Jyväskylä 2010a).

5.2 Research environment

The research was conducted in a virtual environment for several reasons, safety being the most important of them. Virtualization was implemented via Oracle VM VirtualBox (Oracle 2020) and operating system used in this virtual environment was Kali Linux. Kali Linux is an open source operating system that is funded and maintained by a company called Offensive Security. Its' main uses lie in the field of cyber security. Kali Linux includes approximately 600 tools which are able to perform various tasks related to information security, for example penetration testing, security research, digital forensics, security auditing and reverse engineering. (OffSec Services Limited 2020c)

5.3 MSFvenom bad character analysis

Different byte combinations can be given to MSFvenom as bad charecters using the -b parameter (OffSec Services Limited 2020a). Some common bad characters were described in section 2.1, but in this analysis every one byte combination is supplied to MSFvenom in order to see which combinations are accepted and which are rejected. In total there are 256 possible one byte combinations. Two Python scripts were created in order to accomplish this task, and they can be viewed in appendix A. The first script attempts to generate certain shellcodes with each different one byte combination as a bad character. Each successfully generated shellcode file is tagged with the used byte combination. Below is an example of a succesfully created file tagged with the used byte combination:

$$x0a\_linux\_mipsle\_reboot.c$$

The second script is used to check which byte combinations were accepted and which were rejected by comparing the contents of the table which contains each one byte combination to the byte tags of the successfully generated shellcodes.

The chosen architectures were x86, x64, MIPS, ARM, ARM 64, PowerPC, PowerPC 64 and SPARC. For each architecture, one payload was chosen from the MSFvenom selection, and the chosen codes were:

- x86 linux/x86/chmod
- x64 osx/x64/say
- MIPS linux/mipsle/reboot
- ARM osx/armle/vibrate
- ARM 64 linux/aarch64/shell_reverse_tcp
- PowerPC osx/ppc/shell/find_tag
- PowerPC 64 linux/ppc64/shell_find_port
- SPARC solaris/sparc/shell_find_port

After the initial bad character analysis, this thesis also examined whether or not changing the MSFvenom input parameters such as LHOST, LPORT and RHOST affect the program accepts or rejects one byte combinations as bad characters. LHOST stands for the listen address, LPORT stands for the listen port and RHOST stands for the target address (OffSec

Services Limited 2020a). Another set of Python scripts which are available in appendix A were written in order to accomplish this task.

The chosen architectures and payloads for LHOST tests were:

- x86 linux/x86/shell/reverse_tcp
- x64 linux/x64/shell/reverse_tcp
- MIPS linux/mipsbe/shell/reverse_tcp
- ARM osx/armle/shell_reverse_tcp
- ARM 64 linux/aarch64/shell_reverse_tcp
- PowerPC linux/ppc/shell_reverse_tcp
- PowerPC 64 linux/ppc64/shell_reverse_tcp
- SPARC bsd/sparc/shell_reverse_tcp

For LPORT and RHOST tests the ARM 64 architecture was discarded as no suitable payload could be found. The chosen architectures and payloads for LPORT and RHOST tests were:

- x86 bsd/x86/shell_bind_tcp
- x64 bsd/x64/shell_bind_tcp
- MIPS linux/mipsle/shell_bind_tcp
- ARM linux/armle/shell/bind_tcp
- PowerPC osx/ppc/shell_bind_tcp
- PowerPC 64 linux/ppc64/shell_bind_tcp
- SPARC bsd/sparc/shell_bind_tcp

5.4 Creating the shellcode database

This process was split in two parts. The first part was to collect shellcodes in large quantities from different sources and the second part was to add these shellcodes into a one database. The main sources for collecting shellcodes were MSFvenom, Exploit Database and Shell-Storm. MSFvenom, a part of the Metasploit framework, is a tool for generating payloads, shellcode included, and encoding them (OffSec Services Limited 2020a). Exploit Database is a collection or a repository of public exploits and corresponding vulnerable software. The

main purpose of Exploit Database is to help penetration testers and vulnerability researchers in their work. The exploits are collected from direct submissions, mailing lists and other public sources (OffSec Services Limited 2020b). Shell-Storm is a database dedicated for shellcodes, which have been gathered via contributions. However, the shellcodes available at the site are not typically used in real life situations, rather they are intended for study cases (”Shell-Storm Shellcodes Database” 2020).

Shell-Storm shellcodes were downloaded directly from the website using the wget tool and Exploit Database shellcodes were downloaded from the project's GitHub repository (OffSec Services Limited 2020d). For MSFvenom, a Python script, which can be viewed in appendix B, was made in order to automate the process of generating multiple shellcodes. The script reads the payload names from a text file which includes every Metasploit payload and attempts to generate shellcodes from them. The script also checks whether a file already exists or not, so it will waste time by overwriting previously generated shellcodes. It is a very crude script with room for improvement, and it does not successfully create every payload in the list, but it accomplishes what is needed for the purposes of this thesis. The problem with this script is that it generalizes MSFvenom's commands and parameters, and the same command may not work for every payload. The script allows creating shellcodes with or without the MSFvenom -b parameter. The script has some predefined bad characters, but it also lets the user enter custom bad characters. The script can edited for different shellcode formats by changing a value of one string in the code. This script tags the filename based on which bad character option was used. After running the script multiple times with different values and formats, duplicate files were identified and deleted with a tool called rdfind, and then additional examination was manually performed with the diff command. To best knowledge, each piece of code in this set which was generated with MSFvenom should be unique.

Even though Exploit Database and Shell-Storm are valid databases, as such they were not suitable for this thesis because they are biased towards certain, perhaps more popular architectures. This applies to MSFvenom as well because it does not support each architecture equally (OffSec Services Limited 2020a), meaning that there are more shellcodes available for some architectures and less for others. Using these resources to train machine learning systems without adjusting the distribution of architectures would most likely create a class

imbalance problem as discussed in section 4.1.

The primary resources for the database were Exploit Database and Shell-Storm, and only the shellcodes which clearly stated the target architecture were selected, as manually inspecting the ones that didn't was too large of a task for the scope of this thesis. The shellcodes from these two sources were combined into a one single database whose structure can be seen in table 1. From this table it can be seen that the selection of these two sources is very biased towards the x86 architecture.

Architecture	Quantity
x86	1014
x86-64	191
x64	7
ARM	83
PowerPC	40
MIPS	31
SPARC	26
SuperH	8
CRISv32	2
RISC-V 64	1

Table 1. Architectures and the number of shellcodes after adding them from Exploit Database and Shell-Storm

To combat this imbalance issue, the shellcodes generated with MSFvenom were added to this emerging thesis database. The structure of the final database can be seen in table 2. This final database is still biased towards x86 but not as heavily as previously. This database contains shellcodes in various formats such as C, ELF and raw bytes. Using this database as such to train machine learning systems would most likely cause the class imbalance problem described in section 4.1. However, this database can be used to craft balanced datasets which then can be used in training without having to worry about the class imbalance problem.

Architecture	Quantity
x86	4033
x64	2852
x86-64	191
ARM	1592
ARM 64	666
StrongARM	3
MIPS	1758
MIPS 64	3
PowerPC	1796
PowerPC 64	1303
PowerPC e500v2	3
SPARC	1787
SuperH	8
CRISv32	2
RISC-V 64	1

Table 2. Final shellcode database

5.5 Testing a machine learning based ISA detection system

The application under examination is called ISAdetect (Kairajärvi, Costin, and Hämäläinen 2020b) and it was downloaded and installed from the project's GitHub repository (Kairajärvi, Costin, and Hämäläinen 2020a). However, the actual testing was done with a live demo[1]. This was due to a scikit-learn version mismatch between the trained classifiers and the version that this tool uses. Probably because of this, the GitHub version of the application gave slightly different results than the live demo. In addition, this could have made a negative impact on the reliability and validity of the results. Hence, the assumption was made that the web-based live demo is more reliable and the testing was conducted with that version of the tool. According to Kairajärvi, Costin, and Hämäläinen (2020b), currently ISAdetect

1. https://isadetect.com/

supports the following architectures: alpha, amd64, arm64, armel, armhf, hppa, i386, ia64, m68k, mips, mips64el, mipsel, powerpc, powerpcspe, powerpc64, powerpc64el, riscv, s390, s390x, sh4, sparc, sparc64 and x32. In addition, currently ISAdetect accepts shellcodes in raw byte and ELF formats (Kairajärvi, Costin, and Hämäläinen 2020b). The shellcodes for the testing were selected from the database with these architectures and requirements in mind as there was no point in testing unsupported architectures and formats. Each piece of shellcode was tested multiple times in order to see if the application would give the same result on each run. ISAdetect can be set to analyze code-only sections, full binaries with code and data sections and small fragments of less than 2000 bytes (Kairajärvi, Costin, and Hämäläinen 2020b). From these options the code-only and fragment options were tested. Code-only is the default option and fragment was included because some of the test files are so small. In both tests the used classifier was random forest as it is the default option and also the best performing classifier (Kairajärvi, Costin, and Hämäläinen 2020b). After these tests the performance of each classifier was tested with a smaller set of shellcodes. In addition to random forest, these classifiers are 1 nearest neighbor, 3 nearest neighbor, decision tree, naïve Bayes, neural net and SVM/SMO (Kairajärvi, Costin, and Hämäläinen 2020b). Based on the papers by Clemens (2015) and Kairajärvi, Costin, and Hämäläinen (2020b), it was decided that 90% would be an acceptable accuracy for the detection.

The size of the shellcode files vary from 16 bytes to 1,5 megabytes. However, the file size does not necessarily reflect the size of the payload. In some cases it might be smaller than the actual file size. Also, the goal was to mostly use unique files in the testing, so the shellcodes created in section 5.3 were not used. Table 3 shows the structure of the test set. These shellcodes are either full ELF files or raw binary executable files. The file extension reveals the format.

Architecture	Quantity
ARM	17
ARM 64	7
MIPS	46
MIPS 64	3
PowerPC	30
PowerPC 64	13
SPARC	23

Table 3. Shellcode collection for running tests

6 Results

In this chapter, the results of the detection and the MSFvenom bad character analysis are presented. For the sake of clarity, the presentation of the results is divided into several sections. The results of the code-only tests can be viewed in section 6.2 and the results of the fragment option tests can be viewed in section 6.3. The MSFvenom bad character analysis results can be found in section 6.6.

6.1 Detection results

The detection results are presented in tables and there is one table for each architecture. In each table, the first column lists the shellcode that is being tested, the second column shows the shellcode file size in bytes, the third and fourth columns show the original and detected architectures, the fifth and sixth colums show the original and detected word sizes, the seventh and the eighth columns show the original and detected endiannesses, the ninth column show the prediction probability and finally, the tenth column show whether or not the instruction set architecture was detected correctly.

6.2 Results from testing shellcodes with the code-only option

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_linux_armbe_meterpreter_reverse_http.elf	1022588	arm	arm	32	32	big	little	0.55	no
no_bc_linux_armbe_meterpreter_reverse_https.elf	1022588	arm	arm	32	32	big	little	0.55	no
no_bc_linux_armbe_meterpreter_reverse_tcp.elf	1022588	arm	arm	32	32	big	little	0.55	no
no_bc_linux_armbe_shell_bind_tcp	118	arm	m68k	32	32	big	big	0.16	no
no_bc_linux_armle_adduser	119	arm	mips	32	64	little	little	0.16	no
no_bc_linux_armle_adduser.elf	203	arm	ia64	32	64	little	little	0.15	no
no_bc_linux_armle_meterpreter_bind_tcp	232	arm	arm	32	32	little	little	0.38	yes
no_bc_linux_armle_meterpreter_bind_tcp.elf	316	arm	arm	32	32	little	little	0.31	yes
no_bc_linux_armle_meterpreter_reverse_http.elf	1022588	arm	arm	32	32	little	little	0.89	yes
no_bc_linux_armle_meterpreter_reverse_https.elf	1022588	arm	arm	32	32	little	little	0.89	yes
no_bc_linux_armle_meterpreter_reverse_tcp	260	arm	arm	32	32	little	little	0.35	yes
no_bc_linux_armle_meterpreter_reverse_tcp.elf	344	arm	arm	32	32	little	little	0.28	yes
no_bc_osx_armle_execute_bind_tcp	248	arm	arm	32	32	little	little	0.19	yes
no_bc_osx_armle_execute_reverse_tcp	184	arm	arm	32	32	little	little	0.18	yes
no_bc_osx_armle_vibrate	16	arm	arm	32	32	little	little	0.24	yes
xff_linux_armle_shell_reverse_tcp	172	arm	arm	32	32	little	little	0.18	yes
xff_linux_armle_shell_reverse_tcp.elf	256	arm	arm	32	32	little	little	0.24	yes
no_bc_apple_ios_aarch64_shell_reverse_tcp	152	arm64	arm64	64	64	little	little	0.24	yes
no_bc_linux_aarch64_meterpreter_reverse_http.elf	1092000	arm64	arm64	64	64	little	little	0.82	yes
no_bc_linux_aarch64_meterpreter_reverse_https.elf	1092000	arm64	arm64	64	64	little	little	0.82	yes
no_bc_linux_aarch64_meterpreter_reverse_tcp	212	arm64	arm64	64	64	little	little	0.27	yes
no_bc_linux_aarch64_meterpreter_reverse_tcp.elf	332	arm64	arm64	64	64	little	little	0.25	yes
x0dxff_linux_aarch64_shell_reverse_tcp	152	arm64	arm64	64	64	little	little	0.23	yes
x0dxff_linux_aarch64_shell_reverse_tcp.elf	272	arm64	ia64	64	64	little	little	0.2	no

Table 4. Detection results for ARM and ARM 64 using the code-only option

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_linux_mipsbe_meterpreter_reverse_http.elf	1460700	mips	mips	32	32	big	big	0.81	yes
no_bc_linux_mipsbe_meterpreter_reverse_https.elf	1460700	mips	mips	32	32	big	big	0.81	yes
no_bc_linux_mipsbe_meterpreter_reverse_tcp	272	mips	ia64	32	64	big	little	0.11	no
no_bc_linux_mipsbe_meterpreter_reverse_tcp.elf	356	mips	sparc	32	32	big	big	0.13	no
no_bc_linux_mipsbe_reboot	32	mips	arm64	32	64	big	little	0.1	no
no_bc_linux_mipsbe_reboot.elf	116	mips	ia64	32	64	big	little	0.13	no
no_bc_linux_mipsbe_shell_bind_tcp	232	mips	mips	32	32	big	big	0.33	yes
no_bc_linux_mipsbe_shell_bind_tcp.elf	316	mips	mips	32	32	big	big	0.37	yes
no_bc_linux_mipsle_meterpreter_reverse_http.elf	1463172	mips	mips	32	32	little	little	0.84	yes
no_bc_linux_mipsle_meterpreter_reverse_https.elf	1463172	mips	mips	32	32	little	little	0.84	yes
no_bc_linux_mipsle_meterpreter_reverse_tcp	272	mips	ia64	32	64	little	little	0.11	no
no_bc_linux_mipsle_meterpreter_reverse_tcp.elf	356	mips	sparc	32	32	little	big	0.13	no
no_bc_linux_mipsle_reboot	32	mips	arm64	32	64	little	little	0.1	no
no_bc_linux_mipsle_reboot.elf	116	mips	amd64	32	64	little	little	0.132	no
no_bc_linux_mipsle_shell_bind_tcp	232	mips	mips	32	32	little	little	0.23	yes
no_bc_linux_mipsle_shell_bind_tcp.elf	316	mips	mips	32	32	little	little	0.25	yes
x00_linux_mipsbe_meterpreter_reverse_tcp	376	mips	m68k	32	32	big	big	0.17	no
x00_linux_mipsbe_meterpreter_reverse_tcp.elf	460	mips	m68k	32	32	big	big	0.22	no
x00_linux_mipsbe_shell_reverse_tcp	376	mips	m68k	32	32	big	big	0.18	no
x00_linux_mipsbe_shell_reverse_tcp.elf	460	mips	sparc	32	32	big	big	0.1	no
x00_linux_mipsle_meterpreter_reverse_tcp	376	mips	m68k	32	32	little	big	0.18	no
x00_linux_mipsle_meterpreter_reverse_tcp.elf	460	mips	m68k	32	32	little	big	0.13	no
x00_linux_mipsle_shell_reverse_tcp	376	mips	m68k	32	32	little	big	0.13	no
x00_linux_mipsle_shell_reverse_tcp.elf	460	mips	riscv	32	64	little	little	0.16	no
x00x0d_linux_mipsbe_meterpreter_reverse_tcp	380	mips	m68k	32	32	big	big	0.18	no
x00x0d_linux_mipsbe_meterpreter_reverse_tcp.elf	464	mips	m68k	32	32	big	big	0.21	no
x00x0d_linux_mipsbe_shell_bind_tcp	340	mips	armhf	32	32	big	little	0.17	no
x00x0d_linux_mipsbe_shell_bind_tcp.elf	424	mips	sparc	32	64	big	big	0.15	no
x00x0d_linux_mipsbe_shell_reverse_tcp	380	mips	sparc	32	64	big	big	0.13	no
x00x0d_linux_mipsbe_shell_reverse_tcp.elf	464	mips	m68k	32	32	big	big	0.12	no
x00x0d_linux_mipsle_meterpreter_reverse_tcp	380	mips	m68k	32	32	little	big	0.11	no
x00x0d_linux_mipsle_meterpreter_reverse_tcp.elf	464	mips	sparc	32	32	little	big	0.19	no
x00x0d_linux_mipsle_shell_bind_tcp	340	mips	sparc	32	32	little	big	0.12	no
x00x0d_linux_mipsle_shell_bind_tcp.elf	424	mips	alpha	32	64	little	little	0.16	no
x00x0d_linux_mipsle_shell_reverse_tcp	380	mips	riscv	32	64	little	little	0.12	no
x00x0d_linux_mipsle_shell_reverse_tcp.elf	464	mips	m68k	32	32	little	big	0.14	no
x0d_linux_mipsbe_shell_bind_tcp	340	mips	m68k	32	32	big	big	0.11	no
x0d_linux_mipsbe_shell_bind_tcp.elf	424	mips	i386	32	32	big	little	0.12	no
x0d_linux_mipsle_shell_bind_tcp	340	mips	m68k	32	32	little	big	0.13	no
x0d_linux_mipsle_shell_bind_tcp.elf	424	mips	m68k	32	32	little	big	0.11	no
x5c_linux_mipsbe_meterpreter_reverse_tcp	376	mips	m68k	32	32	big	big	0.15	no
x5c_linux_mipsbe_shell_bind_tcp	336	mips	i386	32	32	big	little	0.13	no
x5c_linux_mipsbe_shell_reverse_tcp	376	mips	riscv	32	64	big	little	0.13	no
x5c_linux_mipsle_meterpreter_reverse_tcp	376	mips	arm64	32	64	little	little	0.15	no
x5c_linux_mipsle_shell_bind_tcp	336	mips	m68k	32	32	little	big	0.12	no
x5c_linux_mipsle_shell_reverse_tcp	376	mips	ia64	32	64	little	little	0.14	no
no_bc_linux_mips64_meterpreter_reverse_http.elf	1568856	mips64	mips64	64	64	big	little	0.49	no
no_bc_linux_mips64_meterpreter_reverse_https.elf	1568856	mips64	mips64	64	64	big	little	0.49	no
no_bc_linux_mips64_meterpreter_reverse_tcp.elf	1568856	mips64	mips64	64	64	big	little	0.49	no

