Sami Kairajärvi

Automatic identification of architecture and endianness using binary file contents

in Information Technology Master's Thesis

May 14, 2019

University of Jyväskylä

Faculty of Information Technology

Author: Sami Kairajärvi

Contact information: `sami.m.kairajarvi@student.jyu.fi`

Supervisor: Andrei Costin

Title: Automatic identification of architecture and endianness using binary file contents

Työn nimi: Arkkitehtuurin ja tavujärjestyksen automaattinen tunnistaminen binääritiedoston sisällön avulla

Project: Master's Thesis

Study line: Cyber security

Page count: 71+5

Abstract: This thesis explores how architecture and endianness of executable code can be identified using binary file contents, as falsely identifying the architecture caused about 10% of failures of firmware analysis in a recent study (Costin et al. 2014). A literature review was performed to identify the current state-of-the-art methods and how they could be improved in terms of algorithms, performance, data sets, and support tools. The thorough review identified methods presented by Clemens (2015) and De Nicolao et al. (2018) as the state-of-the-art and found that they had good results. However, these methods were found lacking essential tools to acquire or build the data sets as well as requiring more comprehensive comparison of classifier performance on full binaries. An experimental evaluation was performed to test classifier performance on different situations. For example, when training and testing classifiers with only code sections from executable files, all the classifiers performed equally well achieving over 98% accuracy. On samples with very small code sections 3-nearest neighbors and SVM had the best performance achieving 90% accuracy at 128 bytes. At the same time, random forest classifier performed the best classifying full binaries when trained with code sections at 90% accuracy and 99.2% when trained using full binaries.

Keywords: Firmware Analysis, Supervised Machine Learning, Classification, Binary Code

Suomenkielinen tiivistelmä: Tässä tutkielmassa tutkitaan kuinka suoritettavan ohjelman arkkitehtuurin ja tavujärjestyksen voi tunnistaa binääritiedoston sisällön avulla. Tätä tietoa tarvittiin esimerkiksi siksi, että arkkitehtuurin väärä tunnistus aiheutti 10% laiteohjelmistoanalyysin epäonnistumisista viimeaikaisessa tutkimuksessa (Costin et al. 2014). Tutkielmassa tehtiin kirjallisuuskatsaus, jossa etsittiin tämänhetkiset parhaat metodit sekä kuinka niitä voisi parantaa. Clemens (2015) ja De Nicolao et al. (2018) tunnistettiin johtavina tutkimuksina, ja niissä esitetyt menetelmät antoivat hyviä luokittelutuloksia. Puutteita löytyi aineistonkeräyksestä sekä luokittelijoiden laajemmasta vertailusta kokonaisten binäärien luokittelussa. Tutkielman kokeellisessa osiossa luokittelijoiden suorituskykyä mitattiin erilaisissa tilanteissa. Luokittelijoita esimerkiksi opetettiin ja testattiin vain suoritettavien ohjelmien koodiosioilla, jolloin kaikki luokittelijat saavuttivat vähintään 98% tarkkuuden. 3-lähimmän naapurin menetelmä sekä SVM luokittelija suoriutuivat parhaiten pienten näytteiden luokittelussa saavuttaen 90% tarkkuuden jo 128 tavun kokoisten tiedostojen kohdalla. Satunnaismetsä luokittelijalla oli paras suorituskyky kokonaisten binääritiedostojen luokittelussa 90% tarkkuudella, kun luokittelija opetettiin binääritiedoston koodiosilla, ja 99.2%, kun luokittelija opetettiin kokonaisilla binääritiedostoilla.

Avainsanat: Laiteohjelmiston analysointi, valvottu koneoppiminen, luokittelu, binäärikoodi

List of Figures

List of Tables

Contents

1 Introduction

The IoT revolution is already under way. Samsung Co-CEO Boo-Keun Yoon declared in 2015 that 90% of Samsung products would be connected to the web by 2017, and 100% by the year 2020. (Hyppönen, Nyman, et al. 2017) The number of IoT devices in 2020 is estimated to be from 20 to 30 billion. (Gartner 2017; Nordrum 2016).

IoT is usually composed of IoT-enabled devices (the "things" in IoT), connectivity technologies such as WiFi, Bluetooth, Bluetooth Low Energy (BLE) and Zigbee, the cloud infrastructure that can store the data and offer analytics, dashboards, and sometimes also smartphone applications to control the IoT devices. Zhang et al. (2014) define ”Things” in ”Internet of Things” (IoT) ”a physical or virtual object which connects to the Internet and has the ability to communicate with human users or other objects”. IoT-enabled devices should be viewed as computers that can, in case a of a smart light-bulb, also produce light (Hyppönen, Nyman, et al. 2017).

IoT devices contain and execute software which is referred in the literature as firmware. Zaddach and Costin (2013) define firmware as all code running on the hardware’s processor. It can hold a lot of valuable data that an attacker can use to launch more effective attacks, for example cryptographic keys, vulnerabilities, backdoors, or Personally Identifiable Information (PII). These could be used to reset the device to default settings, install custom SSL certificates or reroute traffic to attacker’s servers (Bhattarai and Wang 2018). Statistics indicate that 92.6% of the vulnerabilities discovered in current industrial control systems are in software or firmware, whereas only 7.4% are associated with hardware (Zhu et al. 2017). Errors or vulnerabilities in these kinds of environments can have serious implications affecting both digital and physical worlds. (Shoshitaishvili et al. 2015; Costin and Francillon 2014) Due to the expected massive amount of IoT devices, manual analysis to find vulnerabilities is not feasible, so automated methods are needed.

Vulnerabilities can be automatically identified by analyzing the firmware with static and dynamic analysis, symbolic execution or domain-specific analysis. Firmware is usually distributed as an archive or file system image. Static analysis focuses on analyzing the source code or the application without executing it, while dynamic analysis tests the application by running it. Costin, Zarras, and Francillon (2016) used an existing system with the same architecture as the firmware to dynamically analyze firmware. One problem Costin et al. (2014) faced was that the CPU architecture required by the firmware’s code was improperly detected, which explained about 10% of failures of analysis. Therefore, a correct architecture identification can help emulate more firmware images and as a consequence can lead to quicker discovery of more vulnerabilities. Within this context, the present thesis will explore methods to automatically identify the target CPU architecture of binary files, as such binary files are present in various firmware updates intended for IoT and embedded devices.

A computing architecture, i.e., CPU and hardware, has many attributes which determines its properties. In the context of this thesis, the two important attributes of a computing architecture are the Instruction Set Architecture (ISA) and the endianness. ISA is defined by opcodes, which are instructions for the CPU encoded as a stream of bytes and they are unique to a given architecture. The endianness determines the byte order in the memory, within the executable files and on the CPU execution pipeline.

A few recent papers propose methods to automate the identification of the architecture of a given executable file using machine learning. Clemens (2015) used Byte Frequency Distribution (BFD) to classify different ISAs and created several discriminatory signatures for identifying endianness to differentiate architectures with same op codes but different endianness. The author tried different classifiers such as neural net, Support Vector Machine (SVM), naive Bayes, nearest neighbor and decision trees to see which has the best performance. Clemens' results showed that SVM and nearest neighbor had the highest accuracy. De Nicolao et al. (2018) continued on Clemens (2015) work in ELISA framework and added more machine learning features in addition to BFD and endianness signatures. The researchers utilized function epilog and prolog signatures from angr framework (Shoshitaishvili et al. 2016) to improve discriminating between similar architectures. De Nicolao et al. (2018) used multi-class logistic regression for classifying and achieved a better accuracy than Clemens (2015).

Given all this context, this thesis aims to answer the following questions:

RQ1: What are the current state of the art methods and tools related to identifying binary code architecture and endianness?

RQ2: How could existing methods and approaches be improved in terms of algorithms, performance, data sets, and support tools?

RQ3: What method for identifying architecture and endianness has the best performance overall and in which cases existing methods outperform the others?

This thesis expects to validate the results of the current state of the art methods and improve the existing methods in terms of algorithms, performance, data sets, and support tools. A literature review will be performed to determine the current state of the art methods for identifying architecture and endianness. This will be accomplished by exploring the top journals of the field and performing queries on online science literature databases, e.g., Google Scholar, Research Gate, Web of Science and JYX. A toolset will be implemented to support (re)creation of datasets of binary files for different CPU architectures. The identified and most promising classification methods will be implemented and trained with data obtained from the toolset. Classifier performance will be evaluated using metrics such as F1 score, accuracy, recall and Area Under the Curve (AUC). Cross-validation will be performed to estimate how well the methods would perform in practice, and to avoid overfitting.

This thesis is divided into 6 chapters: Chapter 2 discusses how the topic of this thesis is relevant in the context of security-oriented IoT and embedded firmware analysis. Chapter 3 gives an overview of machine learning and different classifiers used in this thesis. Chapter 4 introduces the methodology used to test the classifier performance in different situations. In Chapter 5 the results of the experiment are presented and compared to the previous research. Finally, the Chapters 6 and 7 go through the major findings of the thesis and propose possible future research ideas.

2 Overview of security-oriented firmware analysis

Zaddach and Costin (2013) define firmware as all code running on the hardware's processor. In recent years, research on security of IoT and embedded firmware and software has been considerably advanced and diversified. For example, Costin et al. (2014) analyzed the firmware of 32 thousand firmware packages and discovered 38 previously unknown vulnerabilities in over 693 firmware images. To efficiently find vulnerabilities within such large population of firmware packages and devices, automated methods are needed as manually going through every firmware image is not feasible. An automated end-to-end firmware analysis for vulnerabilities consists of collecting firmware images using web crawlers, unpacking the firmware, classifying it, and then using static, dynamic and hybrid analyzes to discover vulnerabilities within the firmware.

This chapter goes through the steps in firmware analysis to see the relevancy of architecture identification to firmware analysis, and answers the first two research questions, namely "RQ1: What are the current state of the art methods and tools related to identifying binary code architecture and endianness?" and "RQ2: How could existing methods and approaches be improved in terms of algorithms, performance, data sets, and support tools?".

2.1 Unpacking

Zaddach and Costin (2013) state that firmware is usually distributed as an archive or file system image. The authors classify firmwares to three categories based on their complexity. A full-blown firmware consists of a complete Linux or Windows file system. It includes a full operating system, bootloader, applications and required libraries. The second firmware category is integrated firmware, which contains the applications and only a small, proprietary operating system, or none at all. Firmware can also be a partial update containing only updates for some of the files. Depending on the complexity of the packed firmware, it can contain objects such as bootloader, kernel, file-system images, user-land binaries, resources and support files as well as web servers. To access the files inside the packed firmware, it needs to be unpacked. (Zaddach and Costin 2013)

Binwalk (ReFirmLabs 2018), Binary Analysis Tool (BAT) (Hemel et al. 2011) and Binary Analysis Next Generation (BANG) (Hemel 2018) are the de facto tools for unpacking firmwares. They use signatures of common file systems and file formats to unpack firmware images. If a signature matches, the object, e.g., a file system or an individual file, is carved out from the firmware image and extracted as the type is known from the signature. (Liu et al. 2016) As the extraction method is based on signatures, if an unknown pattern or file format is met, the unpacker does not know if it is a successfully extracted data file or if it is just packed in a format the unpacker does not recognize (Costin et al. 2014). The format may not be recognized, because the device manufacturer may have modified an existing compression algorithm or even implemented new compression algorithm (Chen et al. 2016). If the firmware unpacking is not successful because the file format is not recognized, brute force methods can be applied. File carving is a method where every offset of the firmware image is tried with all known unpackers. It is a slow and imprecise approach, but sometimes it is the only way to unpack the files. (Costin et al. 2014)

Automatic unpacking of firmware has its problems. The firmware image can contain compressed data and binwalk has developed a method to recursively extract compressed data from firmware called ’Matryoshka’ mode. This method is not perfect as it can lead to path explosion if it tries to extract data from every executable file within the firmware image, in which case the unpacking is not guaranteed to end within reasonable time or resource, e.g., storage or CPU, constraints. The firmware image may also fail to extract because it only contains a partial file system, multiple embedded or partial file systems, or it may be encrypted. (Chen et al. 2016)

2.2 Identification and classification

After unpacking the firmware, it should be classified as it can make further analysis easier or help unpack firmwares from the same vendor in the future. Costin, Zarras, and Francillon (2017) were the first to apply machine learning in the context of firmware classification. They try to automate finding the brand and the model of the device the firmware is made for and propose different firmware features and machine learning algorithms to get this information. The researchers achieve the best results with random forest classifier with 90% classification accuracy. With the help of statistical confidence interval, the authors estimate that in a data set of 172 000 firmware images, the classifier could correctly classify the firmware in 93.5% ±4.3% of the cases. The research suggests same kind of techniques could be used to complement firmware vulnerability discovery techniques.

This information is valuable if it is collected and saved in a database, where it could be stored along inputs to trigger the vulnerabilities. When the same device with newer firmware version would be encountered, the vulnerability triggering input could be automatically tested to see if the vulnerability was fixed or if it is still present. Also, sometimes different vendors use same Free

Open Source Software (FOSS) components, and if a vulnerability is found from within one of these components, they can then be fingerprinted and used to find same vulnerabilities within different firmware images or on the Internet using for example Shodan, Censys or ZoomEye. The classification information could also be used to create vendor-specific analysis techniques, such as fine-tune unpacking to only run unpackers that have been found effective for the specific device or vendor to save time and computing resources as file carving is not needed. (Costin, Zarras, and Francillon 2017)

2.3 Static analysis

Static and dynamic analysis are common methods of finding vulnerabilities from firmware. In static analysis vulnerabilities are detected from the programs included in the firmware without executing them on a real device or an emulator as opposed to dynamic analysis. (Xie et al. 2017)

Costin et al. (2014) used static analysis in their large-scale study of the security of embedded devices. They justify using it because running the actual physical devices in such a large scale does not scale as it is expensive, the devices might not be operable outside the system they were designed to be run on and because some vulnerabilities can be hard to find just by analyzing the running device. By analyzing the files included within the device firmware, the authors were able to find private RSA keys and self-signed certificates, hard-coded password hashes and possible backdoors. In addition, the researchers were able to correlate the findings to other devices or firmware likely affected by the same vulnerability using correlation techniques. (Costin et al. 2014)

Another way of using static analysis to find vulnerabilities is to detect code that is similar to code that contains a vulnerability. Li et al. (2016) created VulPecker, a code-similarity based framework to detect vulnerabilities through source code analysis. Vulnerabilities are characterized by vulnerability diff hunks, which include both the vulnerable code and the fix to it. Multiple features are extracted from the vulnerable code such as component features, expression features and statement features that describe the different changes made to the code. Component features describe what modifications were done in the patch to the variable types, function names or arguments. Expression features describe what modifications were done to the if, for and while conditions, and statement features describe what lines were added, deleted or moved. These kinds of changes were transformed into numeric values, which were used to train a support vector machine classifier. VulPecker was able to find 40 vulnerabilities that were not published in National Vulnerability Database (NVD), and among these vulnerabilities, 18 of them were not known for their existence. Li et al. (2016)

Finding vulnerabilities from the software components included in IoT firmware using static analysis is challenging, as the source code is not included in most firmware. Dynamic analysis overcomes many limitations of static analysis, but it usually requires an emulator to execute the code (Zaddach et al. 2014) Before the firmware can be emulated for dynamic analysis, the architecture of it needs to be known.

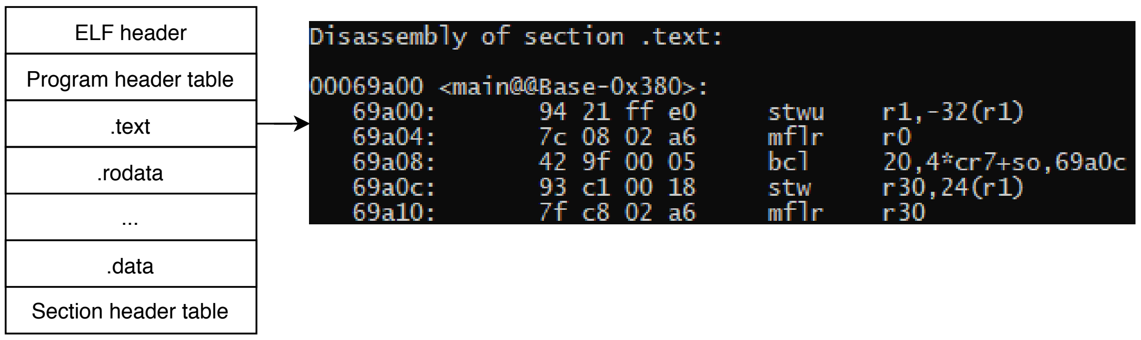
2.4 Architecture identification

Executable files can give a lot of information about the computer they were designed to be run on, and this can be used to identify the architecture. They can contain a header if they are in a standardized file format such as ELF, PE or Mach-O. The header includes information about the offsets and properties of different sections, such as file type, architecture and endianness. (De Nicolao et al. 2018) This information could be useful for analysis, but it is not always available, because, as stated before, the firmware could contain only a minimal set of binaries where header information could be completely missing (Clemens 2015).

The header information is followed by code and data sections. Most IoT and embedded devices are using modified Harvard architecture where the code and data sections are separated (Karagiozidis 2018). Code sections contain primarily code to be executed by the processor while data sections can contain strings, constants or jump tables. This separation is an advantage and can be used to extract only the code sections from a binary file for further analysis.

The code sections contain instructions for the CPU encoded as a byte stream. The byte streams from code sections are also commonly known as object code. Computers are said to share the same architecture if the encoding of these instructions is the same. The encoding is referred to as instruction set or Instruction Set Architecture (ISA). The instructions consist of two parts: opcode and operand. (Clemens 2015) The opcode specifies the operation the program wants to perform. Closely related to the opcode is the endianness, which describes whether the most significant (big endian) or least significant (little endian) byte is ordered first. Operand corresponds to the data used by an instruction. The operand can be immediate, register or memory address. Immediate operand is a fixed value, register operand refers to a register while memory address operand refers to a memory address. (Sikorski and Honig 2012) Figure 1 shows an overview of an ELF file structure. It also shows part of the disassembly of a code section ".text", which is composed of byte streams, e.g., "94 21 ff e0", which evaluates to stwu instruction in powerpc architecture.