Table 5. Detection results for MIPS and MIPS 64 using the code-only option

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_aix_ppc_shell_interact	56	ppc	sparc	32	32	big	big	0.15	no
no_bc_aix_ppc_shell_reverse_tcp	204	ppc	m68k	32	32	big	big	0.16	no
no_bc_linux_ppc_meterpreter_reverse_http.elf	1210840	ppc	ppc	32	32	big	big	0.47	yes
no_bc_linux_ppc_meterpreter_reverse_https.elf	1210840	ppc	ppc	32	32	big	big	0.47	yes
no_bc_linux_ppc_meterpreter_reverse_tcp.elf	1210840	ppc	ppc	32	32	big	big	0.47	yes
no_bc_linux_ppc_shell_find_port	171	ppc	sparc	32	64	big	big	0.13	no
no_bc_linux_ppc_shell_reverse_tcp	183	ppc	m68k	32	32	big	big	0.09	no
no_bc_osx_ppc_shell_reverse_tcp	100	ppc	ppc	32	64	big	big	0.2	no
x00_linux_ppc_shell_find_port	171	ppc	m68k	32	32	big	big	0.12	no
x00_osx_ppc_shell_bind_tcp	228	ppc	i386	32	32	big	little	0.11	no
x00_osx_ppc_shell_reverse_tcp	176	ppc	m68k	32	32	big	big	0.19	no
x00x0a_aix_ppc_shell_find_port	220	ppc	sparc	32	32	big	big	0.13	no
x00x0a_linux_ppc_shell_reverse_tcp	260	ppc	mips64	32	64	big	little	0.16	no
x00x0a_osx_ppc_shell_find_tag	76	ppc	m68k	32	32	big	big	0.11	no
x00x0d_osx_ppc_shell_bind_tcp	228	ppc	i386	32	32	big	little	0.15	no
x00x0d_osx_ppc_shell_reverse_tcp	176	ppc	m68k	32	32	big	big	0.13	no
x0a_aix_ppc_shell_find_port	220	ppc	sparc	32	32	big	big	0.13	no
x0a_aix_ppc_shell_reverse_tcp	280	ppc	armhf	32	32	big	little	0.1	no
x0a_linux_ppc_shell_reverse_tcp	260	ppc	arm64	32	64	big	little	0.11	no
x0a_osx_ppc_shell_find_tag	76	ppc	m68k	32	32	big	big	0.12	no
x0ax0d_aix_ppc_shell_find_port	220	ppc	sparc	32	32	big	big	0.15	no
x0ax0d_aix_ppc_shell_reverse_tcp	280	ppc	i386	32	32	big	little	0.19	no
x0ax0d_linux_ppc_shell_find_port	171	ppc	m68k	32	32	big	big	0.13	no
x0ax0d_osx_ppc_shell_bind_tcp	228	ppc	i386	32	32	big	little	0.15	no
x0ax0d_osx_ppc_shell_reverse_tcp	176	ppc	i386	32	32	big	little	0.13	no
x0d_osx_ppc_shell_find_tag	76	ppc	m68k	32	32	big	big	0.12	no
x5c_aix_ppc_shell_reverse_tcp	280	ppc	m68k	32	32	big	big	0.13	no
x5c_linux_ppc_shell_bind_tcp	300	ppc	armhf	32	32	big	little	0.14	no
x5c_linux_ppc_shell_find_port	171	ppc	m68k	32	32	big	big	0.12	no
x5c_linux_ppc_shell_reverse_tcp	260	ppc	arm64	32	64	big	little	0.12	no
no_bc_linux_ppc64le_meterpreter_reverse_http.elf	1169208	ppc64	ppc64	64	64	little	little	0.93	yes
no_bc_linux_ppc64le_meterpreter_reverse_https.elf	1169208	ppc64	ppc64	64	64	little	little	0.93	yes
no_bc_linux_ppc64le_meterpreter_reverse_tcp.elf	1169208	ppc64	ppc64	64	64	little	little	0.93	yes
no_bc_linux_ppc64_shell_bind_tcp	223	ppc64	arm64	64	64	little	little	0.13	no
no_bc_linux_ppc64_shell_find_port	171	ppc64	arm64	64	64	little	little	0.13	no
no_bc_linux_ppc64_shell_reverse_tcp	183	ppc64	sparc	64	64	little	big	0.11	no
x00_linux_ppc64_shell_find_port	171	ppc64	arm64	64	64	little	little	0.14	no
x00x0a_linux_ppc64_shell_find_port	171	ppc64	arm64	64	64	little	little	0.13	no
x00x0d_linux_ppc64_shell_find_port	171	ppc64	arm64	64	64	little	little	0.11	no
x0a_linux_ppc64_shell_find_port	171	ppc64	arm64	64	64	little	little	0.13	no
x0ax0d_linux_ppc64_shell_find_port	171	ppc64	arm64	64	64	little	little	0.13	no
x0d_linux_ppc64_shell_find_port	171	ppc64	arm64	64	64	little	little	0.12	no
x5c_linux_ppc64_shell_find_port	171	ppc64	arm64	64	64	little	little	0.13	no

Table 6. Detection results for PowerPC and PowerPC 64 using the code-only option

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_bsd_sparc_shell_bind_tcp	164	sparc	sparc	32	32	big	big	0.34	yes
no_bc_bsd_sparc_shell_reverse_tcp	128	sparc	sparc	32	32	big	big	0.32	yes
no_bc_solaris_sparc_shell_bind_tcp	180	sparc	sparc	32	32	big	big	0.34	yes
no_bc_solaris_sparc_shell_find_port	136	sparc	sparc	32	32	big	big	0.45	yes
no_bc_solaris_sparc_shell_reverse_tcp	144	sparc	sparc	32	32	big	big	0.35	yes
x00_bsd_sparc_shell_reverse_tcp	180	sparc	m68k	32	32	big	big	0.17	no
x00_solaris_sparc_shell_bind_tcp	232	sparc	m68k	32	32	big	big	0.15	no
x00_solaris_sparc_shell_find_port	188	sparc	sparc	32	32	big	big	0.14	yes
x00_solaris_sparc_shell_reverse_tcp	196	sparc	sh4	32	32	big	little	0.13	no
x00x0a_bsd_sparc_shell_reverse_tcp	180	sparc	armhf	32	32	big	little	0.18	no
x00x0a_solaris_sparc_shell_bind_tcp	232	sparc	armhf	32	32	big	little	0.17	no
x00x0a_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	0.15	no
x00x0a_solaris_sparc_shell_reverse_tcp	196	sparc	sparc	32	32	big	big	0.15	yes
x00x0d_bsd_sparc_shell_reverse_tcp	180	sparc	riscv	32	64	big	little	0.13	no
x00x0d_solaris_sparc_shell_bind_tcp	232	sparc	riscv	32	64	big	little	0.18	no
x00x0d_solaris_sparc_shell_find_port	188	sparc	armhf	32	32	big	little	0.13	no
x00x0d_solaris_sparc_shell_reverse_tcp	196	sparc	m68k	32	32	big	big	0.14	no
x0a_solaris_sparc_shell_find_port	188	sparc	m68k	32	32	big	big	0.15	no
x0ax0d_solaris_sparc_shell_find_port	188	sparc	arm64	32	64	big	little	0.15	no
x0d_solaris_sparc_shell_find_port	188	sparc	sparc	32	32	big	big	0.14	yes
x5c_solaris_sparc_shell_bind_tcp	232	sparc	sparc	32	32	big	big	0.17	yes
x5c_solaris_sparc_shell_find_port	136	sparc	sparc	32	32	big	big	0.44	yes
x5c_solaris_sparc_shell_reverse_tcp	196	sparc	sparc	32	32	big	big	0.13	yes

Table 7. Detection results for SPARC using the code-only option

6.3 Results from testing shellcodes with the fragment option

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_linux_armbe_meterpreter_reverse_http.elf	1022588	arm	arm	32	32	big	little	0.726	no
no_bc_linux_armbe_meterpreter_reverse_https.elf	1022588	arm	arm	32	32	big	little	0.726	no
no_bc_linux_armbe_meterpreter_reverse_tcp.elf	1022588	arm	arm	32	32	big	little	0.726	no
no_bc_linux_armbe_shell_bind_tcp	118	arm	sh4	32	32	big	little	0.723	no
no_bc_linux_armle_adduser	119	arm	sh4	32	32	little	little	0.935	no
no_bc_linux_armle_adduser.elf	203	arm	ia64	32	64	little	little	0.582	no
no_bc_linux_armle_meterpreter_bind_tcp	232	arm	arm	32	32	little	little	0.991	yes
no_bc_linux_armle_meterpreter_bind_tcp.elf	316	arm	ia64	32	64	little	little	0.902	no
no_bc_linux_armle_meterpreter_reverse_http.elf	1022588	arm	arm	32	32	little	little	0.739	yes
no_bc_linux_armle_meterpreter_reverse_https.elf	1022588	arm	arm	32	32	little	little	0.739	yes
no_bc_linux_armle_meterpreter_reverse_tcp	260	arm	arm	32	32	little	little	0.917	yes
no_bc_linux_armle_meterpreter_reverse_tcp.elf	344	arm	ia64	32	64	little	little	0.949	no
no_bc_osx_armle_execute_bind_tcp	248	arm	arm	32	32	little	little	0.989	yes
no_bc_osx_armle_execute_reverse_tcp	184	arm	arm	32	32	little	little	0.983	yes
no_bc_osx_armle_vibrate	16	arm	armhf	32	32	little	little	0.978	no
xff_linux_armle_shell_reverse_tcp	172	arm	arm	32	32	little	little	0.587	yes
xff_linux_armle_shell_reverse_tcp.elf	256	arm	ia64	32	64	little	little	0.979	no
no_bc_apple_ios_aarch64_shell_reverse_tcp	152	arm64	arm64	64	64	little	little	0.538	yes
no_bc_linux_aarch64_meterpreter_reverse_http.elf	1092000	arm64	arm64	64	64	little	little	0.797	yes
no_bc_linux_aarch64_meterpreter_reverse_https.elf	1092000	arm64	arm64	64	64	little	little	0.797	yes
no_bc_linux_aarch64_meterpreter_reverse_tcp	212	arm64	arm64	64	64	little	little	0.996	yes
no_bc_linux_aarch64_meterpreter_reverse_tcp.elf	332	arm64	ia64	64	64	little	little	0.934	no
x0dxff_linux_aarch64_shell_reverse_tcp	152	arm64	arm64	64	64	little	little	0.679	yes
x0dxff_linux_aarch64_shell_reverse_tcp.elf	272	arm64	ia64	64	64	little	little	0.999	no

Table 8. Detection results for ARM and ARM 64 using the fragment option

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_linux_mipsbe_meterpreter_reverse_http.elf	1460700	mips	mips	32	32	big	big	0.876	yes
no_bc_linux_mipsbe_meterpreter_reverse_https.elf	1460700	mips	mips	32	32	big	big	0.876	yes
no_bc_linux_mipsbe_meterpreter_reverse_tcp	272	mips	mips	32	32	big	big	0.331	yes
no_bc_linux_mipsbe_meterpreter_reverse_tcp.elf	356	mips	mips	32	32	big	big	0.308	yes
no_bc_linux_mipsbe_reboot	32	mips	sh4	32	32	big	little	0.695	no
no_bc_linux_mipsbe_reboot.elf	116	mips	ia64	32	64	big	little	0.985	no
no_bc_linux_mipsbe_shell_bind_tcp	232	mips	m68k	32	32	big	big	0.638	no
no_bc_linux_mipsbe_shell_bind_tcp.elf	316	mips	mips	32	32	big	big	0.2	yes
no_bc_linux_mipsle_meterpreter_reverse_http.elf	1463172	mips	mips	32	32	little	little	0.880	yes
no_bc_linux_mipsle_meterpreter_reverse_https.elf	1463172	mips	mips	32	32	little	little	0.880	yes
no_bc_linux_mipsle_meterpreter_reverse_tcp	272	mips	mips	32	32	little	little	0.332	yes
no_bc_linux_mipsle_meterpreter_reverse_tcp.elf	356	mips	mips	32	32	little	little	0.357	yes
no_bc_linux_mipsle_reboot	32	mips	sh4	32	32	little	little	0.695	no
no_bc_linux_mipsle_reboot.elf	116	mips	ia64	32	64	little	little	0.985	no
no_bc_linux_mipsle_shell_bind_tcp	232	mips	m68k	32	32	little	big	0.705	no
no_bc_linux_mipsle_shell_bind_tcp.elf	316	mips	mips64	32	64	little	little	0.208	no
x00_linux_mipsbe_meterpreter_reverse_tcp	376	mips	sh4	32	32	big	little	0.976	no
x00_linux_mipsbe_meterpreter_reverse_tcp.elf	460	mips	sh4	32	32	big	little	0.578	no
x00_linux_mipsbe_shell_reverse_tcp	376	mips	hppa	32	32	big	big	0.266	no
x00_linux_mipsbe_shell_reverse_tcp.elf	460	mips	armhf	32	32	big	little	0.504	no
x00_linux_mipsle_meterpreter_reverse_tcp	376	mips	sh4	32	32	little	little	0.89	no
x00_linux_mipsle_meterpreter_reverse_tcp.elf	460	mips	sh4	32	32	little	little	0.459	no
x00_linux_mipsle_shell_reverse_tcp	376	mips	riscv	32	64	little	little	0.857	no
x00_linux_mipsle_shell_reverse_tcp.elf	460	mips	riscv	32	64	little	little	0.394	no
x00x0d_linux_mipsbe_meterpreter_reverse_tcp	380	mips	sh4	32	32	big	little	0.94	no
x00x0d_linux_mipsbe_meterpreter_reverse_tcp.elf	464	mips	m68k	32	32	big	big	0.339	no
x00x0d_linux_mipsbe_shell_bind_tcp	340	mips	sh4	32	32	big	little	0.921	no
x00x0d_linux_mipsbe_shell_bind_tcp.elf	424	mips	arm64	32	64	big	little	0.123	no
x00x0d_linux_mipsbe_shell_reverse_tcp	380	mips	sh4	32	32	big	little	0.719	no
x00x0d_linux_mipsbe_shell_reverse_tcp.elf	464	mips	sh4	32	32	big	little	0.283	no
x00x0d_linux_mipsle_meterpreter_reverse_tcp	380	mips	sh4	32	32	little	little	0.847	no
x00x0d_linux_mipsle_meterpreter_reverse_tcp.elf	464	mips	arm64	32	64	little	little	0.155	no
x00x0d_linux_mipsle_shell_bind_tcp	340	mips	sh4	32	32	little	little	0.896	no
x00x0d_linux_mipsle_shell_bind_tcp.elf	424	mips	arm64	32	64	little	little	0.231	no
x00x0d_linux_mipsle_shell_reverse_tcp	380	mips	sh4	32	32	little	little	0.855	no
x00x0d_linux_mipsle_shell_reverse_tcp.elf	464	mips	ppc	32	32	little	big	0.124	no
x0d_linux_mipsbe_shell_bind_tcp	340	mips	sh4	32	32	big	little	0.630	no
x0d_linux_mipsbe_shell_bind_tcp.elf	424	mips	sh4	32	32	big	little	0.253	no
x0d_linux_mipsle_shell_bind_tcp	340	mips	sh4	32	32	little	little	0.950	no
x0d_linux_mipsle_shell_bind_tcp.elf	424	mips	alpha	32	64	little	little	0.151	no
x5c_linux_mipsbe_meterpreter_reverse_tcp	376	mips	sh4	32	32	big	little	0.998	no
x5c_linux_mipsbe_shell_bind_tcp	336	mips	riscv	32	64	big	little	0.601	no
x5c_linux_mipsbe_shell_reverse_tcp	376	mips	ppcspe	32	32	big	big	0.805	no
x5c_linux_mipsle_meterpreter_reverse_tcp	376	mips	sh4	32	32	little	little	0.581	no
x5c_linux_mipsle_shell_bind_tcp	336	mips	arm64	32	64	little	little	0.590	no
x5c_linux_mipsle_shell_reverse_tcp	376	mips	armhf	32	32	little	little	0.954	no
no_bc_linux_mips64_meterpreter_reverse_http.elf	1568856	mips64	ia64	64	64	big	little	0.485	no
no_bc_linux_mips64_meterpreter_reverse_https.elf	1568856	mips64	ia64	64	64	big	little	0.485	no
no_bc_linux_mips64_meterpreter_reverse_tcp.elf	1568856	mips64	ia64	64	64	big	little	0.485	no

Table 9. Detection results for MIPS and MIPS 64 using the fragment option

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_aix_ppc_shell_interact	56	ppc	m68k	32	32	big	big	0.288	no
no_bc_aix_ppc_shell_reverse_tcp	204	ppc	ppcspe	32	32	big	big	0.981	no
no_bc_linux_ppc_meterpreter_reverse_http.elf	1210840	ppc	ppc	32	32	big	big	0.874	yes
no_bc_linux_ppc_meterpreter_reverse_https.elf	1210840	ppc	ppc	32	32	big	big	0.874	yes
no_bc_linux_ppc_meterpreter_reverse_tcp.elf	1210840	ppc	ppc	32	32	big	big	0.874	yes
no_bc_linux_ppc_shell_find_port	171	ppc	ppc64	32	64	big	little	0.855	no
no_bc_linux_ppc_shell_reverse_tcp	183	ppc	ppc64	32	64	big	little	0.783	no
no_bc_osx_ppc_shell_reverse_tcp	100	ppc	ppc	32	32	big	big	0.396	yes
x00_linux_ppc_shell_find_port	171	ppc	ppc64	32	64	big	little	0.848	no
x00_osx_ppc_shell_bind_tcp	228	ppc	sh4	32	32	big	little	0.902	no
x00_osx_ppc_shell_reverse_tcp	176	ppc	sh4	32	32	big	little	0.618	no
x00x0a_aix_ppc_shell_find_port	220	ppc	ppcspe	32	32	big	big	0.871	no
x00x0a_linux_ppc_shell_reverse_tcp	260	ppc	sh4	32	32	big	little	0.835	no
x00x0a_osx_ppc_shell_find_tag	76	ppc	ppcspe	32	32	big	big	0.84	no
x00x0d_osx_ppc_shell_bind_tcp	228	ppc	armhf	32	32	big	big	0.455	no
x00x0d_osx_ppc_shell_reverse_tcp	176	ppc	sh4	32	32	big	little	0.694	no
x0a_aix_ppc_shell_find_port	220	ppc	ppcspe	32	32	big	big	0.816	no
x0a_aix_ppc_shell_reverse_tcp	280	ppc	armhf	32	32	big	little	0.291	no
x0a_linux_ppc_shell_reverse_tcp	260	ppc	sh4	32	32	big	little	0.762	no
x0a_osx_ppc_shell_find_tag	76	ppc	ppcspe	32	32	big	big	0.849	no
x0ax0d_aix_ppc_shell_find_port	220	ppc	ppcspe	32	32	big	big	0.814	no
x0ax0d_aix_ppc_shell_reverse_tcp	280	ppc	sh4	32	32	big	little	0.9	no
x0ax0d_linux_ppc_shell_find_port	171	ppc	ppc64	32	64	big	little	0.846	no
x0ax0d_osx_ppc_shell_bind_tcp	228	ppc	armhf	32	32	big	little	0.308	no
x0ax0d_osx_ppc_shell_reverse_tcp	176	ppc	risv	32	64	big	little	0.917	no
x0d_osx_ppc_shell_find_tag	76	ppc	ppcspe	32	32	big	big	0.88	no
x5c_aix_ppc_shell_reverse_tcp	280	ppc	sh4	32	32	big	little	0.596	no
x5c_linux_ppc_shell_bind_tcp	300	ppc	sh4	32	32	big	little	0.721	no
x5c_linux_ppc_shell_find_port	171	ppc	ppc64	32	64	big	little	0.87	no
x5c_linux_ppc_shell_reverse_tcp	260	ppc	riscv	32	64	big	little	0.754	no
no_bc_linux_ppc64le_meterpreter_reverse_http.elf	1169208	ppc64	ppc64	64	64	little	big	0.907	no
no_bc_linux_ppc64le_meterpreter_reverse_https.elf	1169208	ppc64	ppc64	64	64	little	big	0.907	no
no_bc_linux_ppc64le_meterpreter_reverse_tcp.elf	1169208	ppc64	ppc64	64	64	little	big	0.907	no
no_bc_linux_ppc64_shell_bind_tcp	223	ppc64	ppc64	64	64	little	little	0.985	yes
no_bc_linux_ppc64_shell_find_port	171	ppc64	ppc64	64	64	little	little	0.954	yes
no_bc_linux_ppc64_shell_reverse_tcp	183	ppc64	ppc64	64	64	little	little	0.957	yes
x00_linux_ppc64_shell_find_port	171	ppc64	ppc64	64	64	little	little	0.950	yes
x00x0a_linux_ppc64_shell_find_port	171	ppc64	ppc64	64	64	little	little	0.945	yes
x00x0d_linux_ppc64_shell_find_port	171	ppc64	ppc64	64	64	little	little	0.947	yes
x0a_linux_ppc64_shell_find_port	171	ppc64	ppc64	64	64	little	little	0.950	yes
x0ax0d_linux_ppc64_shell_find_port	171	ppc64	ppc64	64	64	little	little	0.936	yes
x0d_linux_ppc64_shell_find_port	171	ppc64	ppc64	64	64	little	little	0.947	yes
x5c_linux_ppc64_shell_find_port	171	ppc64	ppc64	64	64	little	little	0.951	yes

Table 10. Detection results for PowerPC and PowerPC 64 using the fragment option

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_bsd_sparc_shell_bind_tcp	164	sparc	sparc	32	32	big	big	0.998	yes
no_bc_bsd_sparc_shell_reverse_tcp	128	sparc	sparc	32	32	big	big	0.999	yes
no_bc_solaris_sparc_shell_bind_tcp	180	sparc	sparc	32	32	big	big	0.998	yes
no_bc_solaris_sparc_shell_find_port	136	sparc	sparc	32	32	big	big	0.986	yes
no_bc_solaris_sparc_shell_reverse_tcp	144	sparc	sparc	32	32	big	big	0.998	yes
x00_bsd_sparc_shell_reverse_tcp	180	sparc	sh4	32	32	big	little	0.922	no
x00_solaris_sparc_shell_bind_tcp	232	sparc	sh4	32	32	big	little	0.814	no
x00_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	0.86	no
x00_solaris_sparc_shell_reverse_tcp	196	sparc	sh4	32	32	big	little	0.94	no
x00x0a_bsd_sparc_shell_reverse_tcp	180	sparc	sh4	32	32	big	little	0.609	no
x00x0a_solaris_sparc_shell_bind_tcp	232	sparc	sh4	32	32	big	little	0.794	no
x00x0a_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	0.810	no
x00x0a_solaris_sparc_shell_reverse_tcp	196	sparc	sh4	32	32	big	little	0.920	no
x00x0d_bsd_sparc_shell_reverse_tcp	180	sparc	sh4	32	32	big	little	0.875	no
x00x0d_solaris_sparc_shell_bind_tcp	232	sparc	sh4	32	32	big	little	0.728	no
x00x0d_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	0.88	no
x00x0d_solaris_sparc_shell_reverse_tcp	196	sparc	armhf	32	32	big	little	0.552	no
x0a_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	0.882	no
x0ax0d_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	0.785	no
x0d_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	0.767	no
x5c_solaris_sparc_shell_bind_tcp	232	sparc	sh4	32	32	big	little	0.816	no
x5c_solaris_sparc_shell_find_port	136	sparc	sparc	32	32	big	big	0.986	yes
x5c_solaris_sparc_shell_reverse_tcp	196	sparc	sh4	32	32	big	little	0.846	no

Table 11. Detection results for SPARC using the fragment option

6.4 Analyzing the results of the scans

The overall detection accuracy in the code-only tests found in section 6.2 was 30,22%. For files under 2000 bytes the detection accuracy was 23,53%. It was also checked how well the program performs with different architectures. The performance was it its' best with the ARM architecture as the detection accuracy with these shellcodes was 70,83%. With the SPARC shellcodes the detection accuracy was 47,83%. With MIPS and PPC the performance was considerably worse as the detection accuracy with the MIPS shellcodes was 16,33% and 13,95% with the PowerPC shellcodes. One possibly noteworthy factor is that the program fared a lot better with the shellcodes which were generated without the MSFvenom's -b parameter. When this parameter is used to tell the program to avoid certain characters, by default it attempts to encode the generated shellcode as well (OffSec Services Limited 2020a). These encoded shellcodes made a huge negative impact on the detection accuracy of ISAdetect. When using the code-only option, the accuracy for the unencoded shellcodes was 56,90% and only 11.11% for the encoded shellcodes. Table 12 shows these results in a clear and simplified format.

Target of scan	Accuracy
Every file in set	30,22%
Files under 2000 bytes in size	23,53%
Unencoded files	56,90%
Encoded files	11,11%
ARM	70,83%
MIPS	16,33%
PowerPC	13,95%
SPARC	47,83%

Table 12. Overall detection results with the code-only option

The overall detection accuracy in the fragment tests found in section 6.3 was 29,50%. For files under 2000 bytes the detection accuracy was 25,51%, so it is almost the same as in the code-only tests but still slightly better. More testing is required in order to find out whether or not the fragment option performs considerably better with files under 2000 bytes than the code-only scanning option. Similarly to the code-only option, the fragment option performed the best with ARM architecture as the detection accuracy with these shellcodes was 50,00%. The second best performance was with the PowerPC architecture, here the detection accuracy was 32,56%. With the SPARC shellcodes the accuracy was 26,09% and finally, 18,37% with the MIPS shellcodes. Similarly to the code-only scans, when using the fragment option there was a huge difference in detection accuracy between encoded and unencoded shellcodes. With the fragment option, the detection accuracy for the unencoded shellcodes was 53,45% and 12,35% for the encoded shellcodes. Table 13 shows these results in a clear and simplified format.

Target of scan	Accuracy
Every file in set	29,50%
Files under 2000 bytes in size	25,51%
Unencoded files	53,45%
Encoded files	12,35%
ARM	50%
MIPS	18,37%
PowerPC	32,56%
SPARC	26,09%

Table 13. Overall detection results with the fragment option

Another factor to take into consideration is that the vast majority of the ARM architecture shellcodes are unencoded. As noted before, ISAdetect seems to handle unencoded shellcodes significantly better, so it is possible that the success with the ARM shellcodes is in part caused by this. Most likely further research with this matter would be a worthwhile thing to do.

6.5 Results from testing the classifiers

A set of 30 shellcodes was selected for testing the classifiers. The random forest classifier correctly detected the instruction set architecture from these shellcodes with the ratio of 46,67%. The other classifiers were tested in order to see whether they can beat this level of accuracy or does random forest preserve its' status as the highest performing classifier (Kairajärvi, Costin, and Hämäläinen 2020b).

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_linux_armbe_shell_bind_tcp	118	arm	m68k	32	32	big	big	0.16	no
no_bc_linux_armle_adduser	119	arm	mips	32	64	little	little	0.16	no
no_bc_linux_armle_adduser.elf	203	arm	ia64	32	64	little	little	0.15	no
no_bc_linux_armle_meterpreter_reverse_tcp	260	arm	arm	32	32	little	little	0.35	yes
no_bc_osx_armle_vibrate	16	arm	arm	32	32	little	little	0.24	yes
xff_linux_armle_shell_reverse_tcp.elf	256	arm	arm	32	32	little	little	0.24	yes
no_bc_linux_aarch64_meterpreter_reverse_https.elf	1092000	arm64	arm64	64	64	little	little	0.82	yes
no_bc_linux_aarch64_meterpreter_reverse_tcp.elf	332	arm64	arm64	64	64	little	little	0.25	yes
x0dxff_linux_aarch64_shell_reverse_tcp	152	arm64	arm64	64	64	little	little	0.23	yes
x0dxff_linux_aarch64_shell_reverse_tcp.elf	272	arm64	ia64	64	64	little	little	0.2	no
no_bc_linux_mipsbe_shell_bind_tcp	232	mips	mips	32	32	big	big	0.33	yes
no_bc_linux_mipsle_meterpreter_reverse_http.elf	1463172	mips	mips	32	32	little	little	0.84	yes
no_bc_linux_mipsle_shell_bind_tcp.elf	316	mips	mips	32	32	little	little	0.25	yes
x00x0d_linux_mipsbe_shell_reverse_tcp.elf	464	mips	m68k	32	32	big	big	0.12	no
x00x0d_linux_mipsle_shell_bind_tcp	340	mips	sparc	32	32	little	big	0.12	no
x5c_linux_mipsle_meterpreter_reverse_tcp	376	mips	arm64	32	64	little	little	0.15	no
no_bc_linux_mips64_meterpreter_reverse_https.elf	1568856	mips64	mips64	64	64	big	little	0.49	no
no_bc_linux_ppc_meterpreter_reverse_tcp.elf	1210840	ppc	ppc	32	32	big	big	0.47	yes
x00x0d_osx_ppc_shell_reverse_tcp	176	ppc	m68k	32	32	big	big	0.13	no
x0a_linux_ppc_shell_reverse_tcp	260	ppc	arm64	32	64	big	little	0.11	no
x0d_osx_ppc_shell_find_tag	76	ppc	m68k	32	32	big	big	0.12	no
no_bc_linux_ppc64_shell_reverse_tcp	183	ppc64	sparc	64	64	little	big	0.11	no
no_bc_linux_ppc64le_meterpreter_reverse_http.elf	1169208	ppc64	ppc64	64	64	little	little	0.93	yes
x5c_linux_ppc64_shell_find_port	171	ppc64	arm64	64	64	little	little	0.13	no
no_bc_bsd_sparc_shell_reverse_tcp	128	sparc	sparc	32	32	big	big	0.32	yes
x00_solaris_sparc_shell_find_port	188	sparc	sparc	32	32	big	big	0.14	yes
x00_solaris_sparc_shell_reverse_tcp	196	sparc	sh4	32	32	big	little	0.13	no
x00x0d_solaris_sparc_shell_bind_tcp	232	sparc	riscv	32	64	big	little	0.18	no
x0ax0d_solaris_sparc_shell_find_port	188	sparc	arm64	32	64	big	little	0.15	no
x5c_solaris_sparc_shell_bind_tcp	232	sparc	sparc	32	32	big	big	0.17	yes

Table 14. Detection results for random forest classifier

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_linux_armbe_shell_bind_tcp	118	arm	armhf	32	32	big	little	1	no
no_bc_linux_armle_adduser	119	arm	sh4	32	32	little	little	1	no
no_bc_linux_armle_adduser.elf	203	arm	ia64	32	64	little	little	1	no
no_bc_linux_armle_meterpreter_reverse_tcp	260	arm	arm	32	32	little	little	1	yes
no_bc_osx_armle_vibrate	16	arm	armhf	32	32	little	little	1	no
xff_linux_armle_shell_reverse_tcp.elf	256	arm	sparc	32	64	little	big	1	no
no_bc_linux_aarch64_meterpreter_reverse_https.elf	1092000	arm64	arm64	64	64	little	little	1	yes
no_bc_linux_aarch64_meterpreter_reverse_tcp.elf	332	arm64	ia64	64	64	little	little	1	no
x0dxff_linux_aarch64_shell_reverse_tcp	152	arm64	ppc	64	32	little	big	1	no
x0dxff_linux_aarch64_shell_reverse_tcp.elf	272	arm64	ia64	64	64	little	little	1	no
no_bc_linux_mipsbe_shell_bind_tcp	232	mips	sh4	32	32	big	little	1	no
no_bc_linux_mipsle_meterpreter_reverse_http.elf	1463172	mips	mips	32	32	little	little	1	yes
no_bc_linux_mipsle_shell_bind_tcp.elf	316	mips	m68k	32	32	little	big	1	no
x00x0d_linux_mipsbe_shell_reverse_tcp.elf	464	mips	m68k	32	32	big	big	1	no
x00x0d_linux_mipsle_shell_bind_tcp	340	mips	sh4	32	32	little	little	1	no
x5c_linux_mipsle_meterpreter_reverse_tcp	376	mips	riscv	32	64	little	little	1	no
no_bc_linux_mips64_meterpreter_reverse_https.elf	1568856	mips64	ia64	64	64	big	little	1	no
no_bc_linux_ppc_meterpreter_reverse_tcp.elf	1210840	ppc	ppc	32	32	big	big	1	yes
x00x0d_osx_ppc_shell_reverse_tcp	176	ppc	sh4	32	32	big	little	1	no
x0a_linux_ppc_shell_reverse_tcp	260	ppc	sh4	32	32	big	little	1	no
x0d_osx_ppc_shell_find_tag	76	ppc	sh4	32	32	big	little	1	no
no_bc_linux_ppc64_shell_reverse_tcp	183	ppc64	sh4	64	32	little	little	1	no
no_bc_linux_ppc64le_meterpreter_reverse_http.elf	1169208	ppc64	ppc64	64	64	little	big	1	no
x5c_linux_ppc64_shell_find_port	171	ppc64	sh4	64	32	little	little	1	no
no_bc_bsd_sparc_shell_reverse_tcp	128	sparc	sparc	32	32	big	big	1	yes
x00_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	1	no
x00_solaris_sparc_shell_reverse_tcp	196	sparc	sh4	32	32	big	little	1	no
x00x0d_solaris_sparc_shell_bind_tcp	232	sparc	sh4	32	32	big	little	1	no
x0ax0d_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	1	no
x5c_solaris_sparc_shell_bind_tcp	232	sparc	sh4	32	32	big	little	1	no

Table 15. Detection results for 1 nearest neighbor classifier

The 1 nearest neighbor classifier detected the instruction set architecture from this set of shellcodes with the accuracy of 16,67%. This classifier gave a very high prediction probability regardless of whether the prediction was correct or not.