The instruction is targeted to operands r1 and -32(r1).



```
Disassembly of section .text:

00069a00 <main@@Base-0x380>:
   69a00:       94 21 ff e0     stwu    r1,-32(r1)
   69a04:       7c 08 02 a6     mflr    r0
   69a08:       42 9f 00 05     bcl     20,4*cr7+so,69a0c
   69a0c:       93 c1 00 18     stw     r30,24(r1)
   69a10:       7f c8 02 a6     mflr    r30
```

Figure 1: Example of an ELF file structure

McDaniel and Heydari (2003) introduced the idea to use file contents to identify the file type. Previous methods utilized metadata such as fixed file extension, fixed "magic numbers" and proprietary descriptive file wrappers. The authors propose three file recognition algorithms: byte frequency analysis, byte frequency cross-correlation analysis and file header/trailer analysis. They showed that many file types have characteristic patterns that can be used to differentiate them from other file formats. They reported accuracies ranging from 23% to 96% depending on the algorithm used. (McDaniel and Heydari 2003)

Clemens (2015) applied the techniques introduced by McDaniel and Heydari (2003) and proposed methods for classifying object code with its target architecture and endianness. The author used byte-value histograms combined with signatures to train different classifiers using supervised machine learning. It is a type of machine learning where the corresponding correct output is known and it is used to train the model, which then tries to make predictions about future instances (Kotsiantis, I. Zaharakis, and P. Pintelas 2007). This approach produced promising results and showed that machine learning can be an effective tool when classifying architecture of object code. Classification accuracy varied depending on the used classifier from 92.2% to 98.4%. The authors would like to have a larger data set including more embedded platforms, microcontroller code and more "GPU code" samples. They would also want samples using different compilers such as LLVM/Clang and Microsoft Visual Studio to show that the compiler does not have an effect on the classification performance. (Clemens 2015)

Clemens (2015) assumes that different CPU architectures have different byte frequency distributions. The author calculates the frequencies of each byte in each object code sample and normalizes it by dividing it by the total number of bytes in a sample. The calculation of the frequency of each byte i can be formulated as follows:

$$f_i = \frac{count(i)}{\sum_{i=0}^{255} count(i)} \tag{2.1}$$

Because some architectures use the same opcodes and differ only by their endianness, Clemens (2015) introduced four patterns to determine the endianness. He utilizes common operations such as incrementing by one and typical stack values to predict if the sample is little or big endian. For example, on big endian architectures, one is presented as 0x00000001, while on little endian architectures it is presented as 0x01000000. As adding one is more likely to occur than adding 16777216, on big endian samples 0x00000001 should occur more often than 0x01000000. Because of this assumption, Clemens (2015) proposed four 2-byte frequency counts to determine the endianness: {’0xfffe’, ’0xfeff’, ’0x0001’, ’0x0100’}. (Clemens 2015)

De Nicolao et al. (2018) use the work of Clemens (2015) as part of their research. Their objective is to separate code sections from binary executables, but as a preliminary step they need to identify ISA if it is unknown. They extend the feature set of Clemens (2015) by adding more signatures to use as an input for the classifier. They obtain a global accuracy of 99.8%, which is higher than Clemens (2015) acquired. The authors name few things to improve on architecture detection in further research. First, they only use byte level features and ignore instruction-level features such as trying to group bytes corresponding to code into valid instructions. Second, they acknowledge that when testing the model against complete binaries with both code and data sections, large data sections could affect byte frequency distribution and because of that, the architecture detection. They also failed to identify architecture of packed files because of the high-entropy uniform data. Finally, they did not test their framework on real firmware because of the lack of ground truth data, which would require manual reverse engineering of each sample. (De Nicolao et al. 2018)

De Nicolao et al. (2018) added more signatures in addition to the four endianness features by Clemens (2015) from angr framework (Shoshitaishvili et al. 2016). It contains a collection of function epilogs and prologs for some architectures which might help differentiate between architectures with the same opcodes but different endianness. The features are calculated by obtaining the number of matches in one sample and normalizing it by dividing the number of bytes in the sample:

$$f_{pattern} = \frac{\#matches(pattern, file)}{len(file)} \tag{2.2}$$

Overfitting is a common problem in supervised learning, where the model fits the training data too closely, which increases the generalization error, i.e., makes the model perform worse when predicting outcome for values of previously unseen data. (Ng 2004) Regularization is often used in machine learning to avoid overfitting, for example when there is not enough training data or there is a large number of features to be learned. In addition to additional signatures, De Nicolao et al. (2018) used L1 regularization in their logistic regression model. L1 regularization encourages sparsity of the model, i.e., model having only a few nonzero components, by adding a L1 penalty to the coefficients. This way the models become more simple and easily interpreted, consisting of only the significant features. (Schmidt, Fung, and Rosales 2007) On the other hand, using regularization makes training more computationally expensive and in case of L1 regularization, makes some solvers unusable. (Lee et al. 2006)

There exists also other research on recognizing the architecture of a binary. Costin, Zarras, and Francillon (2016) go through every executable inside the firmware and use ELF headers to identify the architecture if the header is present. The authors determine the architecture of the firmware by counting the amount of architecture specific binaries the firmware contains. This information is used to launch an emulator for that specific architecture.

cpu_rec (Granboulan 2018) uses the fact that the probability distribution of one byte depends on the value of the previous one. The author uses Markov chains as it composes of conditional probabilities of bytes when knowing the previous one. To do classification based on Markov chains, Kullback-Leibler divergence is used to calculate how much the Markov chains for different architectures differ. It also utilizes a sliding window, so the solution should be able to handle files with code for multiple architectures in them. The author does not publish any performance measures other than that the analysis of 1 MB takes 60 seconds on 1GB of RAM. A future research idea is suggested to find a more precise method for detecting the the start of the executable code.

Binwalk (ReFirmLabs 2018) uses architecture specific signatures (33 signatures for 9 architectures as of 3.2.2019) and Capstone disassembler (9 configurations for 4 architectures as of 3.2.2019) to identify the architecture. As expressed by Clemens (2015), using only signatures has some limitations. They can lead to false positives if the signatures are not unique compared to the other architectures. Also the disassembly method requires complete support from a disassembler framework, which might not always be available or working perfectly.

The classifiers proposed in the previous research have good performance regarding the accuracy, but there is a gap of collecting and pre-processing binaries from different architectures for analysis purposes. For example, the

Clemens (2015) data set is not publicly available and is limited in size. Acquiring such data sets or developing own tools takes a long time, huge human effort, and can be prone to errors. As a part of this thesis, tools to obtain a data set will be provided. This allows researchers to focus on developing, extending and improving the classifiers and not waste valuable time on setting up the infrastructure and acquiring the data set.

There is also a need for more comprehensive evaluation of classifier performance on full binaries that includes more classifiers and architectures. De Nicolao et al. (2018) tested only the performance of logistic regression on a small data set consisting of full binaries. In this regard, the present thesis bridges this gap by conducting a larger experiment, where the classification performance is tested with a large dataset consisting of full binaries for as many architectures as possible. Different classifiers are also used to see if some classifiers perform better than others when classifying full binaries.

2.5 Emulation

Unlike static analysis, dynamic analysis relies on executing the software by running the firmware either on real device or inside an emulator. Costin, Zarras, and Francillon (2016) list different ways to emulate the firmware. In perfect emulation the firmware is complete, including bootloader and kernel, and there exists a perfect emulator configuration that mimics the hardware. Because of the huge amount of different CPU architectures and custom hardware used, this type of setup is not feasible to implement. (Costin, Zarras, and Francillon 2016)

Another type of emulation reuses the original kernel. It would lead to more accurate emulation, but unfortunately kernels for embedded systems are often customized, so they would not work well with generic emulators. The researchers also noticed that only 5% of the firmware images in their data set contained the kernel. (Costin, Zarras, and Francillon 2016)

Third type of emulation uses a generic system that has the same CPU architecture as the firmware as a base for the analysis. Firmware would be unpacked inside the system and programs would be executed inside the generic system. It allows the programs to be run in a consistent environment where they can be monitored and controlled. This approach has few drawbacks, for example the emulation is quite slow and the firmware can’t be fully emulated with specific kernel extensions. However, the researchers found this approach to offer the best trade-off between emulation accuracy, complexity and speed. (Costin, Zarras, and Francillon 2016)

The last emulation method would improve the performance and emulation management by using chroot to emulate an environment for architectures other than the host architecture. It would utilize Linux kernel's ability to call an interpreter to execute an ELF executable for a foreign architecture. The authors found this method unfeasible because it was not stable. (Costin, Zarras, and Francillon 2016)

These kinds of ideas were utilized by Chen et al. (2016) in creation of FIRMADYNE, a software-based full system emulation using custom kernels. It falls into the third category presented by Costin, Zarras, and Francillon (2016) as it emulates the CPU architecture of the device and unpacks the firmware inside it. FIRMADYNE improves the emulation by implementing a "learning" mode, where the kernel records all system interactions with the networking subsystem, allowing the kernel to communicate with the emulated firmware in the final emulation. The researchers were able to successfully emulate over 96.6% of the Linux firmwares of their test set consisting of 9486 firmware images. The authors acknowledge that FIRMADYNE might become unstable if the firmware image holds out-of-tree kernel modules that clash with the kernel version used in emulation, but this did not become a problem in more than 99% of the cases.

2.6 Dynamic analysis

When the firmware is successfully emulated, it can be analyzed using wide range of different techniques such as symbolic and concolic execution, and fuzzing.

Normally when a program is executed, it is run with a specific input and a single control flow path is explored. With symbolic execution multiple paths that a program could take under different inputs can be explored simultaneously. (Baldoni et al. 2018) Problem with running symbolic execution on firmware is the lack of source code that many symbolic execution engines require. Some symbolic execution engines are based on the instrumentation and execution monitoring of firmware on real devices. They require custom software to be run on the device, which might not be possible, because firmware is frequently cryptographically signed by the manufacturer and because the large amount of different hardware architectures would require per-device development effort. (Shoshitaishvili et al. 2015)

Firmalice was able to tackle these problems using concolic execution, which is comprised of both concrete, i.e., testing on particular inputs, and symbolic execution. However, it is not a fully automated solution, as it needs the analyst to define the security policy, which describes operations that require authentication. (Shoshitaishvili et al. 2015)

Fuzzing is a type of dynamic analysis and can be defined as "a method for discovering faults in software by providing unexpected input and monitoring for exceptions" (Sutton, Greene, and Amini 2007). It has become one of the most popular ways of finding vulnerabilities from programs and has been used to find critical security vulnerabilities such as Heartbleed which other analysis methods did not find. The OpenSSL code was too complex for static analysis and dynamic analysis methods are not designed to find vulnerabilities like Heartbleed, which required incorrect or unexpected input. (Wheeler 2014). Fuzzers are given a set of known inputs, which they start their analysis from, and either mutate those inputs or learn the input format to generate new inputs (Rawat et al. 2017).

Dynamic and static analysis can be used in combination to discover vulnerabilities from firmware. Costin, Zarras, and Francillon (2016) used both static and dynamic analysis to find vulnerabilities from embedded web interfaces. They used static analysis to analyze the web document root for vulnerabilities in the web applications and then dynamic analysis to confirm these vulnerabilities as one of the disadvantages of static analysis tools is the high number of false positives.

2.7 Domain-specific analysis

Where static and dynamic analysis can find weird, unintended program states, i.e., vulnerabilities, the devices can also hold backdoors in them, which can manifest as explicit, intentional or even normal program functionality like debug functions, hard coded credentials or ways of automating software updates without the need of user interaction. Attackers with the knowledge of a backdoor or backdoors in a specific device could use them to bypass authentication, escalate privileges, or perform or access functionality that is not available to a normal, legitimate user. (Thomas and Francillon 2018)

A backdoor is composed of four components: input source, trigger, payload and privileged state. A backdoor is activated by a trigger, which is waiting for satisfying the trigger condition. It could be a specific payload, e.g., hard coded password, input causing a buffer overflow or even a single UDP packet (Heffner 2013), from a specific input source, e.g., socket or standard input. The successful activation of a trigger transitions the system state from normal to a backdoor-activated state, which is essentially a state of escalated privileges, i.e., a privileged state. (Thomas and Francillon 2018)

In some cases, it can be hard to say if a backdoor was placed on a system intentionally, or if it is just a vulnerability. In some cases the backdoor is clearly intentional as the transition to the privileged state is explicit, for example when hard-coded credentials are used to bypass user authentication, but if the transition is non-explicit, i.e., based on a bug, it can be hard to say if it is intentional or not. An example of this is a buffer overflow vulnerability, which is a backdoor if it is was placed there deliberately, and just a vulnerability if not. From a technical point of view it is indistinguishable from an accidental vulnerability, but in some cases from a non-technical point of view could be argued as intentional. To get a sense of the intention, version control logs could be reviewed to see if there was a reason to change the software, or if the code is unreachable by normal program control-flow. (Thomas and Francillon 2018)

Finding backdoors is still a manual process which is comprised of reverse engineering binaries and listening to system events for suspicious activity. Some semi-automated solutions exist such as Firmalice, (Shoshitaishvili et al. 2015), which tries to find a path to a privileged state without proper credentials, HumIDIFy (Thomas, Garcia, and Chothia 2017), which uses both machine learning and targeted static analysis to find abnormal behavior from services used in Linux-based embedded firmware, and Stringer (Thomas, Chothia, and Garcia 2017), which looks for functionalities only executed by a successful comparison with static data. (Thomas and Francillon 2018)

3 Machine learning

This chapter reviews scientific literature and introduces main concepts of machine learning. It first explains the difference between machine learning, artificial intelligence and deep learning. It then goes into detail with machine learning explaining different kinds of classification tasks and techniques to find out what classification category architecture identification falls into, what metrics exist to measure classifier performance and how the performance of different classifiers can be evaluated.

3.1 Overview of artificial intelligence and its subfields

Many times, especially in media, terms Artificial Intelligence (AI), Machine Learning (ML) and Deep Learning (DL) are used interchangeably, while they represent different things. This section tries to define AI and ML, while a brief description of DL is given in Section 3.5.

AI is one of the newest fields in science. It made its appearance in 1956 and while there is no single formal definition of it, it could be seen as a field both trying to understand how humans perceive, predict and manipulate the world we live in, and also how to build intelligent entities. (Russell and Norvig 2016)

It holds many different subfields, mainly Natural Language Processing (NLP), knowledge representation, automated reasoning, machine learning, Computer Vision (CV) and robotics. They focus on different problems in the vast field of AI. NLP seeks to enable communication through natural language like English while knowledge representation tries to store the knowledge, i.e., what it knows or hears. Automated reasoning uses the stored knowledge to both answer questions as well as to draw new conclusions from it. Machine learning tries to adapt to new circumstances and to detect patterns. Computer vision studies how entities could perceive objects and robotics seeks to interact with the objects by manipulating and moving them. (Russell and Norvig 2016) While all of the different aspects of AI are equally important, this thesis focuses on machine learning as the objective is to find and utilize patterns from executable files.

Machine learning can be seen as a "set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under certainty" (Robert 2014). It has many uses cases and has driven advances in many different fields, which can be explained by creation of more sophisticated machine learning models, large and available data sets and platforms that allow the use of large amounts of computing resources to be used to train models based on these large data sets

(Abadi et al. 2016). For example, challenges in data storage, organization and searching created a whole new field of research, data mining, while statistical and computational problems in biology and medicine created bioinformatics (Friedman, Hastie, and Tibshirani 2009). In cyber security, machine learning is used in, for example, intrusion detection, identifying botnet traffic, firmware analysis and as in this thesis, architecture identification (Buczak and Guven 2016; Livadas et al. 2006; Costin, Zarras, and Francillon 2017).

Machine learning can be divided into two main types: predictive or supervised learning, and descriptive or unsupervised learning. Supervised learning is done using data with known labels or classes (the corresponding correct output) while in unsupervised learning the data is unlabeled. (Kotsiantis, I. D. Zaharakis, and P. E. Pintelas 2006) There is also a third type, reinforcement learning, that is discussed briefly in Section 3.4.

3.2 Unsupervised learning

The most noticeable difference between unsupervised and supervised learning is that unsupervised learning does not have a direct measure of success: the class labels in supervised learning are not present here. Heuristic arguments must be used to motivate the algorithm and to judge the quality of the results. (Hastie, Tibshirani, and Friedman 2009) Classifying objects using unsupervised learning is called clustering. The main difference between clustering and classification is that classification has well-defined target classes defined by the training data, while clustering tries to define these classes based on the data and the similarity measures (Mukhopadhyay 2018).

Cluster analysis is a way to perform clustering. It can be seen as a way of finding patterns from data that would be considered pure unstructured noise (Ghahramani 2004). It uses distance or dissimilarity measures that are used to group similar objects together into clusters, so that those within the cluster are more closely related to one another than objects assigned to different clusters (Dy and Brodley 2004; Ghahramani 2004).

3.3 Supervised learning

Supervised learning is based on using input data, which is either measured or preset, to predict the values of the outputs. For example, in this thesis the object is to classify binaries based on their content (input) to match an architecture (output). The predictions supervised learning algorithms make can be categorized to classification if the predicted value is discrete, and regression if the predicted value is continuous. (Friedman, Hastie, and Tibshirani 2009)

Architecture identification falls into classification, as the output is a prediction of the class label. In case of regression, the output could be the byte frequency distribution of a binary for a different architecture.

This thesis explores different classifiers' ability to identify the architecture based on the binary file contents, i.e., classify binaries based on the architecture. Classification is a basic task that requires the construction of a classifier, which assigns a class label to instances described by a set of features. It can be divided into binary, multi-class, multi-label and hierarchical tasks. Binary classification is the most popular classification task and it tries to classify the input into one, and only one, of two non-overlapping classes. In multi-class classification the input is classified into one, and only one, of *n* non-overlapping classes. Multi-label classification classifies input into several of *n* non-overlapping classes. In hierarchical classification the input is classified into class, which are divided into subclasses or grouped into superclasses. (Sokolova and Lapalme 2009) The hierarchical approach allows the problem to be decomposed into smaller problems which can be solved more efficiently. For example, the classifier can first classify the input into one of the top level classes and then into the class only within the top class. (Dumais and Chen 2000) The identification of architecture falls into the multi-class classification as there is input (byte frequency distribution and signatures) which corresponds to one of the many different architectures.

Most of the times multi-class classification problems can be decomposed into many binary ones and sometimes this is needed to be able to use some classifiers (Li, Zhang, and Ogihara 2004). For example, perceptron-like classifiers are binary, so to use them in multi-class problems, the problem must be reduced to a set of multiple binary classification problems (Kotsiantis, I. D. Zaharakis, and P. E. Pintelas 2006). The decomposition is usually done either with "One-vs-One" (OVO) or "One-vs-All" (OVA) strategy. In OVO, the problem is divided into many binary problems and one classifier is trained to differentiate between two classes. The outputs of all the classifiers combined and used to predict the output class. In OVA, a classifier is trained for each class, so the classifier giving a positive answer indicates the output class. (Galar et al. 2011) An example workflow of supervised learning can be seen in Figure 2. The following sections go through the different steps, as it is used as a basis for the practical part of this thesis.