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_linux_armbe_shell_bind_tcp	118	arm	armhf	32	32	big	little	0.667	no
no_bc_linux_armle_adduser	119	arm	sh4	32	32	little	little	1	no
no_bc_linux_armle_adduser.elf	203	arm	ia64	32	64	little	little	1	no
no_bc_linux_armle_meterpreter_reverse_tcp	260	arm	arm	32	32	little	little	1	yes
no_bc_osx_armle_vibrate	16	arm	armhf	32	32	little	little	1	no
xff_linux_armle_shell_reverse_tcp.elf	256	arm	ia64	32	64	little	little	0.667	no
no_bc_linux_aarch64_meterpreter_reverse_https.elf	1092000	arm64	arm64	64	64	little	little	1	yes
no_bc_linux_aarch64_meterpreter_reverse_tcp.elf	332	arm64	ia64	64	64	little	little	1	no
x0dxff_linux_aarch64_shell_reverse_tcp	152	arm64	arm64	64	64	little	little	0.667	yes
x0dxff_linux_aarch64_shell_reverse_tcp.elf	272	arm64	ia64	64	64	little	little	1	no
no_bc_linux_mipsbe_shell_bind_tcp	232	mips	sh4	32	32	big	little	1	no
no_bc_linux_mipsle_meterpreter_reverse_http.elf	1463172	mips	mips	32	32	little	little	0.667	yes
no_bc_linux_mipsle_shell_bind_tcp.elf	316	mips	m68k	32	32	little	big	1	no
x00x0d_linux_mipsbe_shell_reverse_tcp.elf	464	mips	m68k	32	32	big	big	0.667	no
x00x0d_linux_mipsle_shell_bind_tcp	340	mips	sh4	32	32	little	little	1	no
x5c_linux_mipsle_meterpreter_reverse_tcp	376	mips	riscv	32	64	little	little	1	no
no_bc_linux_mips64_meterpreter_reverse_https.elf	1568856	mips64	ia64	64	64	big	little	1	no
no_bc_linux_ppc_meterpreter_reverse_tcp.elf	1210840	ppc	ppc	32	32	big	big	1	yes
x00x0d_osx_ppc_shell_reverse_tcp	176	ppc	sh4	32	32	big	little	1	no
x0a_linux_ppc_shell_reverse_tcp	260	ppc	sh4	32	32	big	little	1	no
x0d_osx_ppc_shell_find_tag	76	ppc	sh4	32	32	big	little	1	no
no_bc_linux_ppc64_shell_reverse_tcp	183	ppc64	sh4	64	32	little	little	1	no
no_bc_linux_ppc64le_meterpreter_reverse_http.elf	1169208	ppc64	ppc64	64	64	little	big	0.667	no
x5c_linux_ppc64_shell_find_port	171	ppc64	sh4	64	32	little	little	1	no
no_bc_bsd_sparc_shell_reverse_tcp	128	sparc	sparc	32	32	big	big	0.667	yes
x00_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	1	no
x00_solaris_sparc_shell_reverse_tcp	196	sparc	sh4	32	32	big	little	1	no
x00x0d_solaris_sparc_shell_bind_tcp	232	sparc	sh4	32	32	big	little	1	no
x0ax0d_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	1	no
x5c_solaris_sparc_shell_bind_tcp	232	sparc	sh4	32	32	big	little	1	no

Table 16. Detection results for 3 nearest neighbor classifier

The 3 nearest neighbor detected the instruction set architecture correctly with the accuracy of 20%. Prediction probability was high throughout the tests while occasionally dipping to 0.667.

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_linux_armbe_shell_bind_tcp	118	arm	x32	32	32	big	little	1	no
no_bc_linux_armle_adduser	119	arm	x32	32	32	little	little	1	no
no_bc_linux_armle_adduser.elf	203	arm	sh4	32	32	little	little	1	no
no_bc_linux_armle_meterpreter_reverse_tcp	260	arm	riscv	32	64	little	little	1	no
no_bc_osx_armle_vibrate	16	arm	sparc	32	64	little	big	1	no
xff_linux_armle_shell_reverse_tcp.elf	256	arm	ia64	32	64	little	little	1	no
no_bc_linux_aarch64_meterpreter_reverse_https.elf	1092000	arm64	arm64	64	64	little	little	1	yes
no_bc_linux_aarch64_meterpreter_reverse_tcp.elf	332	arm64	ia64	64	64	little	little	1	no
x0dxff_linux_aarch64_shell_reverse_tcp	152	arm64	sparc	64	32	little	big	1	no
x0dxff_linux_aarch64_shell_reverse_tcp.elf	272	arm64	ia64	64	64	little	little	1	no
no_bc_linux_mipsbe_shell_bind_tcp	232	mips	sparc	32	64	big	big	1	no
no_bc_linux_mipsle_meterpreter_reverse_http.elf	1463172	mips	mips	32	32	little	little	1	yes
no_bc_linux_mipsle_shell_bind_tcp.elf	316	mips	mips	32	32	little	little	1	yes
x00x0d_linux_mipsbe_shell_reverse_tcp.elf	464	mips	arm64	32	64	big	little	1	no
x00x0d_linux_mipsle_shell_bind_tcp	340	mips	sh4	32	32	little	little	1	no
x5c_linux_mipsle_meterpreter_reverse_tcp	376	mips	sh4	32	32	little	little	1	no
no_bc_linux_mips64_meterpreter_reverse_https.elf	1568856	mips64	ia64	64	64	big	little	1	no
no_bc_linux_ppc_meterpreter_reverse_tcp.elf	1210840	ppc	ppcspe	32	32	big	big	1	no
x00x0d_osx_ppc_shell_reverse_tcp	176	ppc	i386	32	32	big	little	1	no
x0a_linux_ppc_shell_reverse_tcp	260	ppc	ppc64	32	64	big	big	1	no
x0d_osx_ppc_shell_find_tag	76	ppc	riscv	32	64	big	little	1	no
no_bc_linux_ppc64_shell_reverse_tcp	183	ppc64	sparc	64	64	little	big	1	no
no_bc_linux_ppc64le_meterpreter_reverse_http.elf	1169208	ppc64	ppc64	64	64	little	little	1	yes
x5c_linux_ppc64_shell_find_port	171	ppc64	sparc	64	64	little	big	1	no
no_bc_bsd_sparc_shell_reverse_tcp	128	sparc	sparc	32	32	big	big	1	yes
x00_solaris_sparc_shell_find_port	188	sparc	sparc	32	32	big	big	1	yes
x00_solaris_sparc_shell_reverse_tcp	196	sparc	x32	32	32	big	little	1	no
x00x0d_solaris_sparc_shell_bind_tcp	232	sparc	sparc	32	64	big	big	1	no
x0ax0d_solaris_sparc_shell_find_port	188	sparc	hppa	32	32	big	big	1	no
x5c_solaris_sparc_shell_bind_tcp	232	sparc	amd64	32	64	big	little	1	no

Table 17. Detection results for decision tree classifier

The decision tree classifier managed an accuracy of 20% as well. Prediction probability was 1 on both correct and incorrect predictions.

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_linux_armbe_shell_bind_tcp	118	arm	armhf	32	32	big	little	1	no
no_bc_linux_armle_adduser	119	arm	armhf	32	32	little	little	1	no
no_bc_linux_armle_adduser.elf	203	arm	armhf	32	32	little	little	1	no
no_bc_linux_armle_meterpreter_reverse_tcp	260	arm	armhf	32	32	little	little	1	no
no_bc_osx_armle_vibrate	16	arm	armhf	32	32	little	little	1	no
xff_linux_armle_shell_reverse_tcp.elf	256	arm	armhf	32	32	little	little	1	no
no_bc_linux_aarch64_meterpreter_reverse_https.elf	1092000	arm64	arm64	64	64	little	little	1	yes
no_bc_linux_aarch64_meterpreter_reverse_tcp.elf	332	arm64	arm64	64	64	little	little	0.999	yes
x0dxff_linux_aarch64_shell_reverse_tcp	152	arm64	sh4	64	32	little	little	1	no
x0dxff_linux_aarch64_shell_reverse_tcp.elf	272	arm64	sparc	64	32	little	big	1	no
no_bc_linux_mipsbe_shell_bind_tcp	232	mips	mips	32	32	big	big	1	yes
no_bc_linux_mipsle_meterpreter_reverse_http.elf	1463172	mips	mips	32	32	little	little	1	yes
no_bc_linux_mipsle_shell_bind_tcp.elf	316	mips	mips	32	32	little	little	1	yes
x00x0d_linux_mipsbe_shell_reverse_tcp.elf	464	mips	riscv	32	64	big	little	1	no
x00x0d_linux_mipsle_shell_bind_tcp	340	mips	sh4	32	32	little	little	1	no
x5c_linux_mipsle_meterpreter_reverse_tcp	376	mips	armhf	32	32	little	little	1	no
no_bc_linux_mips64_meterpreter_reverse_https.elf	1568856	mips64	m68k	64	32	big	big	1	no
no_bc_linux_ppc_meterpreter_reverse_tcp.elf	1210840	ppc	ppc	32	32	big	big	1	yes
x00x0d_osx_ppc_shell_reverse_tcp	176	ppc	armhf	32	32	big	little	1	no
x0a_linux_ppc_shell_reverse_tcp	260	ppc	armhf	32	32	big	little	1	no
x0d_osx_ppc_shell_find_tag	76	ppc	armhf	32	32	big	little	1	no
no_bc_linux_ppc64_shell_reverse_tcp	183	ppc64	m68k	64	32	little	big	1	no
no_bc_linux_ppc64le_meterpreter_reverse_http.elf	1169208	ppc64	ppc64	64	64	little	little	1	yes
x5c_linux_ppc64_shell_find_port	171	ppc64	m68k	64	32	little	big	1	no
no_bc_bsd_sparc_shell_reverse_tcp	128	sparc	armhf	32	32	big	little	1	no
x00_solaris_sparc_shell_find_port	188	sparc	armhf	32	32	big	little	1	no
x00_solaris_sparc_shell_reverse_tcp	196	sparc	armhf	32	32	big	little	1	no
x00x0d_solaris_sparc_shell_bind_tcp	232	sparc	armhf	32	32	big	little	1	no
x0ax0d_solaris_sparc_shell_find_port	188	sparc	armhf	32	32	big	little	1	no
x5c_solaris_sparc_shell_bind_tcp	232	sparc	armhf	32	32	big	little	0.982	no

Table 18. Detection results for naïve Bayes classifier

When the naïve Bayes classifier was used, the detection accuracy was 23,33%. This classifier gave a high prediction probability as well on both correct and incorrect detections.

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_linux_armbe_shell_bind_tcp	118	arm	alpha	32	64	big	little	0.681	no
no_bc_linux_armle_adduser	119	arm	m68k	32	32	little	big	0.993	no
no_bc_linux_armle_adduser.elf	203	arm	ppcspe	32	32	little	big	0.984	no
no_bc_linux_armle_meterpreter_reverse_tcp	260	arm	arm	32	32	little	little	0.962	yes
no_bc_osx_armle_vibrate	16	arm	arm	32	32	little	little	0.72	yes
xff_linux_armle_shell_reverse_tcp.elf	256	arm	arm	32	32	little	little	0.89	yes
no_bc_linux_aarch64_meterpreter_reverse_https.elf	1092000	arm64	arm64	64	64	little	little	0.608	yes
no_bc_linux_aarch64_meterpreter_reverse_tcp.elf	332	arm64	sh4	64	32	little	little	0.999	no
x0dxff_linux_aarch64_shell_reverse_tcp	152	arm64	m68k	64	32	little	big	0.993	no
x0dxff_linux_aarch64_shell_reverse_tcp.elf	272	arm64	m68k	64	32	little	big	0.991	no
no_bc_linux_mipsbe_shell_bind_tcp	232	mips	mips	32	32	big	little	0.586	no
no_bc_linux_mipsle_meterpreter_reverse_http.elf	1463172	mips	mips	32	32	little	little	0.862	yes
no_bc_linux_mipsle_shell_bind_tcp.elf	316	mips	amd64	32	64	little	little	0.993	no
x00x0d_linux_mipsbe_shell_reverse_tcp.elf	464	mips	mips64	32	64	big	little	0.989	no
x00x0d_linux_mipsle_shell_bind_tcp	340	mips	amd64	32	64	little	little	0.974	no
x5c_linux_mipsle_meterpreter_reverse_tcp	376	mips	m68k	32	32	little	big	0.993	no
no_bc_linux_mips64_meterpreter_reverse_https.elf	1568856	mips64	mips64	64	64	big	little	0.731	no
no_bc_linux_ppc_meterpreter_reverse_tcp.elf	1210840	ppc	ppcspe	32	32	big	big	0.996	no
x00x0d_osx_ppc_shell_reverse_tcp	176	ppc	ppc	32	32	big	big	0.945	yes
x0a_linux_ppc_shell_reverse_tcp	260	ppc	sh4	32	32	big	little	0.960	no
x0d_osx_ppc_shell_find_tag	76	ppc	arm64	32	64	big	little	0.993	no
no_bc_linux_ppc64_shell_reverse_tcp	183	ppc64	m68k	64	32	little	big	0.984	no
no_bc_linux_ppc64le_meterpreter_reverse_http.elf	1169208	ppc64	armhf	64	32	little	little	0.931	no
x5c_linux_ppc64_shell_find_port	171	ppc64	m68k	64	32	little	big	0.980	no
no_bc_bsd_sparc_shell_reverse_tcp	128	sparc	sparc	32	32	big	big	0.954	yes
x00_solaris_sparc_shell_find_port	188	sparc	armhf	32	32	big	little	0.553	no
x00_solaris_sparc_shell_reverse_tcp	196	sparc	armhf	32	32	big	little	0.993	no
x00x0d_solaris_sparc_shell_bind_tcp	232	sparc	armhf	32	32	big	little	0.645	no
x0ax0d_solaris_sparc_shell_find_port	188	sparc	ppcspe	32	32	big	big	0.998	no
x5c_solaris_sparc_shell_bind_tcp	232	sparc	armhf	32	32	big	little	0.446	no

Table 19. Detection results for neural net classifier

Similarly to the naïve Bayes, the neural net classifier achieved the accuracy of 23,33% as well. In this case as well the probability of prediction was quite high throughout the tests regardless of whether the shellcode's instruction set architecture was detected correctly or not.