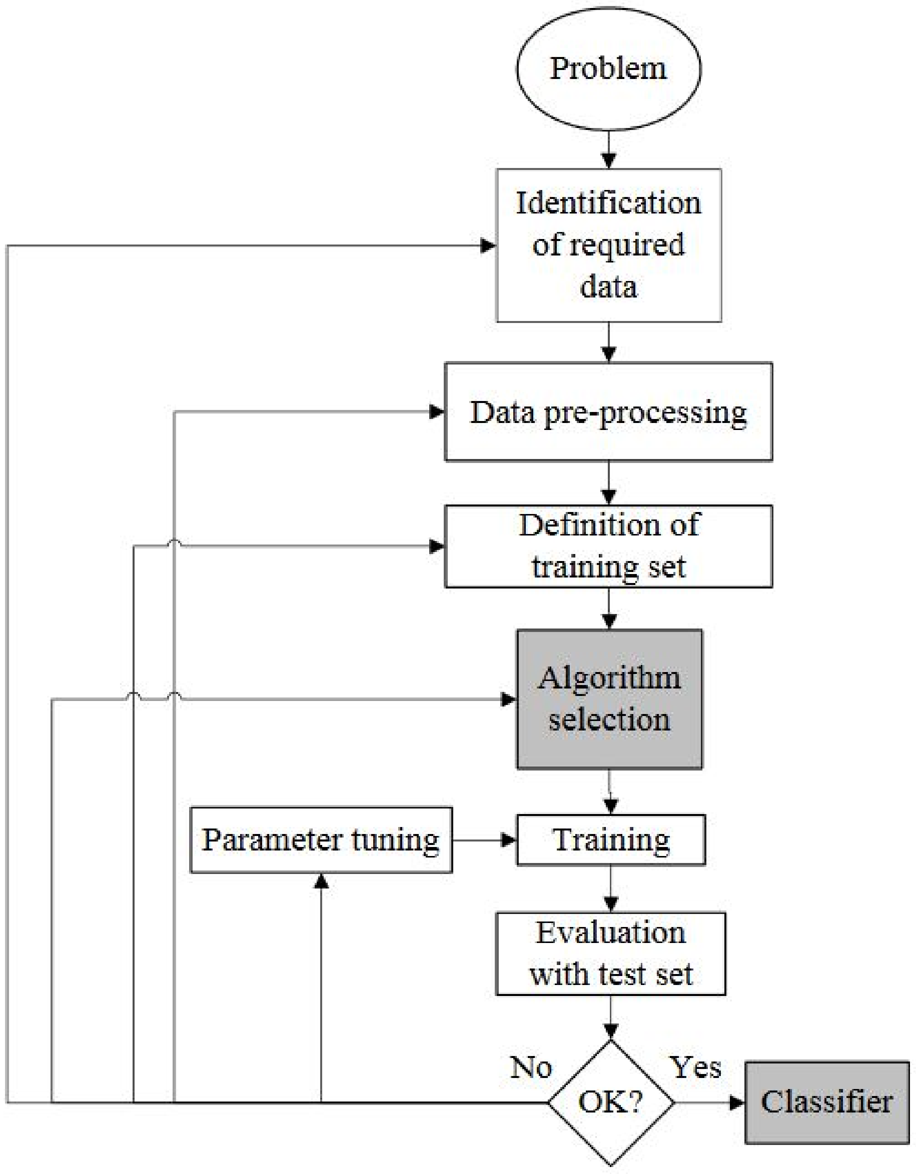


Figure 2: Overview of steps in supervised learning (Kotsiantis, I. Zaharakis, and P. Pintelas 2007)

3.3.1 Identification of required data

To be able to test the classifier performance, a data set is needed. First, it needs to include as many architectures as possible, so the classifier's ability to differentiate between many different architectures can be seen. Second, the data set should be as big as possible. For example, Rahimi and Recht (2008) were able to reduce testing error by up to 40% when doubling the data set size. Third, the data set acquisition should be automated to minimize the time needed to gather the data set, to be able to handle new architectures and to reduce errors from manual work. And last, because supervised learning is used, the data needs to contain the ground truth values, i.e., what architecture the binary file is made for.

The Debian repository was chosen as the data source for the data set for this thesis, because it contains Debian Linux installation medias for many different architectures. It is a trusted source and installation media downloaded from there can be assumed to contain only executables for the specific architecture. One full installation of Debian can contain up to 1300 files, which contain executable files for the specific architecture, which is used to train the classifiers (Clemens 2015). Debian Linux distribution currently supports 10 architectures, but when combining architectures that are not supported any more, and architectures that have Debian ported to them, Debian installation media can be acquired for 24 architectures.

3.3.2 Data pre-processing and feature selection

Before the data can be used to train a classifier, it needs to be pre-processed. The goal is to have a comma-separated list containing the features for each file in the data set. The Debian repository contains only the installation media, which needs to be downloaded and unpacked to access the binaries. Also, the code sections need to be extracted from the binaries, as using the whole binary can affect the byte frequency distribution.

To obtain a data set and to pre-process it, a toolset [1] was developed for this thesis, which first scrapes the Debian repository for Debian installation media. The workflow can be seen in Figure 3.

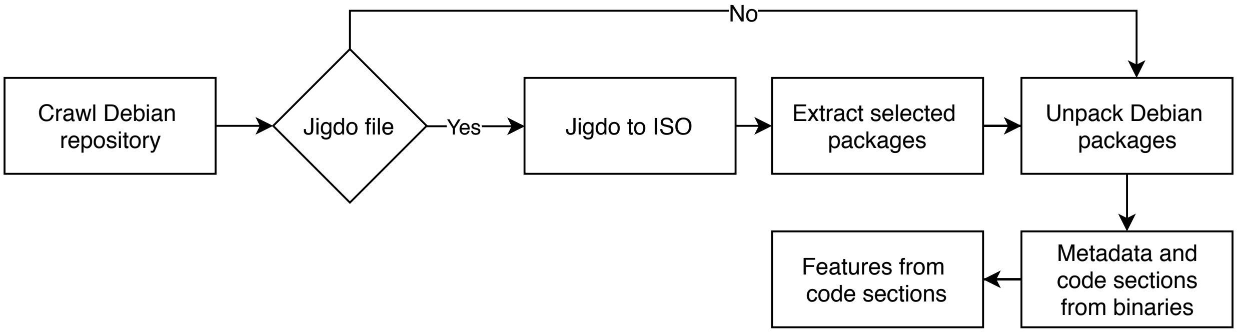

No
Crawl Debian repository
Jigdo file
Yes
Jigdo to ISO
Extract selected packages
Unpack Debian packages
Metadata and code sections from binaries
Features from code sections

Figure 3: Workflow of the tool to download data sets

Both "Debian CDs/DVDs archive" (2019) and "Debian Ports" (2019) repositories are crawled using "firmadyne scraper" (2018) to obtain Jigdo files and Debian packages for a total of 23 architectures.

1. https://github.com/kairis/

In this part the execution of the workflow forks. If the file is a Jigdo file, the workflow continues normally, but in case of the Debian ports, they are downloaded straight as Debian packages and the steps where ISO files are downloaded and extracted can be skipped.

Jigdo (Jigsaw Download) is a way to download large files, such as Debian CDs and DVDs, which get updated often. It downloads all the Debian packages defined in the Jigdo file to assemble one ISO file, which ideally, i.e., no bugs, is supposed to be an exact copy of an officially released CD/DVD. (“jigdo” 2018) The process continues until all the Jigdo files are converted into ISOs.

Next, the ISO files are mounted using *fuseiso*, which allows mounting as non-sudo user. To keep the storage requirements manageable, the framework extracts only the Debian packages specified by the user from the ISO files. To achieve this, “Debian packages” (2018) is crawled to get a list of packages to be extracted. The configuration file of the framework lists categories of files to be downloaded from the site. For this thesis, only web servers, shells, utilities, embedded software, kernels, libraries and administration utilities were downloaded. It does not matter which categories are downloaded, but another student’s thesis needed relevant software that could be found from IoT/embedded devices, so the categories were chosen for that purpose.

Next, the Debian packages are unpacked using *dbkg-deb*. The Debian package should include a control file, that holds a lot of metadata including the software name, version information, file listing and a MD5 sum of each file. All of this information is not used for this thesis, but could be useful in other types of research.

Executable files are located from the extracted Debian packages using Linux utility *file* and also metadata such as architecture and endianness is collected. The code sections are separated from the executables by first finding the code sections with *objdump* the same way Clemens (2015) did: extracting only sections that are labeled as executable code, and then carving them from the executable using *objcopy*.

Every module in the toolset outputs a JSON file that contains relevant information for each step and it is used as an input for the next module in the workflow. The resulting JSON files are documented in Appendix A

3.3.3 Algorithm selection

To choose a fitting algorithm to the problem, different classifiers should be evaluated, as the performance of classifiers tends to vary depending on the data set. Kiang (2003) reviews the performance of well-known classification methods based on statistical and AI techniques: neural networks, decision tree method (C4.5), linear discriminant analysis, logistic regression analysis and kth-nearest-neighbor models. They use synthetic data to perform a controlled experiment where the data can be manipulated to introduce imperfections to it, which is used to investigate how the classifiers perform when certain assumptions about the data characteristics are violated. The results showed that neural nets and logistic regression classifiers perform the best under most scenarios, but the authors recommended to employ a number of different classification algorithms as classifier performance may vary based on the data. Kotsiantis, I. Zaharakis, and P. Pintelas (2007) came also to the conclusion that no single learning algorithm can uniformly outperform other algorithms over all data sets.

3.3.4 Evaluation

There exist many commonly accepted performance evaluation measures for machine learning, mainly accuracy, precision, recall, F-measure (also known as F1 score or F score) and Area Under the Curve (AUC). They are based on values from confusion matrix, which shows the correctly and incorrectly recognized samples for each class as can be seen in Table 1. (Sokolova, Japkowicz, and Szpakowicz 2006)

Table 1: A confusion matrix for binary classification (Sokolova, Japkowicz, and Szpakowicz 2006)

Class / Recognized	as Positive	as Negative
Positive	tp	fn
Negative	fp	tn

In multi-class situation, different average values can be taken to evaluate the performance over all classes. In macro-averaging, the evaluation measure is computed for each class within the data set and an average is taken over all of the classes. In micro-averaging, the measure is calculated globally, so large classes will dominate small ones having a bigger effect on the performance measure. (Uysal 2016)

3.3.5 Evaluation metrics

Accuracy is a good measure for practical analysis when the classes are balanced (Amancio et al. 2014; Buczak and Guven 2016). It is defined as the the sum of true positives and true negatives divided by the total number of predictions (Amancio et al. 2014) and it assesses the the overall effectiveness of the algorithm (Sokolova, Japkowicz, and Szpakowicz 2006).

$$accuracy = \frac{tp + tn}{tp + tn + fp + fn} \tag{3.1}$$

Precision is a function of correctly classified (true positives) divided by total number of samples predicted as positives (Sokolova and Lapalme 2009). It estimates the predictive power of the algorithm. (Sokolova, Japkowicz, and Szpakowicz 2006)

$$precision = \frac{tp}{tp + fp} \tag{3.2}$$

Recall, or sensitivity, is defined as correctly classified positives divided by the number of positive samples in the data and it measures the effectiveness of the classifier to identify positive labels (Sokolova and Lapalme 2009).

$$recall = \frac{tp}{tp + fn} \tag{3.3}$$

F-measure is a standard way of summarizing precision and recall (Castillo et al. 2007). It favors precision when $\beta > 1$ and recall otherwise (Sokolova, Japkowicz, and Szpakowicz 2006).

$$F - measure = \frac{(\beta^2 + 1) * precision * recall}{\beta^2 * precision + recall} \tag{3.4}$$

The F1-score is a special case of F-measure when $\beta = 1$. It simplifies the metric to a harmonic mean of precision and recall.

$$F1 - score = \frac{2 * precision * recall}{precision + recall} \tag{3.5}$$

Receiver Operating Charasteristic (ROC) curves have been used in machine learning to evaluate classification performance between classifiers as they depict the trade-off between hit (true positive) and false alarm (false positive) rates. As ROC curve is a two-dimensional representation of classifier performance, AUC is used to represent the ROC performance in a single scalar value. (Fawcett 2006) When AUC is calculated only using a single point on the ROC curve, it can be seen as balanced accuracy (Sokolova, Japkowicz, and Szpakowicz 2006), representing classifier’s ability to avoid false classification (Sokolova and Lapalme 2009).

$$AUC_b = \frac{sensitivity + specificity}{2} \tag{3.6}$$

where

$$specificity = \frac{tn}{fp + tn} \tag{3.7}$$

Ferri, Hernández-Orallo, and Modroiu (2009) showed that accuracy, F-score and AUC gave good results when evaluating classifier performance. Accuracy and F-score performed well under noisy data set and they handled data sets with concept drift or other strong changes in the data set. AUC is best when distortion is produced as a product of using a bad algorithm or too small training set.

(Sokolova, Japkowicz, and Szpakowicz 2006) evaluated different performance measures and came to the conclusion that higher accuracy does not guarantee better overall performance of a classifier and that a combination of measures should be used, as they give a balanced evaluation of the classifier's performance.

3.3.6 Cross-validation

To evaluate the generalization ability of the classifiers, i.e., the performance on previously unseen data, cross-validation is used. There exists a few variations of it: holdout method, k-fold cross-validation and leave-one-out (Cawley and Talbot 2003). In the holdout method, the data set is split into two mutually exclusive subsets: a training set and a test set. For example, 2/3 of the data is used for training and the rest for testing. The weakness of this method is that it makes inefficient use of the data as a portion of the data set is not used for the training. (Kohavi et al. 1995)

To use the data set more efficiently, k-fold cross-validation can be used to split the data into k mutually exclusive subsets, or folds. The classifier is trained k times and each time a different fold is used for testing and rest for training the classifier. Accuracy and other performance measures are calculated as the mean of the results obtained with different folds. In addition, stratified cross-validation can be used to ensure the fold contain approximately the same proportions of classes as the original data set. (Kohavi et al. 1995) 10-fold cross-validation will be used in this thesis when evaluating classifier performance in the experiments.

Leave-one-out is a special case of k-fold cross validation, in which the number of folds equals the number of entries in the data set (Wong 2015). It has less bias than 5-fold or 10-fold cross-validation, but adds computational cost and is recommended to use only on smaller data sets where it is computationally feasible (Rodriguez, Perez, and Lozano 2010).

If the classifier performance evaluated using cross-validation is unsatisfactory, one should return to a previous step in the workflow. Bad classifier performance could be explained by choosing either wrong features or using too many of them, using too small or imbalanced data set, or choosing the wrong algorithm for the problem. (Kotsiantis, I. Zaharakis, and P. Pintelas 2007)

3.4 Reinforcement learning

Reinforcement learning is closest to the kind of learning humans and other animals do. It is based on goal-directed learning where the learner is not told what to do, but must instead discover which actions yield the most reward by trying them. To do this, a learning agent is deployed, that can sense the state of the environment to some extent, can make actions to affect the state, and which has a goal or goals set related to the state of the environment. (Sutton and Barto 2018)

To achieve the maximum reward, the agent must tackle the trade-off between exploration and exploitation. The agent needs to exploit what it has already experienced, i.e., prefer actions that it has previously found effective at producing reward, but on the other hand it needs to explore in order to make better actions in the future. The agent needs to try different combinations of actions and through failures favor the actions that appear to be the best. It is a hard problem studied by mathematicians and remains yet to be solved. (Sutton and Barto 2018)

Compared to supervised learning, reinforcement learning does not need labeled data, which might be hard or impossible to acquire for interactive problems. In these situations reinforcement learning is at its best as it has to learn from its own experiences. (Sutton and Barto 2018)

Even though both unsupervised learning and reinforcement learning can work with unlabeled data, unsupervised learning tries to find hidden structures from unstructured data while reinforcement learning tries to maximize the reward. Finding structures might help the learning agent, but it is not the problem reinforcement learning tries to solve. (Sutton and Barto 2018)

3.5 Deep learning

Conventional machine learning methods are limited in their ability to process natural data in its raw form. It might be hard to say what features should be extracted. For example, to train a program to identify cars from a picture, the presence of wheels could be used as a feature, but it is hard to describe to a machine what a wheel looks like when all the machine has available are the pixel values. The wheel has a simple geometric shape, but is affected by shadows, sun glares, different objects obscuring parts of the wheel and other variations. Representation learning was created to tackle this problem by allowing a machine to be fed with raw data, which would be used to automatically discover the features. (LeCun, Bengio, and Hinton 2015; Goodfellow, Bengio, and Courville 2016)

Representation learning might still fail in real-world applications because of the variations explained. Deep learning tries to solve this problem by using many representations that are expressed in terms of other, simpler representations. For example, an image of a person can be imagined to be composed of corners and contours, which are in turn defined as edges. This allows building deep models with many simple layers, which achieves great power and flexibility to represent world as a nested hierarchy of concepts. (Goodfellow, Bengio, and Courville 2016)

3.6 Classifiers

Classifiers can be divided into Bayesian, lazy classifiers, trees and functions (Amancio et al. 2014). The following sections go through the basics of some of the classifiers used in the following experiments and also give examples for what purposes they have been used in cyber security.

3.6.1 Bayesian classifiers

Bayesian classifiers assign membership probabilities to classify new objects (Amancio et al. 2014). They are based on the Bayes theorem states that given a hypothesis H of classes and data x, then

$$P(H|x) = \frac{P(x|H)P(H)}{P(x)} \tag{3.8}$$

where

P(H) represents the prior probability of each class without information about the variable x,

P(H|x) represents the conditional probability of H given the likelihood x and

P(x|H) represents the conditional probability of x given the likelihood H. (Dua and Du 2016)

Bayesian network can be presented as a network where the nodes represent the variables and arcs represent the probabilistic dependencies between the variables. If two nodes are unconnected, they are said to be independent of each other. (Dua and Du 2016) An example of how a Bayesian network could look like is shown in Figure 4.

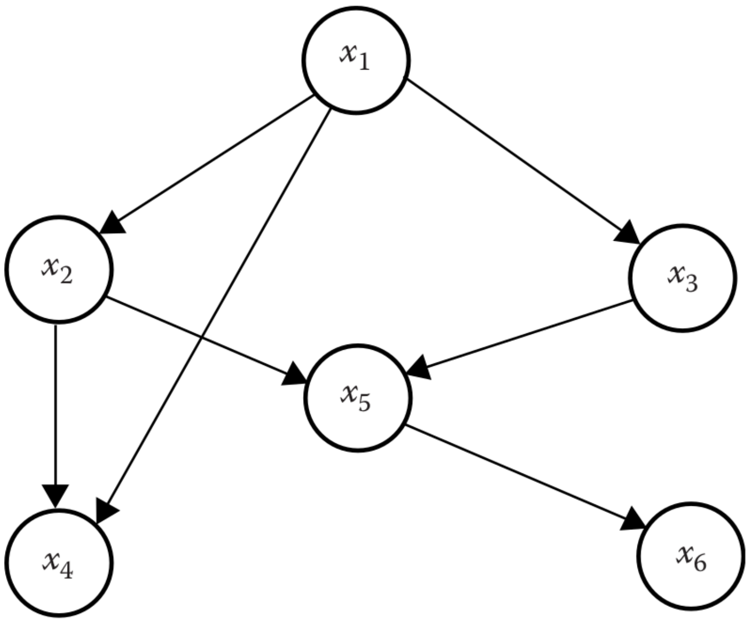


Figure 4: Bayesian network representation (Dua and Du 2016)

Naive Bayes is a simple Bayesian network that assumes conditional independence and normal distribution for attributes in every class (Kotsiantis, I. D. Zaharakis, and P. E. Pintelas 2006). In practice this means that the classifier assumes that the features are independent for each class. Even though this assumption rarely holds making the probability estimates inaccurate, it has given good classification results and has been proven effective in many practical applications such as text classification. Research has shown that naive Bayes performs the best when the features are completely independent, but also when there are functionally dependent features. (Rish et al. 2001) The good classification results have also been explained to be caused by dependencies either canceling each other out or working together to support a certain classification (Zhang 2004)

Even though naive Bayes is a simple method, it has been able to outperform more complex algorithms. Yousefi-Azar et al. (2017) studied the performance of different classifiers using features generated with autoencoder in both malware classification and intrusion detection. Out of the four classifiers used, naive

Bayes performed the best in detecting intrusions, achieving 83.3% accuracy compared to the second best, Xgboost at 78%.