Shellcode	File size	Architecture	Detected architecture	Word size	Detected word size	Endianness	Detected endianness	Prediction probability	Correct
no_bc_linux_armbe_shell_bind_tcp	118	arm	sh4	32	32	big	little	0.719	no
no_bc_linux_armle_adduser	119	arm	sh4	32	32	little	little	0.933	no
no_bc_linux_armle_adduser.elf	203	arm	ia64	32	64	little	little	0.585	no
no_bc_linux_armle_meterpreter_reverse_tcp	260	arm	arm	32	32	little	little	0.917	yes
no_bc_osx_armle_vibrate	16	arm	armhf	32	32	little	little	0.977	no
xff_linux_armle_shell_reverse_tcp.elf	256	arm	ia64	32	64	little	little	0.98	no
no_bc_linux_aarch64_meterpreter_reverse_https.elf	1092000	arm64	arm64	64	64	little	little	0.801	yes
no_bc_linux_aarch64_meterpreter_reverse_tcp.elf	332	arm64	ia64	64	64	little	little	0.933	no
x0dxff_linux_aarch64_shell_reverse_tcp	152	arm64	arm64	64	64	little	little	0.681	yes
x0dxff_linux_aarch64_shell_reverse_tcp.elf	272	arm64	ia64	64	64	little	little	0.999	no
no_bc_linux_mipsbe_shell_bind_tcp	232	mips	m68k	32	32	big	big	0.645	no
no_bc_linux_mipsle_meterpreter_reverse_http.elf	1463172	mips	mips	32	32	little	little	0.877	yes
no_bc_linux_mipsle_shell_bind_tcp.elf	316	mips	mips64	32	64	little	little	0.2	no
x00x0d_linux_mipsbe_shell_reverse_tcp.elf	464	mips	sh4	32	32	big	little	0.281	no
x00x0d_linux_mipsle_shell_bind_tcp	340	mips	sh4	32	32	little	little	0.895	no
x5c_linux_mipsle_meterpreter_reverse_tcp	376	mips	sh4	32	32	little	little	0.58	no
no_bc_linux_mips64_meterpreter_reverse_https.elf	1568856	mips64	ia64	64	64	big	little	0.488	no
no_bc_linux_ppc_meterpreter_reverse_tcp.elf	1210840	ppc	ppc	32	32	big	big	0.877	yes
x00x0d_osx_ppc_shell_reverse_tcp	176	ppc	sh4	32	32	big	little	0.768	no
x0a_linux_ppc_shell_reverse_tcp	260	ppc	sh4	32	32	big	little	0.759	no
x0d_osx_ppc_shell_find_tag	76	ppc	ppcspe	32	32	big	big	0.879	no
no_bc_linux_ppc64_shell_reverse_tcp	183	ppc64	ppc64	64	64	little	little	0.957	yes
no_bc_linux_ppc64le_meterpreter_reverse_http.elf	1169208	ppc64	ppc64	64	64	little	big	0.912	no
x5c_linux_ppc64_shell_find_port	171	ppc64	ppc64	64	64	little	little	0.951	yes
no_bc_bsd_sparc_shell_reverse_tcp	128	sparc	sparc	32	32	big	big	0.999	yes
x00_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	0.859	no
x00_solaris_sparc_shell_reverse_tcp	196	sparc	sh4	32	32	big	little	0.939	no
x00x0d_solaris_sparc_shell_bind_tcp	232	sparc	sh4	32	32	big	little	0.755	no
x0ax0d_solaris_sparc_shell_find_port	188	sparc	sh4	32	32	big	little	0.8	no
x5c_solaris_sparc_shell_bind_tcp	232	sparc	sh4	32	32	big	little	0.822	no

Table 20. Detection results for SVM/SMO classifier

Finally, the SVM/SMO classifier correctly detected the instruction set architecture with the accuracy of 26,67% so in these tests it was the highest performing classifier after random forest. Prediction probability was lower than with the other classifiers but still relatively high in most cases and it did not seem to matter whether the prediction was correct or not.

6.6 Results of MSFvenom bad character analysis

This section presents the results of the MSFvenom bad character analyses in several tables, which show the architecture, the amount of accepted and rejected byte combinations, and the used parameters. The rejected byte combinations for each test can be viewed in detail in appendix C. The used payloads for each test can be seen in section 5.3. In this section, it was not inspected whether or not each generated shellcode is unique as the point was to examine

which byte combinations are accepted and which are not.

Architecture	Number of accepted bytes	Number of rejected bytes
x86	256	0
x64	255	1
MIPS	248	8
ARM	241	15
ARM 64	218	38
PowerPC	238	18
PowerPC 64	188	68
SPARC	241	15

Table 21. Results of MSFvenom bad character analysis

Based on this test, most one byte combinations were accepted and the number of rejected bytes was low for almost every architecture. The only exceptions are ARM 64 with 38 rejections and PowerPC 64 with 68 rejections.

Architecture	LHOST	Number of accepted bytes	Number of rejected bytes	LHOST	Number of accepted bytes	Number of rejected bytes
x86	192.168.1.1	255	1	10.0.0.1	255	1
x64	192.168.1.1	254	2	10.0.0.1	254	2
MIPS	192.168.1.1	231	25	10.0.0.1	231	25
ARM	192.168.1.1	209	47	10.0.0.1	210	46
ARM 64	192.168.1.1	219	37	10.0.0.1	219	37
PowerPC	192.168.1.1	238	18	10.0.0.1	238	18
PowerPC 64	192.168.1.1	182	74	10.0.0.1	181	75
SPARC	192.168.1.1	247	9	10.0.0.1	247	9

Table 22. MSFvenom LHOST analysis

Based on this test, it seems that changing the LHOST parameter in shellcodes that create a reverse shell connection makes minimal difference. The tests were conducted with two different LHOST values: 192.168.1.1 and 10.0.0.1. In most cases, the number of accepted bytes and the number of rejected bytes is the same. The only exceptions are the ARM architecture and the PowerPC 64 architecture. Changing the LHOST parameter in the ARM tests caused one less rejection and in the PowerPC 64 tests it caused one more rejection.

Architecture	RHOST	Number of accepted bytes	Number of rejected bytes	RHOST	Number of accepted bytes	Number of rejected bytes
x86	-	256	0	124.173.232.109	256	0
x64	-	254	2	124.173.232.109	254	2
MIPS	-	237	19	124.173.232.109	237	19
ARM	-	205	51	124.173.232.109	205	51
PowerPC	-	233	23	124.173.232.109	233	23
PowerPC 64	-	183	73	124.173.232.109	183	73
SPARC	-	246	10	124.173.232.109	246	10

Table 23. MSFvenom RHOST analysis

MSFvenom payloads which execute a bind shell have an optional RHOST parameter which was tested in part of the thesis. First, the RHOST parameter was not used at all and then another set of shellcodes were generated with the RHOST parameter enabled and the value was a random IP address: 124.173.232.109. Based on this experiment, changing or enabling the RHOST parameter does not make a significant difference when generating shellcodes with different one byte combinations as bad characters. Mostly the number of accepted and rejected bytes is the same, except for the MIPS architecture and the SPARC architecture. Changing the RHOST parameter resulted in one more rejection in the MIPS tests and one less rejection in the SPARC tests.

Architecture	LPORT	Number of accepted bytes	Number of rejected bytes	LPORT	Number of accepted bytes	Number of rejected bytes
x86	1234	256	0	10	256	0
x64	1234	254	2	10	254	2
MIPS	1234	237	19	10	236	20
ARM	1234	206	50	10	206	50
PowerPC	1234	233	23	10	233	23
PowerPC 64	1234	182	74	10	182	74
SPARC	1234	245	11	10	246	10

Table 24. MSFvenom LPORT analysis

In this test, the used payloads were the same as in the previous test, but the tested parameter was different. This time the focus was on the LPORT parameter, and two different values were used in the tests: 1234 and 10. Based on this test, changing the LPORT parameter makes very little difference in the shellcode creation. In most cases the number of accepted and rejected bytes are the same with MIPS and SPARC being the only exceptions. Changing the LPORT value caused one more rejection for the MIPS architecture and one less rejection for the SPARC architecture.

Next, this thesis will present some statistics about the bytes which caused the most rejections. The 10 most problematic bytes are listed in the following five tables. Table 25 presents the overall 10 most problematic bytes, table 26 presents the 10 most problematic bytes in the first MSFvenom bad character analysis, table 27 presents the 10 most problematic bytes in the LHOST analysis, table 28 presents the 10 most problematic bytes in the RHOST analysis and finally table 29 presents the 10 most problematic bytes in the LPORT analysis.

Byte	Count
0xff	39
0x01	30
0x02	30
0x40	28
0xe0	28
0x04	26
0x03	24
0x20	20
0x2f	20
0x05	18

Table 25. Overall, the 10 most problematic bytes

Byte	Count
0x04	5
0x01	4
0x02	4
0x03	4
0x40	4
0x21	3
0x08	3
0xe1	3
0xf9	3
0xff	3

Table 26. The 10 most problematic bytes in MSFvenom bad character analysis

Byte	Count
0xff	12
0x01	10
0x02	10
0xe0	10
0x03	8
0x04	8
0x40	8
0x80	8
0xc0	8
0x0b	6

Table 27. The 10 most problematic bytes in LHOST analysis

Byte	Count
0xff	12
0x01	8
0x02	8
0x40	8
0xe0	8
0x04	6
0x05	6
0x0c	6
0x20	6
0x2f	6

Table 28. The 10 most problematic bytes in RHOST analysis

Byte	Count
0xff	12
0x01	8
0x02	8
0x40	8
0xe0	8
0x04	7
0x05	6
0x0c	6
0x20	6
0x2f	6

Table 29. The 10 most problematic bytes in LPORT analysis

7 Discussion

The research problem in this thesis was divided into two parts. The first one was to create a representable real-world database of shellcodes and the second one was to see how accurately can a machine learning based application detect the instruction set architecture from the shellcodes of this database. The two main research question were:
RQ1: How to create a significant and representative real-world database of shellcodes?
RQ2: How accurately can a machine learning based ISA identification system detect the correct CPU architecture, word size and endianness from short shellcodes?
Also, this thesis had three sub-questions which were:
SQ1: How to automate the creation of the shellcode database?
SQ2: How can machine learning based ISA identification systems be improved?
SQ3: Which different one byte combinations MSFvenom accepts or rejects as bad characters, and what are the most problematic bytes?

The answer to RQ1 is:
Without skill to personally create shellcodes from scratch, a database like this can be created by using various credible and high-quality sources in the Internet such as Exploit Database and Shell-Storm as well as using dedicated software such as MSFvenom to generate shellcodes. MSFvenom can be executed multiple times with different parameters in order to create slightly different shellcodes from the same payload. After collecting enough shellcodes from these sources, they can be sorted by target architecture and then these collections can be further molded up to the point where each collection contains the same amount of shellcodes as the other.

The answer to RQ2 is:
Based on the tests performed in this thesis, a machine learning based ISA detection system can detect the instruction set architecture from short shellcodes with the accuracy of about 30%. Two different detection options were tested, in the first the program was set to scan code-only sections of the shellcode files and in the second the program was set to scan small fragments. The detection accuracy in the code-only tests was 30,22% and 29,5% in

the fragment tests. For smaller files of under 2000 bytes in size, the detection accuracy with the code-only option was 23,53% and 25,51% with the fragment option. In addition, based on the tests conducted in this thesis, it is easier to detect the instuction set architecture from unencoded shellcode files than from encoded ones. The code-only option reached the accuracy of 56,90% with unencoded shellcodes and 11,11% with encoded files. The fragment option achieved the accuracy of 53,45% with unencoded shellcodes and 12,35% with encoded ones.

The answer to SQ1 is:
In this thesis, the process of generating shellcodes with MSFvenom was automated by creating a Python program which runs MSFvenom in a loop. The other phases of collection were not automated as it was easy enough to download the Exploit Database codes the project's GitHub repository and use wget to download every piece of shellcode from Shell-Storm. However, the potential of automation was not fully realized in this thesis. It is most likely possible to automate the whole process of collecting shellcodes and creating the database by programming a tool which downloads shellcodes from given sources and then sorts them by architecture for example. The same tool could also automatically maintain this database.

The answer to SQ2 is:
Based on the findings of this thesis, and the answer to RQ2, one deduction is that machine learning based systems can be improved by specifically training them to recognize and detect the desired objects, which in the case of this thesis are shellcodes, in the form they are usually encountered in real-world situations. Bell (2014) notes that machine learning systems which use supervised learning can be improved by improving the classifiers. In the case of this thesis, this can be achieved by manually providing the correct ISA for each shellcode so that the program can learn to recognize it more accurately (Bell 2014, 3).

The answer to SQ3 is:
Based on the tests conducted in this thesis, MSFvenom accepts most one byte combinations as bad characters. In most cases, the number of rejected byte combinations was relatively low in the tests performed in this thesis. Generally, in all tests the number of rejected bytes was the lowest with x86, x64 and SPARC architectures and the highest with ARM, ARM 64 and PowerPC 64 architectures. When generating shellcodes with different one byte combinations

as bad characters, the impact of changing different parameters such as local IP address, target IP address or local port number was minimal. Based on the experimentation conducted in this thesis, it seems that at least these aforementioned parameters can be configured relatively freely when generating MSFvenom shellcodes with bad characters. The number of accepted and rejected bytes are listed in section 6.6 and the rejected byte combinations for each test can be viewed in detail in appendix C. Overall the top 10 most problematic bytes in the tests conducted in this thesis were 0xff, 0x01, 0x02, 0x40, 0xe0, 0x04, 0x03, 0x20, 0x2f and 0x05.

The application which was tested in this thesis represents the state-of-the-art and is called ISAdetect. ISAdetect's dataset comprises ISO files, DEB files, ELF files and ELF code sections whose minimum size is 4000 bytes, and it has been trained with this data as well. In addition, this state-of-the-art tool performed very well, gaining high detection accuracy in the tests conducted by the researchers. With smaller test samples of just 8 bytes in size, the team achieved the best results with the SVM classifier which scanned these small samples with the accuracy of approximately 50% and most classifiers reached the accuracy of 90% with test samples of 4000 bytes in size (Kairajärvi, Costin, and Hämäläinen 2020b). Therefore, the results gained in this thesis are not in line with those gained in previous research. At a glance, most of the shellcode files scanned in this thesis fall in the range of 100-300 bytes in size, and when using the random forest classifier, the detection accuracy with these smaller files was about 23% with the code-only option and about 25% with the fragment option. In the tests performed by (Kairajärvi, Costin, and Hämäläinen 2020b), the same classifier achieved the accuracy of nearly 80% with test samples of 128 bytes size and at 256 bytes the accuracy was closing in on 90%. Partly this could be due to the fact that some of the shellcode files used in this thesis were encoded, and as stated before, this impacted heavily on the detection accuracy.

Currently ISAdetect supports many different architectures, but not x86, x86-64 and x64 for example (Kairajärvi, Costin, and Hämäläinen 2020b). Based on the observations made when collecting shellcodes for this thesis, the most common architecture for shellcodes is x86. This can be seen from tables 1 and 2. It might be worthwhile to add support for x86 and various other architectures as well, such as x86-64 and x64, if the intention is to develop the tool to

accurately detect the instruction set architecture from shellcodes as well.

7.1 Limitations

The set of shellcodes used in testing may not have been large and diverse enough and it is not evenly balanced when it comes to different architectures. This was largely because of the availability of shellcodes in the required formats, but perhaps more effort could have been placed into acquiring these kinds of shellcodes as well.

When compared to the amount of architectures that tools such as ISAdetect (Kairajärvi, Costin, and Hämäläinen 2020b) or radare2 (”The Official Radare2 Book” 2020) support, it is a small number. Largely these limitations were caused by the availability of shellcodes and the lack of skill to write more for certain architectures. In addition, only three different sources were used when collecting the shellcodes for the main database. Most likely there are more than just three sources from where to acquire shellcodes for research purposes in such as blogs and other databases and repositories.

7.2 Future work

First, the shellcode database and the collection method of the codes could be improved. A tool which automatically collects shellcodes from given sources and compiles them into a balanced database could be programmed. This tool could also be programmed to automatically maintain the database. When examining the architecture distribution in table 2, it is clear that some architectures are heavily underrepresented. Future work should put emphasis on collecting or writing more shellcode for these architectures in order to be able to reliably conduct tests on these architectures as well. In addition, as noted by Kairajärvi, Costin, and Hämäläinen (2020b), most likely new architectures are emerging due to the rising popularity of Iot. Therefore, this tool could also be programmed to somehow take this possibility in consideration, for example by writing code to which it is easy to add new features such as possible new architectures. Furthermore, in future research shellcodes could also be searched from other sources as well than the three which were used in this thesis.

Second, the Python scripts created for this thesis could be refactored and possibly combined into a one single program. Currently, the scripts have a lot of similar code which could be generalized. This would make the scripts easier to maintain and upgrade for any possible future usage.

Third, a comparison could be made with reverse engineering software such as radare2 to see if this kind method yields correct results faster or more accurately. Radare2 is a framework for reverse engineering which supports various architectures, file formats and operating systems (”The Official Radare2 Book” 2020).

Finally, the issues mentioned in section 6.4 should be taken into consideration in future research. More testing should be done with ISAdetect’s code-only and fragment options in order to see if a clear winner could be identified. In this thesis, the fragment option fared a little better with shellcodes of under 2000 bytes in size, but the difference compared to the code-only option was so small that it is difficult to declare fragment as the better option for scanning short shellcodes. Also, it was noticed that ISAdetect was most accurate with the ARM architecture shellcodes and also that the program in general seems to handle unencoded shellcodes better than encoded ones. The majority of the ARM shellcodes were in fact unencoded which may explain why ISAdetect had more success with these shellcodes. Future work should conduct more experiments in this area in order to figure out whether or not the success with the ARM shellcodes was caused by the fact that the majority of these files were unencoded.

8 Conclusion

The aim of this study was to examine how accurately a machine learning based ISA detection system detects the instruction set architecture from short shellcodes. This was achieved by building a representative real-world database of shellcodes for different architectures and then using these codes to create a set of shellcodes for the testing phase. After building the test set, the detection tool was tested against these shellcodes in order to find out how well it performs and also to gain an understanding of its' current functionality against shellcodes. The results of these tests suggested that the current training is not sufficient enough for detecting ISA from short shellcodes accurately and reliably as the overall detection accuracy was approximately 30%. However, occasionally the program was able to correctly detect ISA from very small shellcode files, for example from a file of just 16 bytes in size, which indicates that this program and in general these kinds of ISA detection methods show a lot of potential.

This was a quantitative study and the used research method was experimental research. The research itself was conducted in a virtual environment. This method was suitable and it enabled efficient and reliable testing and the virtual environment allowed to safely handle and store the shellcodes. In addition, this was a relevant study as according to Chen et al. (2016, 107) code injection is still a viable method of attack and will be in the future as well, unless software vulnerabilities suddenly disappear.

This thesis contributes a representative real-world database of approximately 20000 shellcodes which can be used to create training datasets for machine learning based ISA detection systems and information about how a state-of-the-art machine learning ISA detection tool currently performs against short shellcodes. In addition, this thesis contributes a set of Python scripts which automate the process of generating shellcodes with MSFvenom to some extent.

The limitations were identified and they mostly deal with the main database and the smaller database used in the tests, and the validity of the results. Future work should take these limitations into account. This research can be taken further by various ways. For example,

the main shellcode database can be improved and expanded to include more architectures and more shellcodes for certain architectures. Possible new emerging architectures can and should be added into the database as well, and a tool could be built, which automatically collects shellcodes and adds them into the database.

Despite the fact that the set goal of 90% could not be achieved in the tests conducted in this thesis, it can be said that the potential of machine learning based ISA detection systems remains high. The detection application tested in this thesis showed potential by correctly detecting the instruction set architecture from very short shellcodes whose file size was in the range of 100-200 bytes, and the smallest correctly detected shellcode file being just 16 bytes in size. Most likely with more training better results can be achieved and currently there is no reason to believe that machine learning based ISA detection tools are not able to handle short shellcodes efficiently and reliably.

Bibliography

Abd-El-Barr, Mostafa, and Hesham El-Rewini. 2005. *Fundamentals of Computer Organization and Architecture.* Wiley Series on Parallel and Distributed Computing. Hoboken, N.J.: Wiley-Interscience.

Anley, Chris, John Heasman, Felix "FX" Linder, and Gerardo Richarte. 2007. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes.* 2nd edition. Wiley Publishing, Inc.

Atlam, Hany F., Robert J. Walters, and Gary B. Wills. 2018. "Intelligence of Things: Opportunities Challenges" (): 1–6. Visited on April 6, 2020. https://ieeexplore.ieee.org/document/8627114.

Bell, Jason. 2014. *Machine Learning: Hands-On for Developers and Technical Professionals.* Somerset, UNITED STATES: John Wiley & Sons, Incorporated.

Borders, Kevin, Atul Prakash, and Mark Zielinski. 2007. "Spector: Automatically Analyzing Shell Code" (): 501–514. Visited on August 5, 2020. https://ieeexplore.ieee.org/document/4413015.

Brinda, Gladis, and Geogen George. 2016. "Detection and Analysis of Shellcode in Malicious Documents". *International Journal of Computer Science and Network* 5 (2): 310–314. Visited on May 6, 2020. https://www.researchgate.net/publication/317165959_Detection_and_Analysis_of_Shellcode_in_Malicious_Documents.

"Buffer Overflow | OWASP". 2020. Library Catalog: owasp.org. https://owasp.org/www-community/vulnerabilities/Buffer_Overflow.

Chen, Mo, Changzhen Hu, Donghai Tian, Xin Wang, Yuan Liu, and Ning Li. 2016. "Shellix: An Efficient Approach for Shellcode Detection". *International Journal of Security and Its Applications* 10 (): 107–122.

Clemens, John. 2015. ”Automatic Classification of Object Code Using Machine Learning”. *Digital Investigation* 14 (): S156–S162. Visited on June 17, 2020. https://doi.org/10.1016/j.diin.2015.05.007.

Deng, Li. 2014. ”A tutorial survey of architectures, algorithms, and applications for deep learning”. *APSIPA Transactions on Signal and Information Processing* 3:e2. Visited on April 4, 2020. https://www.cambridge.org/core/product/identifier/S2048770313000097/type/journal_article.

Edgar, Thomas W., and David O. Manz. 2017. *Research Methods for Cyber Security.* Cambridge, MA: Syngress.

”Endianness”. MDN Web Docs. 2020. Library Catalog: developer.mozilla.org. Visited on June 17, 2020. https://developer.mozilla.org/en-US/docs/Glossary/Endianness.

Fernandes de Mello, Rodrigo, and Moacir Antonelli Ponti. 2018. *Machine Learning: A Practical Approach on the Statistical Learning Theory.* Cham: Springer International Publishing.

Foster, James C., and Vincent Liu. 2006. *Writing Security Tools and Exploits.* Rockland, MA: Syngress. Visited on January 2, 2021.

Foster, James C., Vitaly Osipov, Nish Bhalla, and Niels Heinen. 2005. *Buffer Overflow Attacks : Detect, Exploit, Prevent.* Rockland, MA: Syngress.

Foster, James C., and Mike Price. 2005. *Sockets, Shellcode, Porting, and Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals.* Rockland, Mass: Syngress.

Fox, Anthony, and Magnus O. Myreen. 2010. ”A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture”. In *Interactive Theorem Proving,* edited by Matt Kaufmann and Lawrence C. Paulson, redacted by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, et al., 6172:243–258. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg. Visited on June 17, 2020. http://link.springer.com/10.1007/978-3-642-14052-5_18.

Garnham, Alan. 1987. *Artificial Intelligence: An Introduction.* Milton, UNITED KINGDOM: Routledge.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press.

Han, Jiawei, and Micheline Kamber. 2011. *Data Mining: Concepts and Techniques.* Volume 3rd ed. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA: Morgan Kaufmann.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning : Data Mining, Inference, and Prediction.* 2nd edition. Springer Series in Statistics. New York: Springer-Verlag.

Japkowicz, Nathalie. 2000. "The Class Imbalance Problem: Significance and Strategies". *Proceedings of the 2000 International Conference on Artificial Intelligence ICAI* ().

Kairajärvi, Sami, Andrei Costin, and Timo Hämäläinen. 2019. "Towards usable automated detection of CPU architecture and endianness for arbitrary binary files and object code sequences" (). Visited on August 5, 2020. https://www.researchgate.net/publication/335201405_Towards_usable_automated_detection_of_CPU_architecture_and_endianness_for_arbitrary_binary_files_and_object_code_sequences.

———. 2020a. *ISAdetect.* Original-date: 2019-08-18T14:58:41Z. Visited on November 28, 2020. https://github.com/kairis/isadetect.

———. 2020b. "ISAdetect: Usable Automated Detection of CPU Architecture and Endianness for Executable Binary Files and Object Code", (New York, NY, USA), CODASPY '20 (): 376–380. Visited on November 11, 2020. doi:10.1145/3374664.3375742. https://doi.org/10.1145/3374664.3375742.

Kaplan, Jerry. 2016. *Artificial Intelligence: What Everyone Needs to Know.* Oxford, UNITED STATES: Oxford University Press, Incorporated.

Lantz, Brett. 2013. *Machine Learning with R : Learn How to Use R to Apply Powerful Machine Learning Methods and Gain an Insight Into Real-world Applications.* Birmingham, UK: Packt Publishing.

Mohammed, Mohssen, Muhammad Badruddin Khan, and Eihab Bashier Mohammed Bashier. 2017. *Machine Learning : Algorithms and Applications.* CRC Press.

Murty, M.N., and Rashmi Raghava. 2016. *Support Vector Machines and Perceptrons: Learning, Optimization, Classification, and Application to Social Networks.* SpringerBriefs in Computer Science. Cham: Springer International Publishing.

OffSec Services Limited. 2020a. "MSFvenom | Offensive Security". Visited on November 6, 2020. https://www.offensive-security.com/metasploit-unleashed/msfvenom/.

———. 2020b. "Offensive Security's Exploit Database Archive". Visited on November 6, 2020. https://www.exploit-db.com/shellcodes.

———. 2020c. "Our Most Advanced Penetration Testing Distribution, Ever." Visited on November 6, 2020. https://www.kali.org/.

———. 2020d. *The Exploit Database Git Repository.* Original-date: 2013-12-03T18:50:07Z. Visited on November 6, 2020. https://github.com/offensive-security/exploitdb/tree/master/shellcodes.

Oracle. 2020. "Oracle VM VirtualBox". Visited on November 6, 2020. https://www.virtualbox.org/.

Richert, Willi, and Luis Pedro Coelho. 2013. *Building Machine Learning Systems with Python.* Birmingham: Packt Publishing.

Shaw, Zed. 2020. "Programming, Motherfucker - Do you speak it?" Visited on March 26, 2020. http://programming-motherfucker.com/.

"Shell-Storm Shellcodes Database". 2020. Visited on November 6, 2020. http://shell-storm.org/shellcode/.

Sikorski, Michael, and Andrew Honig. 2012. *Practical Malware Analysis: A Hands-On Guide to Dissecting Malicious Software.* San Francisco, UNITED STATES: No Starch Press, Incorporated.

Sweetman, Dominic. 2007. *See MIPS Run.* Volume 2nd ed. The Morgan Kaufmann Series in Computer Architecture and Design. San Francisco, Calif: Morgan Kaufmann.

”The Official Radare2 Book”. 2020. Visited on December 14, 2020. https://book.rada.re/.

Theodoridis, Sergios. 2015. *Machine Learning : A Bayesian and Optimization Perspective.* London: Academic Press.

University of Jyväskylä. 2010a. ”Doing Research — Jyväskylän yliopiston Koppa”. Visited on December 7, 2020. https://koppa.jyu.fi/avoimet/hum/menetelmapolkuja/en/research-process/doing-research#reliability.

———. 2010b. ”Empirical Research — Jyväskylän yliopiston Koppa”. Visited on May 11, 2020. https://koppa.jyu.fi/avoimet/hum/menetelmapolkuja/en/methodmap/strategies/empirical-research.

———. 2010c. ”Experimental Research — Jyväskylän yliopiston Koppa”. Visited on May 11, 2020. https://koppa.jyu.fi/avoimet/hum/menetelmapolkuja/en/methodmap/strategies/experimental-research.

———. 2010d. ”Quantitative Research — Jyväskylän yliopiston Koppa”. Visited on May 11, 2020. https://koppa.jyu.fi/avoimet/hum/menetelmapolkuja/en/methodmap/strategies/quantitative-research.

———. 2010e. ”Strategies — Jyväskylän yliopiston Koppa”. Visited on May 11, 2020. https://koppa.jyu.fi/avoimet/hum/menetelmapolkuja/en/methodmap/strategies.

Zolotukhin, Mikhail, and Timo Hämäläinen. 2013. ”Support vector machine integrated with game-theoretic approach and genetic algorithm for the detection and classification of malware”, (Atlanta, GA) (): 211–216. Visited on April 17, 2020. http://ieeexplore.ieee.org/document/6824988/.

Appendices

A Python scripts used in bad character analysis of MSFvenom

Script for generating shellcodes with different 1-byte combinations as bad characters:

```
#! /usr/bin/python

import os
import sys

"""
This program attempts to create certain MSFvenom shellcodes with
every possible 1-byte combination.
"""

# The badchars table contains every 1-byte combination from 0x00 to 0xff.
# TODO: maybe generate these instead of hard-coding
badchars =
["\\x00", "\\x01", "\\x02", "\\x03", "\\x04", "\\x05", "\\x06", "\\x07", "\\x08", "\\x09", "\\x0a", "\\x0b", "\\x0c",
"\\x0d", "\\x0e", "\\x0f", "\\x10", "\\x11", "\\x12", "\\x13", "\\x14", "\\x15", "\\x16", "\\x17", "\\x18", "\\x19",
"\\x1a", "\\x1b", "\\x1c", "\\x1d", "\\x1e", "\\x1f", "\\x20", "\\x21", "\\x22", "\\x23", "\\x24", "\\x25", "\\x26",
"\\x27", "\\x28", "\\x29", "\\x2a", "\\x2b", "\\x2c", "\\x2d", "\\x2e", "\\x2f", "\\x30", "\\x31", "\\x32", "\\x33",
"\\x34", "\\x35", "\\x36", "\\x37", "\\x38", "\\x39", "\\x3a", "\\x3b", "\\x3c", "\\x3d", "\\x3e", "\\x3f", "\\x40",
"\\x41", "\\x42", "\\x43", "\\x44", "\\x45", "\\x46", "\\x47", "\\x48", "\\x49", "\\x4a", "\\x4b", "\\x4c", "\\x4d",
"\\x4e", "\\x4f", "\\x50", "\\x51", "\\x52", "\\x53", "\\x54", "\\x55", "\\x56", "\\x57", "\\x58", "\\x59", "\\x5a",
"\\x5b", "\\x5c", "\\x5d", "\\x5e", "\\x5f", "\\x60", "\\x61", "\\x62", "\\x63", "\\x64", "\\x65", "\\x66", "\\x67",
"\\x68", "\\x69", "\\x6a", "\\x6b", "\\x6c", "\\x6d", "\\x6e", "\\x6f", "\\x70", "\\x71", "\\x72", "\\x73", "\\x74",
"\\x75", "\\x76", "\\x77", "\\x78", "\\x79", "\\x7a", "\\x7b", "\\x7c", "\\x7d", "\\x7e", "\\x7f", "\\x80", "\\x81",
"\\x82", "\\x83", "\\x84", "\\x85", "\\x86", "\\x87", "\\x88", "\\x89", "\\x8a", "\\x8b", "\\x8c", "\\x8d", "\\x8e",
"\\x8f", "\\x90", "\\x91", "\\x92", "\\x93", "\\x94", "\\x95", "\\x96", "\\x97", "\\x98", "\\x99", "\\x9a", "\\x9b",
"\\x9c", "\\x9d", "\\x9e", "\\x9f", "\\xa0", "\\xa1", "\\xa2", "\\xa3", "\\xa4", "\\xa5", "\\xa6", "\\xa7", "\\xa8",
"\\xa9", "\\xaa", "\\xab", "\\xac", "\\xad", "\\xae", "\\xaf", "\\xb0", "\\xb1", "\\xb2", "\\xb3", "\\xb4", "\\xb5",
"\\xb6", "\\xb7", "\\xb8", "\\xb9", "\\xba", "\\xbb", "\\xbc", "\\xbd", "\\xbe", "\\xbf", "\\xc0", "\\xc1", "\\xc2",
"\\xc3", "\\xc4", "\\xc5", "\\xc6", "\\xc7", "\\xc8", "\\xc9", "\\xca", "\\xcb", "\\xcc", "\\xcd", "\\xce", "\\xcf",
"\\xd0", "\\xd1", "\\xd2", "\\xd3", "\\xd4", "\\xd5", "\\xd6", "\\xd7", "\\xd8", "\\xd9", "\\xda", "\\xdb", "\\xdc",
"\\xdd", "\\xde", "\\xdf", "\\xe0", "\\xe1", "\\xe2", "\\xe3", "\\xe4", "\\xe5", "\\xe6", "\\xe7", "\\xe8", "\\xe9",
"\\xea", "\\xeb", "\\xec", "\\xed", "\\xee", "\\xef", "\\xf0", "\\xf1", "\\xf2", "\\xf3", "\\xf4", "\\xf5", "\\xf6",
"\\xf7", "\\xf8", "\\xf9", "\\xfa", "\\xfb", "\\xfc", "\\xfd", "\\xfe", "\\xff"]

# Check that the amount of byte combinations is 2^8.
print("Number of 1-byte combinations", len(badchars))

# Supported architectures and used payloads.
pl_x86 = "linux/x86/chmod"
pl_x64 = "osx/x64/say"
pl_mips = "linux/mipsle/reboot"
pl_arm = "osx/armle/vibrate"
pl_ppc = "osx/ppc/shell/find_tag"
pl_ppc64 = "linux/ppc64/shell_find_port"
pl_sparc = "solaris/sparc/shell_find_port"
pl_aarch64 = "linux/aarch64/shell_reverse_tcp"

pl = ""

# The user can enter the desired architecture.
# User input will determine the payload.
# The program will shut down, if an unsupported architecture is supplied.
print("Available architectures: x86, x64, mips, arm, ppc, ppc64, sparc, aarch64")
val = input("Enter architecture: ")
```

```
val = val.lower()
print("Chosen architecture: " + val)

if val == "mips": pl = pl_mips
elif val == "sparc": pl = pl_sparc
elif val == "arm": pl = pl_arm
elif val == "ppc": pl = pl_ppc
elif val == "ppc64": pl = pl_ppc64
elif val == "aarch64": pl = pl_aarch64
elif val == "x86": pl = pl_x86
elif val == "x64": pl = pl_x64
else: sys.exit("Noooooooo! Try again...")

# This loop attempts to give every 1-byte combination
# as a bad character to MSFvenom using the -b parameter.
# The output file name is tagged with the used byte combination.
for i in badchars:
    filename = (i + "_" + pl).strip().replace("/", "_").replace("\\", "") + ".c"
    print( "msfvenom -p" + " " + pl + " " + "-f c" + " " +
    "-b" + " " + "'" + i + "'" + " " + "-o" + " " + filename )
    os.system( "msfvenom -p" + " " + pl + " " + "-f c" + " " +
    "-b" + " " + "'" + i + "'" + " " + "-o" + " " + filename )
```