3.6.2 Lazy classifiers

Lazy, or instance-based classifiers store the training data and only build the model when the data needs to be classified. Other classifiers introduced in this section are eager classifiers, which generalize the training data before attempting to classify (Amancio et al. 2014). This makes lazy classifiers fast to train, but increases classification time and required storage space. (Dua and Du 2016)

k-Nearest Neighbor (kNN) is a lazy classifier, that uses a simple technique which has been proved effective in the field of pattern recognition, but which on the other hand has a high memory requirement and computational complexity (Bhatia et al. 2010). It classifies a data point based on the k data points nearest to it. As seen in Figure 5, from the five nearest data points, three are negative points, so the data point X_{query} would be classified as negative with a confidence of 3/5. (Dua and Du 2016)

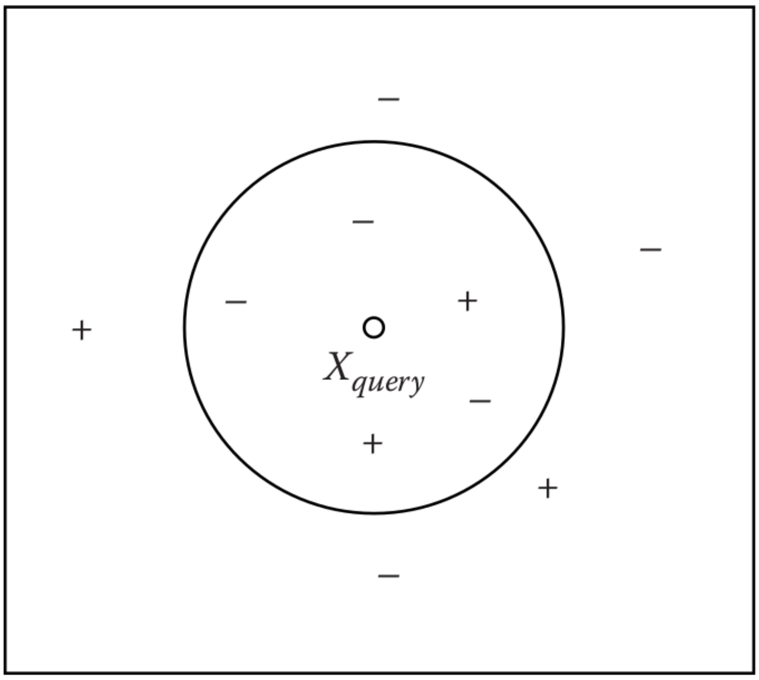


Figure 5: Example of a kNN where k=5 (Dua and Du 2016)

There are many different metrics to calculate the distance between the data points such as Euclidean, cosine, Chi square, and Minkowsky distances (Hu et al. 2016). The most employed distance metric is the Euclidean distance (Dua and Du 2016): if the distance between two data points x_1 and x_2 needs to be calculated in a n dimensional feature space, it can be calculated as

$$dist(x_1, x_2) = \left(\sum_{i=1}^{n} (x_{1i} - x_{2i})^2 \right)^{0.5} \tag{3.9}$$

kNN has many use cases in cyber security. It has been successfully used in classifying malware by using image processing techniques to analyze gray-scale representations of malware binaries. The researchers were able to achieve 97.18% classification accuracy in a data set consisting of 9458 malware samples from 25 different malware families. (Nataraj et al. 2011) kNN has also been used in classifying emails to automatically detect spam and scam emails. It was one of the three algorithms used to vote if the email is legitimate or not. It achieved an accuracy of 91.5% with a data set consisting of 4500 spam emails, 1500 legitimate emails and 529 scam emails. Because the data set was not balanced, the accuracy got worse as the number of neighbors was raised, and 1-NN achieved the best results. (Saberi, Vahidi, and Bidgoli 2007)

kNN has also performed well in detecting anomalies. The performance of kNN was tested using a data set consisting of normal traffic and four types of different malicious traffic: probing, denial of service, remote to local and user to root. There were 494 020 samples consisting of 41 features. kNN had the worst performance when only six features were used, but increasing the number of features to 19 made kNN perform the best achieving 99.89% accuracy. The high computational complexity of kNN can be seen from the training time. The time needed for training increased from 46 hours to 71 hours when the number of features were increased, while the second best parameter required only 27 hours and 30 hours respectively. (Lin, Ke, and Tsai 2015)

3.6.3 Trees

Decision trees use a tree-like graph of where the leaves represent decisions and classifications, and branches represent the conjunction of features that lead to those classifications (Mena 2016; Dua and Du 2016). An example of a decision tree is shown in Figure 6.

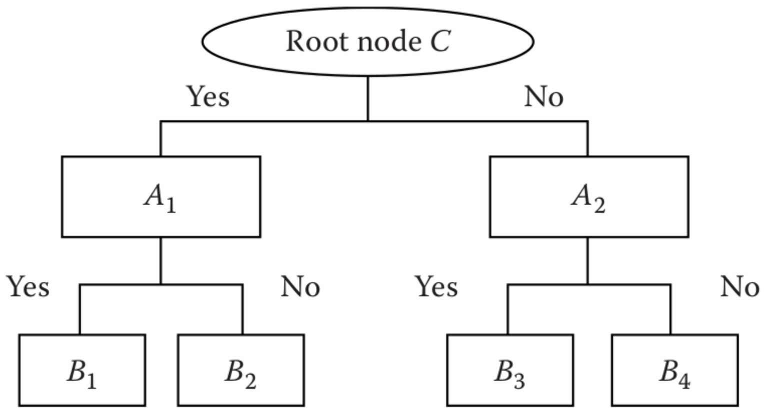


Figure 6: Example of a decision tree (Dua and Du 2016)

When doing classification using a decision tree, the traversing starts from the top of the tree and ends at one of the leaves, which represents one class. Each node computes an inequality, which may depend on one or more of the input variables depending if it is a binary or linear decision tree. (Dua and Du 2016)

Decision trees can handle missing values and noisy data and are able to solve multi-type attribute problems, but can't guarantee the optimal accuracy other classifiers can. Decision trees are not popular on their own, but they are used in other classifiers, such as random forest. This could be explained because seeking smallest decision tree is known to be NP-hard. (Dua and Du 2016)

Random forest improves the predictive accuracy of decision trees (Kononenko and Kukar 2007). It is based on generating a series of decision trees by taking n bootstrap samples from the data set, and for each sample grow an unpruned decision tree. To classify using random forest, the data is ran through each one of the n decision trees, which all give a classification result. Finally, the random forest votes based on the classification results of each tree, i.e., which class got the most votes. (Liaw, Wiener, et al. 2002) Random forest works well with a large number features but can also handle missing features. It also usually doesn't overfit the data, can be trivially parallelized to scale with increasing data and it can handle unbalanced data sets. (Amin, Ryan, and Dorp 2012)

With increasing data, the number of nodes in decision trees will grow exponentially with depth. In some instances the depth needs to be limited to restrict memory usage, but this will limit the potential accuracy. Decision jungles were introduced to increase the efficiency of decision trees by enabling multiple paths from the root to each leaf. Researched showed decision jungles improving memory efficiency while improving generalization in several tasks compared to random forest. (Shotton et al. 2013)

Random forest has been utilized frequently in network intrusion detection

(Resende and Drummond 2018). For example, Shafieian, Zulkernine, and Haque (2015) used random forest to detect slow-read denial of service attacks at the destination. The data set consisted of about 250 000 attack and 250 000 benign TCP packets. An accuracy of 99.37% was resulted. Research found two drawbacks in the method: it is hard to tell how the classification was done and the computational requirements rose as the number of decision trees is increased. (Shafieian, Zulkernine, and Haque 2015)

Random forest classifier also performed well in a study investigating classification accuracies of six classifiers on a phishing email data set. The data set consisted of 44 features and a total of 2889 emails, of which 59.5% were legitimate. Random forest outperformed classifiers such as neural network, logistic regression and support vector machine. (Abu-Nimeh et al. 2007)

3.6.4 Functions

This class of methods includes non-probabilistic classifiers like in lazy classifiers, but the training data is generalized before classifying. These classifiers are based on optimization theory and statistical estimation. (Amancio et al. 2014) For example, in logistic regression, the goal is to find a model that best describes the relationship between the outcome (classification result) and a set of independent variables (features). (Hosmer Jr, Lemeshow, and Sturdivant 2013) An example of a logistic regression graph can be seen in Figure 7.