Script for checking which bytes were accepted and which were rejected:

```
#! /usr/bin/python

import os
from os import listdir
from os.path import isfile, join

"""
This program checks which byte combinations were accepted and which were rejected.
Before running, the shellcodes must be manually moved to a folder which is named
after the target architecture, and the folder name must be written in upper case,
for example: X86, MIPS, PPC64 etc.
TODO: Make this process automatic.
"""

# Same table as in the previous script, removed to save space.
badchars = []

# The user can enter the desired architecture.
print("Available architectures: x86, x64, mips, arm, ppc, ppc64, sparc, aarch64")
val = input("Enter architecture: ")
val = val.upper()
print("Chosen architecture: " + val)

# Create a table of generated shellcode files based on user input.
path = os.path.abspath(os.getcwd()) + "/" + val + "/"
files = [f for f in listdir(path) if isfile(join(path,f))]

# Slice the used byte from the shellcode filename
# and create a new table which contains the accepted
# byte combinations.
files_bc = []
for i in files: files_bc.append("\\" + i[:3])

# Print the total amount of 1-byte combinations, the amount
# of accepted bytes and the amount of rejected bytes.
print("Total: " + str(len(badchars)) + "\n" + "Generated: " + str(len(files_bc)))
print("Rejected:", len(badchars) - len(files_bc))

# Create sets from the tables, compare them and print the
# rejected byte combinations.
set1 = set(badchars)
set2 = set(files_bc)
missing = list(sorted(set1-set2))
print("Rejected bytes:", ' '.join(missing))
```

Script for creating shellcodes for MSFvenom LHOST analysis

```
#! /usr/bin/python

import os
import sys

"""
Program for testing MSFvenom LHOST parameter with
every possible 1-byte combination.
"""

# The badchars table contains every 1-byte combination from 0x00 to 0xff.
badchars = []

# Supported architectures and payloads.
pl_x86 = "linux/x86/shell/reverse_tcp"
pl_x64 = "linux/x64/shell/reverse_tcp"
pl_mips = "linux/mipsbe/shell/reverse_tcp"
pl_arm = "osx/armle/shell_reverse_tcp"
pl_ppc = "linux/ppc/shell_reverse_tcp"
pl_ppc64 = "linux/ppc64/shell_reverse_tcp"
pl_sparc = "bsd/sparc/shell_reverse_tcp"
pl_aarch64 = "linux/aarch64/shell_reverse_tcp"

pl = "" # For deciding which payload to use

# The user can enter the desired architecture.
# User input will determine the payload.
# The program will shut down, if an unsupported architecture is supplied.
print("Available architectures: x86, x64, mips, arm, ppc, ppc64, sparc, aarch64")
val_arch = input("Enter architecture: ")
val_arch = val_arch.lower()
val_ip = input("Choose LHOST IP address: ")
print("Chosen architecture: " + val_arch)
print("Chosen IP address: " + str(val_ip))

if val_arch == "mips": pl = pl_mips
elif val_arch == "sparc": pl = pl_sparc
elif val_arch == "arm": pl = pl_arm
elif val_arch == "ppc": pl = pl_ppc
elif val_arch == "ppc64": pl = pl_ppc64
elif val_arch == "aarch64": pl = pl_aarch64
elif val_arch == "x86": pl = pl_x86
elif val_arch == "x64": pl = pl_x64
else: sys.exit("Noooooooo! Try again...")

# This loop attempts to give every 1-byte combination
# as a bad character to MSFvenom using the -b parameter.
# The output file name is tagged with the used byte combination.
for i in badchars:
    filename = (i + "_" + val_ip + "_" + pl).strip().replace("/", "_").replace("\\", "").replace(".", "_") + ".c"
    command = ( "msfvenom -p" + " " + pl + " " + "LHOST=" + val_ip + " " +
          "-f c" + " " + "-b" + " '" + i + "' " + "-o" + " " + filename )
    print(command)
    os.system(command)
```

Script for creating shellcodes for MSFvenom RHOST analysis

```
#! /usr/bin/python

import os
import sys

"""
Program for testing MSFvenom RHOST parameter with
every possible 1-byte combination.
"""

# The badchars table contains every 1-byte combination from 0x00 to 0xff.
badchars = []

# Supported architectures and payloads.
pl_x86 = "bsd/x86/shell_bind_tcp"
pl_x64 = "bsd/x64/shell_bind_tcp"
pl_mips = "linux/mipsle/shell_bind_tcp"
pl_arm = "linux/armle/shell/bind_tcp "
pl_ppc = "osx/ppc/shell_bind_tcp"
pl_ppc64 = "linux/ppc64/shell_bind_tcp"
pl_sparc = "bsd/sparc/shell_bind_tcp"
pl_aarch64 = ""

pl = ""

# The user can enter the desired architecture.
# User input will determine the payload.
# The program will shut down, if an unsupported architecture is supplied.
print("Available architectures: x86, x64, mips, arm, ppc, ppc64, sparc, aarch64")
val_arch = input("Enter architecture: ")
val_arch = val_arch.lower()
val_ip = input("Choose RHOST IP address or leave empty to ignore RHOST parameter: ")
print("Chosen architecture: " + val_arch)
print("Chosen port: " + str(val_ip))

if val_arch == "mips": pl = pl_mips
elif val_arch == "sparc": pl = pl_sparc
elif val_arch == "arm": pl = pl_arm
elif val_arch == "ppc": pl = pl_ppc
elif val_arch == "ppc64": pl = pl_ppc64
elif val_arch == "aarch64": pl = pl_aarch64
elif val_arch == "x86": pl = pl_x86
elif val_arch == "x64": pl = pl_x64
else: sys.exit("Noooooooo! Try again...")

# This loop attempts to give every 1-byte combination
# as a bad character to MSFvenom using the -b parameter.
# The output file name is tagged with the used byte combination.
if len(val_ip) < 1:
    for i in badchars:
        filename = (i + "_" + val_ip + "_" + pl).strip().replace("/", "_").replace("\\", "").replace(".", "_") + ".c"
        command = ("msfvenom -p" + " " + pl + " " "LPORT=4444" + " " +
                   "-f c" + " " + "-b" + " '" + i + "' " + "-o" + " " + filename)
        print(command)
        os.system(command)
else:
    for i in badchars:
        filename = (i + "_" + val_ip + "_" + pl).strip().replace("/", "_").replace("\\", "").replace(".", "_") + ".c"
        command = ("msfvenom -p" + " " + pl + " " "LPORT=4444" + " " + "RHOST=" + val_ip + " " +
                   "-f c" + " " + "-b" + " '" + i + "' " + "-o" + " " + filename)
        print(command)
        os.system(command)
```

Script for creating shellcodes for MSFvenom LPORT analysis

```
#! /usr/bin/python

import os
import sys

"""
Program for testing MSFvenom LPORT parameter with
every possible 1-byte combination.
"""

# The badchars table contains every 1-byte combination from 0x00 to 0xff.
badchars = []

# Supported architectures and payloads.
pl_x86 = "bsd/x86/shell_bind_tcp"       #LPORT #RHOST
pl_x64 = "bsd/x64/shell_bind_tcp"       #LPORT #RHOST
pl_mips = "linux/mipsle/shell_bind_tcp" #LPORT #RHOST
pl_arm = "linux/armle/shell/bind_tcp "  #LPORT #RHOST
pl_ppc = "osx/ppc/shell_bind_tcp"       #LPORT #RHOST
pl_ppc64 = "linux/ppc64/shell_bind_tcp" #LPORT #RHOST
pl_sparc = "bsd/sparc/shell_bind_tcp"   #LPORT #RHOST

pl = ""

# The user can enter the desired architecture.
# User input will determine the payload.
# The program will shut down, if an unsupported architecture is supplied.
print("Available architectures: x86, x64, mips, arm, ppc, ppc64, sparc, aarch64")
val_arch = input("Enter architecture: ")
val_arch = val_arch.lower()
val_port = input("Choose LPORT: ")
print("Chosen architecture: " + val_arch)
print("Chosen LPORT: " + str(val_port))

if val_arch == "mips": pl = pl_mips
elif val_arch == "sparc": pl = pl_sparc
elif val_arch == "arm": pl = pl_arm
elif val_arch == "ppc": pl = pl_ppc
elif val_arch == "ppc64": pl = pl_ppc64
elif val_arch == "aarch64": pl = pl_aarch64
elif val_arch == "x86": pl = pl_x86
elif val_arch == "x64": pl = pl_x64
else: sys.exit("Noooooooo! Try again...")

# This loop attempts to give every 1-byte combination
# as a bad character to MSFvenom using the -b parameter.
# The output file name is tagged with the used byte combination.
for i in badchars:
    filename = (i + "_" + val_port + "_" + pl).strip().replace("/", "_").replace("\\", "").replace(".", "_") + ".c"
    command = ( "msfvenom -p" + " " + pl + " " + "LPORT=" + val_port + " " +
                "-f c" + " " + "-b" + " '" + i + "' " + "-o" + " " + filename )
    print(command)
    os.system(command)
```

Script for checking the results of LHOST, RHOST and LPORT tests

```
#! /usr/bin/python

import os
from os import listdir
from os.path import isfile, join

badchars = []

# The user can enter the desired architecture.
print("Available architectures: x86, x64, mips, arm, ppc, ppc64, sparc, aarch64")
val_arch = input("Enter architecture: ")
val_arch = val_arch.upper()
val_choice = input("Check LHOST IP(1), RHOST IP(2) or LPORT(3)? ")
val_choice = int(val_choice)
if val_choice == 1:
    print("Checking LHOST")
    val_lhost = input("Enter IP: ")
    print("Chosen architecture: " + val_arch)
    print("Chosen LHOST: " + val_lhost)
    val_lhost = "lhost_" + val_lhost.replace(".", "_")

    # Create a table of generated shellcode files based on user input.
    path = os.path.abspath(os.getcwd()) + "/" + val_lhost + "/" + val_arch + "/"
    files = [f for f in listdir(path) if isfile(join(path,f))]

    # Slice the used byte from the shellcode filename
    # and create a new table which contains the accepted
    # byte combinations.
    files_bc = []
    for i in files: files_bc.append("\\" + i[:3])

    # Print the total amount of 1-byte combinations, the amount
    # of accepted bytes and the amount of rejected bytes.
    print("Total: " + str(len(badchars)) + "\n" + "Generated: " + str(len(files_bc)))
    print("Rejected:", len(badchars) - len(files_bc))

    # Create sets from the tables, compare them and print the
    # rejected byte combinations.
    set1 = set(badchars)
    set2 = set(files_bc)
    missing = list(sorted(set1-set2))
    print("Rejected bytes:", ' '.join(missing))

if val_choice == 2:
    print("Checking RHOST")
    val_rhost = input("Enter IP: ")
    print("Chosen architecture: " + val_arch)
    print("Chosen RHOST: " + val_rhost)
    val_rhost = "rhost_" + val_rhost.replace(".", "_")

    # Create a table of generated shellcode files based on user input.
    path = os.path.abspath(os.getcwd()) + "/" + val_rhost + "/" + val_arch + "/"
    files = [f for f in listdir(path) if isfile(join(path,f))]

    # Slice the used byte from the shellcode filename
    # and create a new table which contains the accepted
    # byte combinations.
    files_bc = []
    for i in files: files_bc.append("\\" + i[:3])

    # Print the total amount of 1-byte combinations, the amount
    # of accepted bytes and the amount of rejected bytes.
```

```
    print("Total: " + str(len(badchars)) + "\n" + "Generated: " + str(len(files_bc)))
    print("Rejected:", len(badchars) - len(files_bc))

    # Create sets from the tables, compare them and print the
    # rejected byte combinations.
    set1 = set(badchars)
    set2 = set(files_bc)
    missing = list(sorted(set1-set2))
    print("Rejected bytes:", ' '.join(missing))

if val_choice == 3:
    print("Checking LPORT")
    val_lport = input("Enter port: ")
    print("Chosen architecture: " + val_arch)
    print("Chosen LPORT: " + val_lport)
    val_lport = "lport_" + val_lport

    # Create a table of generated shellcode files based on user input.
    path = os.path.abspath(os.getcwd()) + "/" + val_lport + "/" + val_arch + "/"
    files = [f for f in listdir(path) if isfile(join(path,f))]

    # Slice the used byte from the shellcode filename
    # and create a new table which contains the accepted
    # byte combinations.
    files_bc = []
    for i in files: files_bc.append("\\" + i[:3])

    # Print the total amount of 1-byte combinations, the amount
    # of accepted bytes and the amount of rejected bytes.
    print("Total: " + str(len(badchars)) + "\n" + "Generated: " + str(len(files_bc)))
    print("Rejected:", len(badchars) - len(files_bc))

    # Create sets from the tables, compare them and print the
    # rejected byte combinations.
    set1 = set(badchars)
    set2 = set(files_bc)
    missing = list(sorted(set1-set2))
    print("Rejected bytes:", ' '.join(missing))
```

B Python script for generating shellcodes with MSFvenom

```
#! /usr/bin/python

import os

# Specify bad characters
# 'badchars' is a string which contains bad characters
badchars = [ "No badchars", "\\x00", "\\x0a", "\\x0d", "\\xff", "\\x00\\x0a",
             "\\x00\\x0d", "\\x00\\xff", "\\x0a\\x0d", "\\x0a\\xff", "\\x0d\\xff", "Custom" ]

def check(sc_name):
    """ This function checks whether the shellcode already exists or not.

    Args:
        sc_name: Filename of the shellcode that is under inspection.

    Returns:
        bool: True if the file exists, False if it does not exist.

    """
    exists = os.path.isfile('./' + sc_name)
    return exists

def create_shellcode(sc_file):
    """ This function creates shellcodes.
    Format can be changed by changing the value of code_format.
    User can choose whether or not to use bad chars.

    This function also inspects whether or not the shellcode already exists
    and skips the creation if it does. The inspection is done using another
    function called check.

    Further information about the used msfvenom parameters can be seen at:
    https://github.com/rapid7/metasploit-framework/wiki/How-to-use-msfvenom

    Example:
        One iteration of the for loop is:
        msfvenom -p <shellcode> -b <bad characters> -f c -o <filename>

    Args:
        sc_name: a text file which includes the names of all the shellcodes.

    Attributes:
        filename: filename for each singular shellcode.
        exists: boolean for checking if the shellcode already exists.
        badchars: bad characters to be avoided.

    """

    for i in sc_file:
        filename = (bc + "_" + i).strip().replace("/", "_").replace("\\", "") + "." + code_format
        exists = check(filename)
        if exists == True:
            print(filename + " already exists")
            continue
        elif bc == "no_bc":
            command = ( "msfvenom -p" + " " + i.strip() + " " + "-f " + code_format + " " +
                        "-o" + " " + filename )
            print(command)
            os.system(command)
        else:
            command = ( "msfvenom -p" + " " + i.strip() + " " + "-f " + code_format + " " +
                        "-b" + " " + "'" + bc + "'" + " " + "-o" + " " + filename )
```

```
            print(command)
            os.system(command)

for (i, item) in enumerate(badchars, start=1): print(i, item)

code_format = "elf"
val_bc = input("Choose bad character: ")
val_bc = int(val_bc)
bc = badchars[val_bc-1]
print(bc + "\n")
if bc == "No badchars": bc = "no_bc"
if bc == "Custom":
    val_custom_bc = input("Enter bad character: ")
    bc = val_custom_bc

file_sc = open('files/msfv_pl.txt', 'r')
create_shellcode(file_sc)
file_sc.close()
```

C Rejected bad bytes in each MSFvenom test

Rejected one byte combinations

x86

No rejections

x64

0x48

mips

0x01, 0x02, 0x04, 0x05, 0x0c, 0x21, 0x24, 0x28

arm

0x03, 0x04, 0x08, 0x1f, 0x20, 0x31, 0x44, 0x74, 0x9e, 0xa0, 0xe1, 0xe5, 0xea, 0xf0, 0xf9

arm64

0x00, 0x01, 0x02, 0x03, 0x07, 0x08, 0x0a, 0x0b, 0x0f, 0x10, 0x11, 0x180x19, 0x1b, 0x21, 0x2f, 0x35, 0x40, 0x41, 0x5c, 0x60, 0x62, 0x68, 0x69, 0x6e, 0x73, 0x80, 0x91, 0xa8, 0xaa, 0xc8, 0xd2, 0xd4, 0xe0, 0xe1, 0xe2, 0xe3, 0xf9

ppc

0x01, 0x02, 0x04, 0x05, 0x2a, 0x2c, 0x38, 0x3b, 0x3c, 0x40, 0x79, 0x7c0x7f, 0x82, 0xa5, 0xa6, 0xfd, 0xff

ppc64

0x01, 0x02, 0x03, 0x04, 0x08, 0x09, 0x0b, 0x0c, 0x11, 0x18, 0x190x1d, 0x21, 0x24, 0x25, 0x28, 0x2a, 0x2c, 0x2f, 0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x40, 0x41, 0x440x4f, 0x61, 0x62, 0x67, 0x68, 0x69, 0x6e, 0x73, 0x78, 0x79, 0x7c, 0x7d, 0x7e, 0x7f, 0x80, 0x81, 0x820x97, 0x98, 0xa0, 0xa1, 0xa3, 0xa5, 0xa6, 0xbe, 0xc3, 0xc8, 0xd0, 0xde, 0xe1, 0xe3, 0xec, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff

sparc

0x03, 0x04, 0x12, 0x14, 0x15, 0x20, 0x23, 0x40, 0x9d, 0xa2, 0xbf, 0xe0, 0xea, 0xfb, 0xf

LHOST tests

x86, LHOST=192.168.1.1
0xc9

x86, LHOST=10.0.0.1
0xc9

x64, LHOST=192.168.1.1
0x48, 0xff

x64, LHOST=10.0.0.1
0x48, 0xff

mips, LHOST=192.168.1.1
0x01, 0x02, 0x03, 0x04, 0x05, 0x08, 0x0b, 0x0c, 0x10, 0x20, 0x21, 0x24, 0x27, 0x28, 0x33, 0x40, 0x4a, 0x58, 0x60, 0x80, 0xc0, 0xea, 0xef, 0xfc, 0xff

mips, LHOST=10.0.0.1
0x01, 0x02, 0x03, 0x04, 0x05, 0x08, 0x0b, 0x0c, 0x10, 0x20, 0x21, 0x24, 0x27, 0x28, 0x33, 0x40, 0x4a, 0x58, 0x60, 0x80, 0x0a, 0xea, 0xef, 0xfc, 0xff

arm, LHOST=192.168.1.1
0x00, 0x01, 0x02, 0x04, 0x05, 0x06, 0x0a, 0x0d, 0x0e, 0x10, 0x11, 0x14, 0x20, 0x2f, 0x3b, 0x45, 0x4d, 0x50, 0x55, 0x5a, 0x5c, 0x60, 0x61, 0x62, 0x68, 0x69, 0x6e, 0x73, 0x7e, 0x80, 0x86, 0x8f, 0xa0, 0xa8, 0xaa, 0xc0, 0xd0, 0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5, 0xeb, 0xef, 0xf8, 0xff

arm, LHOST=10.0.0.1
0x00, 0x01, 0x02, 0x04, 0x05, 0x06, 0x0a, 0x0d, 0x0e, 0x10, 0x11, 0x14, 0x20, 0x2f, 0x3b, 0x45, 0x4d, 0x50, 0x55, 0x5a, 0x5c, 0x60, 0x61, 0x62, 0x68, 0x69, 0x6e, 0x73, 0x7e, 0x80, 0x86, 0x8f, 0xa0, 0xaa, 0xc0, 0xd0, 0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5, 0xeb, 0xef, 0xf8, 0xff

arm64, LHOST=192.168.1.1
0x00, 0x01, 0x02, 0x03, 0x07, 0x08, 0x0b, 0x10, 0x11, 0x18, 0x19, 0x1b, 0x21, 0x2f, 0x35,

0x40, 0x41, 0x5c, 0x60, 0x62, 0x68, 0x69, 0x6e, 0x73, 0x80, 0x91, 0xa8, 0xaa, 0xc0, 0xc8, 0xd2, 0xd4, 0xe0, 0xe1, 0xe2, 0xe3, 0xf9

arm64, LHOST=10.0.0.1
0x00, 0x01, 0x02, 0x03, 0x07, 0x08, 0x0a, 0x0b, 0x10, 0x11, 0x18, 0x19, 0x1b, 0x21, 0x2f, 0x35, 0x40, 0x41, 0x5c, 0x60, 0x62, 0x68, 0x69, 0x6e, 0x73, 0x80, 0x91, 0xa8, 0xaa, 0xc8, 0xd2, 0xd4, 0xe0, 0xe1, 0xe2, 0xe3, 0xf9

ppc, LHOST=192.168.1.1
0x01, 0x02, 0x04, 0x2a, 0x2c, 0x38, 0x3b, 0x40, 0x79, 0x7c, 0x7f, 0x82, 0xa5, 0xa6, 0xc0, 0xe0, 0xfd, 0xff

ppc, LHOST=10.0.0.1
0x01, 0x02, 0x04, 0x2a, 0x2c, 0x38, 0x3b, 0x40, 0x79, 0x7c, 0x7f, 0x82, 0xa5, 0xa6, 0xc0, 0xe0, 0xfd, 0xff

ppc64, LHOST=192.168.1.1
0x01, 0x02, 0x03, 0x04, 0x0b, 0x0c, 0x11, 0x1b, 0x1d, 0x21, 0x24, 0x25, 0x2a, 0x2c, 0x2f, 0x35, 0x37, 0x38, 0x3a, 0x3b, 0x3d, 0x3e, 0x40, 0x41, 0x43, 0x44, 0x5c, 0x61, 0x62, 0x64, 0x67, 0x68, 0x69, 0x6e, 0x73, 0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f, 0x80, 0x81, 0x82, 0x93, 0x96, 0x98, 0x9d, 0xa0, 0xa1, 0xa5, 0xa6, 0xa8, 0xbe, 0xc0, 0xc1, 0xc8, 0xd3, 0xdb, 0xde, 0xe0, 0xe1, 0xec, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff

ppc64, LHOST=10.0.0.1
0x00, 0x01, 0x02, 0x03, 0x04, 0x0a, 0x0b, 0x0c, 0x11, 0x1b, 0x1d, 0x21, 0x24, 0x25, 0x2a, 0x2c, 0x2f, 0x35, 0x37, 0x38, 0x3a, 0x3b, 0x3d, 0x3e, 0x40, 0x41, 0x43, 0x44, 0x5c, 0x61, 0x62, 0x64, 0x67, 0x68, 0x69, 0x6e, 0x73, 0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f, 0x80, 0x81, 0x82, 0x93, 0x96, 0x98, 0x9d, 0xa0, 0xa1, 0xa5, 0xa6, 0xbe, 0xc0, 0xc1, 0xc8, 0xd3, 0xdb, 0xde, 0xe0, 0xe1, 0xec, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff

sparc, LHOST=192.168.1.1
0x03, 0x12, 0x14, 0x20, 0x23, 0xa2, 0xbf, 0xe0, 0xff

sparc, LHOST=10.0.0.1
0x03, 0x12, 0x14, 0x20, 0x23, 0xa2, 0xbf, 0xe0, 0xff

RHOST tests

x86, no RHOST

No rejections

x86, RHOST=124.173.232.109

No rejections

x64, no RHOST

0x48, 0xff

x64, RHOST=124.173.232.109

0x48, 0xff

mips, no RHOST

0x01, 0x02, 0x04, 0x05, 0x0c, 0x10, 0x20, 0x21, 0x23, 0x24, 0x27, 0x28, 0x2f, 0x31, 0xc0, 0xef, 0xf0, 0xfc, 0xff

mips, RHOST=124.173.232.109

0x01, 0x02, 0x04, 0x05, 0x0c, 0x10, 0x20, 0x21, 0x23, 0x24, 0x27, 0x28, 0x2f, 0x31, 0xc0, 0xef, 0xf0, 0xfc, 0xff

arm, no RHOST

0x00, 0x01, 0x02, 0x03, 0x04, 0x06, 0x07, 0x0c, 0x0d, 0x10, 0x11, 0x19, 0x20, 0x22, 0x26, 0x2f, 0x30, 0x40, 0x41, 0x42, 0x4d, 0x50, 0x52, 0x54, 0x5c, 0x63, 0x70, 0x81, 0x82, 0x87, 0x8d, 0x8f, 0x9d, 0x9f, 0xa0, 0xb0, 0xc0, 0xd0, 0xd4, 0xda, 0xe0, 0xe1, 0xe2, 0xe3, 0xe5, 0xea, 0xef, 0xf0, 0xf7, 0xfa, 0xff

arm, RHOST=124.173.232.109

0x00, 0x01, 0x02, 0x03, 0x04, 0x06, 0x07, 0x0c, 0x0d, 0x10, 0x11, 0x19, 0x20, 0x22, 0x26, 0x2f, 0x30, 0x40, 0x41, 0x42, 0x4d, 0x50, 0x52, 0x54, 0x5c, 0x63, 0x70, 0x81, 0x82, 0x87, 0x8d, 0x8f, 0x9d, 0x9f, 0xa0, 0xb0, 0xc0, 0xd0, 0xd4, 0xda, 0xe0, 0xe1, 0xe2, 0xe3, 0xe5, 0xea, 0xef, 0xf0, 0xf7, 0xfa, 0xff

ppc, no RHOST

0x01, 0x02, 0x04, 0x05, 0x2a, 0x2c, 0x38, 0x3b, 0x40, 0x60, 0x79, 0x7c, 0x7f, 0x82, 0x90,

0xa5, 0xa6, 0xac, 0xe0, 0xe8, 0xf8, 0xfd, 0xff

ppc, RHOST=124.173.232.109
0x01, 0x02, 0x04, 0x05, 0x2a, 0x2c, 0x38, 0x3b, 0x40, 0x60, 0x79, 0x7c, 0x7f, 0x82, 0x90, 0xa5, 0xa6, 0xac, 0xe0, 0xe8, 0xf8, 0xfd, 0xff

ppc64, no RHOST
0x01, 0x02, 0x03, 0x05, 0x06, 0x0b, 0x0c, 0x11, 0x1b, 0x1d, 0x21, 0x24, 0x25, 0x2a, 0x2c, 0x2f, 0x36, 0x37, 0x38, 0x3b, 0x3d, 0x3e, 0x40, 0x41, 0x44, 0x5c, 0x61, 0x62, 0x64, 0x67, 0x68, 0x69, 0x6e, 0x73, 0x75, 0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f, 0x80, 0x81, 0x82, 0x96, 0x97, 0x98, 0x9d, 0xa0, 0xa1, 0xa3, 0xa5, 0xa6, 0xab, 0xbe, 0xc1, 0xc8, 0xdb, 0xde, 0xe0, 0xe1, 0xec, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff

ppc64, RHOST=124.173.232.109
0x01, 0x02, 0x03, 0x05, 0x06, 0x0b, 0x0c, 0x11, 0x1b, 0x1d, 0x21, 0x24, 0x25, 0x2a, 0x2c, 0x2f, 0x36, 0x37, 0x38, 0x3b, 0x3d, 0x3e, 0x40, 0x41, 0x44, 0x5c, 0x61, 0x62, 0x64, 0x67, 0x68, 0x69, 0x6e, 0x73, 0x75, 0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f, 0x80, 0x81, 0x82, 0x96, 0x97, 0x98, 0x9d, 0xa0, 0xa1, 0xa3, 0xa5, 0xa6, 0xab, 0xbe, 0xc1, 0xc8, 0xdb, 0xde, 0xe0, 0xe1, 0xec, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff

sparc, no RHOST
0x03, 0x12, 0x14, 0x20, 0x23, 0x40, 0xa2, 0xbf, 0xe0, 0xff

sparc, RHOST=124.173.232.109
0x03, 0x12, 0x14, 0x20, 0x23, 0x40, 0xa2, 0xbf, 0xe0, 0xff

LPORT tests

x86, LPORT=1234

No rejections

x86, LPORT=10

No rejections

x64, LPORT=1234

0x48, 0xff

x64, LPORT=10

0x48, 0xff

mips, LPORT=1234

0x01, 0x02, 0x04, 0x05, 0x0c, 0x10, 0x20, 0x21, 0x23, 0x24, 0x27, 0x28, 0x2f, 0x31, 0xc0, 0xef, 0xf0, 0xfc, 0xff

mips, LPORT=10

0x01, 0x02, 0x04, 0x05, 0x0a, 0x0c, 0x10, 0x20, 0x21, 0x23, 0x24, 0x27, 0x28, 0x2f, 0x31, 0xc0, 0xef, 0xf0, 0xfc, 0xff

arm, LPORT=1234

0x00, 0x01, 0x02, 0x03, 0x04, 0x06, 0x07, 0x0c, 0x0d, 0x10, 0x19, 0x20, 0x22, 0x26, 0x2f, 0x30, 0x40, 0x41, 0x42, 0x4d, 0x50, 0x52, 0x54, 0x63, 0x70, 0x81, 0x82, 0x87, 0x8d, 0x8f, 0x9d, 0x9f, 0xa0, 0xb0, 0xc0, 0xd0, 0xd2, 0xd4, 0xda, 0xe0, 0xe1, 0xe2, 0xe3, 0xe5, 0xea, 0xef, 0xf0, 0xf7, 0xfa, 0xff

arm, LPORT=10

0x00, 0x01, 0x02, 0x03, 0x04, 0x06, 0x07, 0x0a, 0x0c, 0x0d, 0x10, 0x19, 0x20, 0x22, 0x26, 0x2f, 0x30, 0x40, 0x41, 0x42, 0x4d, 0x50, 0x52, 0x54, 0x63, 0x70, 0x81, 0x82, 0x87, 0x8d, 0x8f, 0x9d, 0x9f, 0xa0, 0xb0, 0xc0, 0xd0, 0xd4, 0xda, 0xe0, 0xe1, 0xe2, 0xe3, 0xe5, 0xea, 0xef, 0xf0, 0xf7, 0xfa, 0xff

ppc, LPORT=1234

0x01, 0x02, 0x04, 0x05, 0x2a, 0x2c, 0x38, 0x3b, 0x40, 0x60, 0x79, 0x7c, 0x7f, 0x82, 0x90,

0xa5, 0xa6, 0xac, 0xe0, 0xe8, 0xf8, 0xfd, 0xff

ppc, LPORT=10

0x01, 0x02, 0x04, 0x05, 0x2a, 0x2c, 0x38, 0x3b, 0x40, 0x60, 0x79, 0x7c, 0x7f, 0x82, 0x90, 0xa5, 0xa6, 0xac, 0xe0, 0xe8, 0xf8, 0xfd, 0xff

ppc64, LPORT=1234

0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x0b, 0x0c, 0x11, 0x1b, 0x1d, 0x21, 0x24, 0x25, 0x2a, 0x2c, 0x2f, 0x36, 0x37, 0x38, 0x3b, 0x3d, 0x3e, 0x40, 0x41, 0x44, 0x61, 0x62, 0x64, 0x67, 0x68, 0x69, 0x6e, 0x73, 0x75, 0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f, 0x80, 0x81, 0x82, 0x96, 0x97, 0x98, 0x9d, 0xa0, 0xa1, 0xa3, 0xa5, 0xa6, 0xab, 0xbe, 0xc1, 0xc8, 0xd2, 0xdb, 0xde, 0xe0, 0xe1, 0xec, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff

ppc64, LPORT=10

0x00, 0x01, 0x02, 0x03, 0x05, 0x06, 0x0a, 0x0b, 0x0c, 0x11, 0x1b, 0x1d, 0x21, 0x24, 0x25, 0x2a, 0x2c, 0x2f, 0x36, 0x37, 0x38, 0x3b, 0x3d, 0x3e, 0x40, 0x41, 0x44, 0x61, 0x62, 0x64, 0x67, 0x68, 0x69, 0x6e, 0x73, 0x75, 0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f, 0x80, 0x81, 0x82, 0x96, 0x97, 0x98, 0x9d, 0xa0, 0xa1, 0xa3, 0xa5, 0xa6, 0xab, 0xbe, 0xc1, 0xc8, 0xdb, 0xde, 0xe0, 0xe1, 0xec, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff

sparc, LPORT=1234

0x03, 0x12, 0x14, 0x20, 0x23, 0x40, 0xa2, 0xbf, 0xe0, 0xfd, 0xff

sparc, LPORT=10

0x03, 0x12, 0x14, 0x20, 0x23, 0x40, 0xa2, 0xbf, 0xe0, 0xff