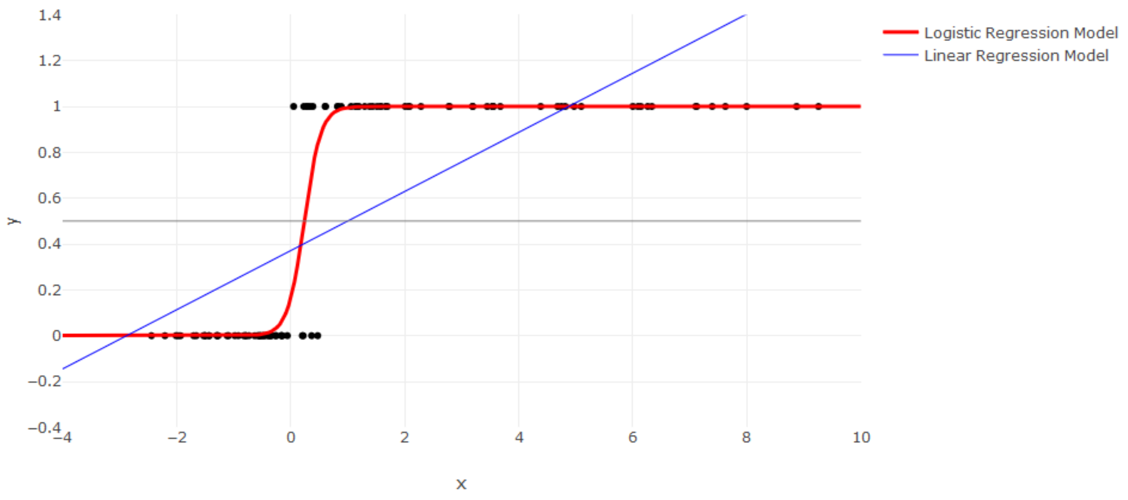


Figure 7: Example of a logistic regression

As can be seen in Figure 7, it is not possible to fit a linear line between the observations to achieve a good fit, so a logistic model is needed. To make classifications with logistic regression, a decision boundary must be defined. For example, in Figure 7, it could be the gray line at $y = 0.5$. All the data points above it would be classified belonging to the same class and everything under it to another class.

Logistic regression was successfully used to detect botnet traffic and was chosen because of it being light-weight, easy to implement and interpret, and not requiring a lot of computing resources. The data set consisted of eight different botnet families and had over 10 000 flows of malicious traffic and the same number of benign traffic. Logistic regression achieved over 95% accuracy, AUC of 98.5% and recall of 96.7%. (Bapat et al. 2018)

Neural networks are an effective alternative for statistical functions like logistic regression, which are affected by the data distribution. Neural network is composed of interconnected neurons, where each neuron is connected to every other neuron in the same layer by weight. These layers are called hidden layers. Together, these hidden layers map the input vector (features) to an output vector (classification result) through non-linear information processing. During training the weights of the network are adjusted to match the desired output by calculating an error calculated based on the difference of actual and desired output. (Gardner and Dorling 1998; Dua and Du 2016) An example of an artificial network can be seen in Figure 8. It is called a multilayer perceptron as it holds multiple hidden layers.

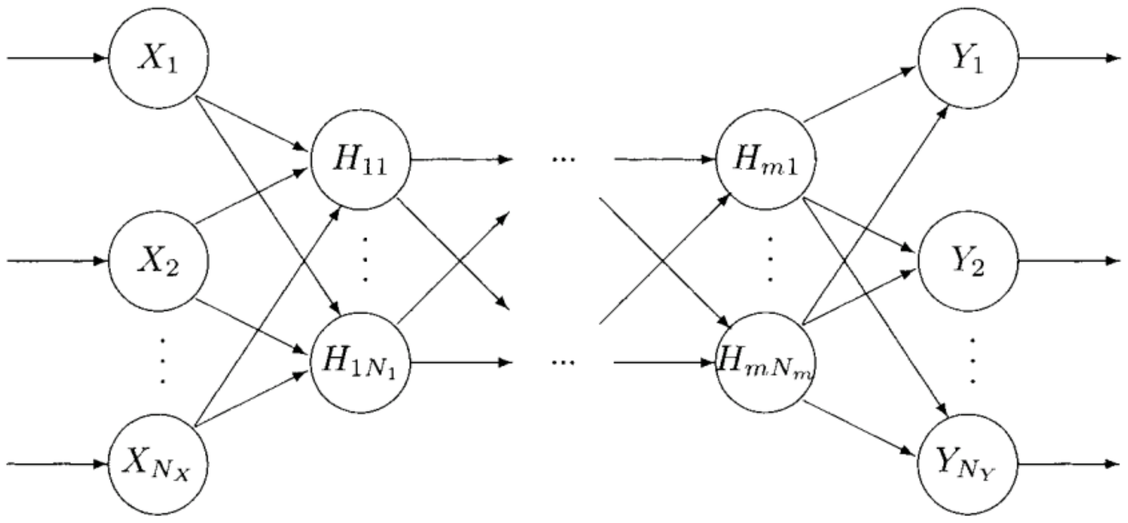


Figure 8: Example of a multilayer perceptron. It is composed of input neurons $X_1..X_N$, N number of neurons H in m hidden layers, and output neurons $Y_1..Y_N$ (Kononenko and Kukar 2007)

There exists many variations of neural networks apart from multi-layer perceptrons. A Long Short Term Memory (LSTM) recurrent neural network was used for intrusion detection and the performance was compared against six other classifiers. The long short term memory recurrent neural network performed the best with 96.63% accuracy. (Kim et al. 2016) In other research the same neural network was used to classify malware and its performance was compared against seven other classifiers. LSTM achieved a 94% accuracy using op codes as features from a data set of 280 malware and 271 benign files. (HaddadPajouh et al. 2018)

Along logistic regression and neural networks, Support Vector Machines (SVM) have had success in solving both regression and classification problems. Unlike other classifiers, SVMs don't try to minimize the number of used attributes, but use all the available features to create a higher dimensional input space, and the objective is to create an optimal separating hyperplane into the space. (Kononenko and Kukar 2007; Suykens and Vandewalle 1999). The optimal hyperplane has the highest distance to the nearest data points on each side. Figure 9 shows an example of a support vector machine. Figure 9a shows the hyperlane separating the two classes and Figure 9b shows the support vectors, which are the closest data points to the hyperplane. (Dua and Du 2016)

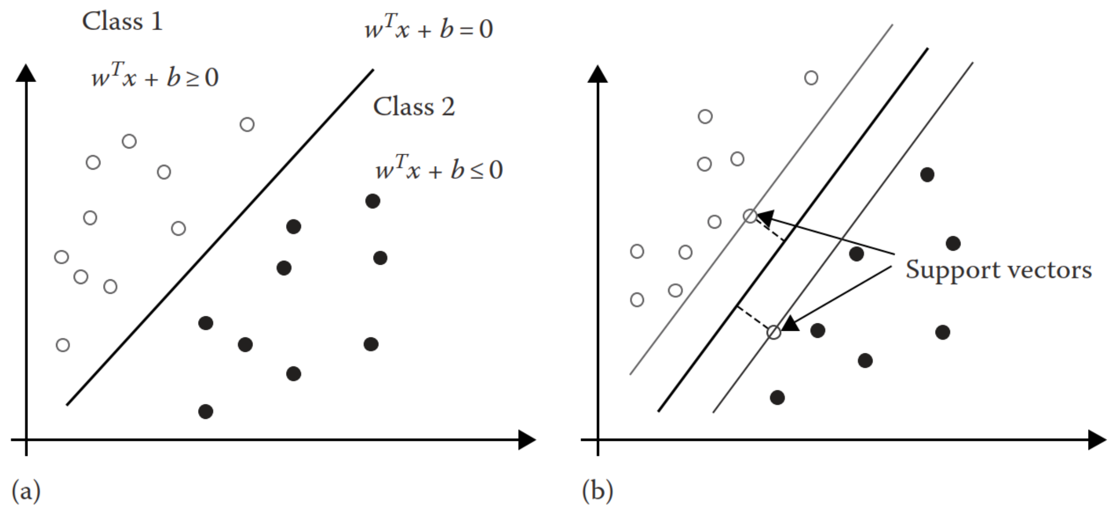


Figure 9: Example of SVM classifier (Dua and Du 2016)

SVM performs better in controlling the overfitting problem than neural nets and can perform well even with small training sets. (Dua and Du 2016) It has also said to have good generalization performance and learning ability with high dimensional or noisy data (Enache and Sgarciu 2014). SVM can also scale better than neural networks as the amount of samples or features grow: it can handle a large number of features which would take a long time for neural network to train. (Mukkamala, Sung, and Abraham 2005)

In a recent study, SVM was chosen to classify Android malware, because of its guaranteed performance in high-dimensional data, fast convergence rate and good generalization performance. It performed well in practice, achieving an accuracy of 94.15%. The performance was evaluated against DroidRisk, which classifies Android applications based on applications permission requests. Using DroidRisk, only an accuracy of 85.63% was achieved. (Sikos 2018)

4 Methodology

This chapter introduces the methods used to evaluate and compare classifier performance as well as how the data set was obtained.

4.1 Choosing the research methodology

The object of the practical part of this thesis is to evaluate and compare classifier performance on different situations. In previous research, classifiers have been trained using a data set and classifier performance has been evaluated using different performance measures (Clemens 2015; De Nicolao et al. 2018). To help with choosing the methodology for this thesis, the graph depicted in Figure 10 was used. It gives many different choices for methodology and it represents the focus different methods have: the closer to the circle's center a method is, the broader the focus is. The arrows show the path that was taken for this thesis to go from a broad method to a narrow method that can actually be used to conduct research.

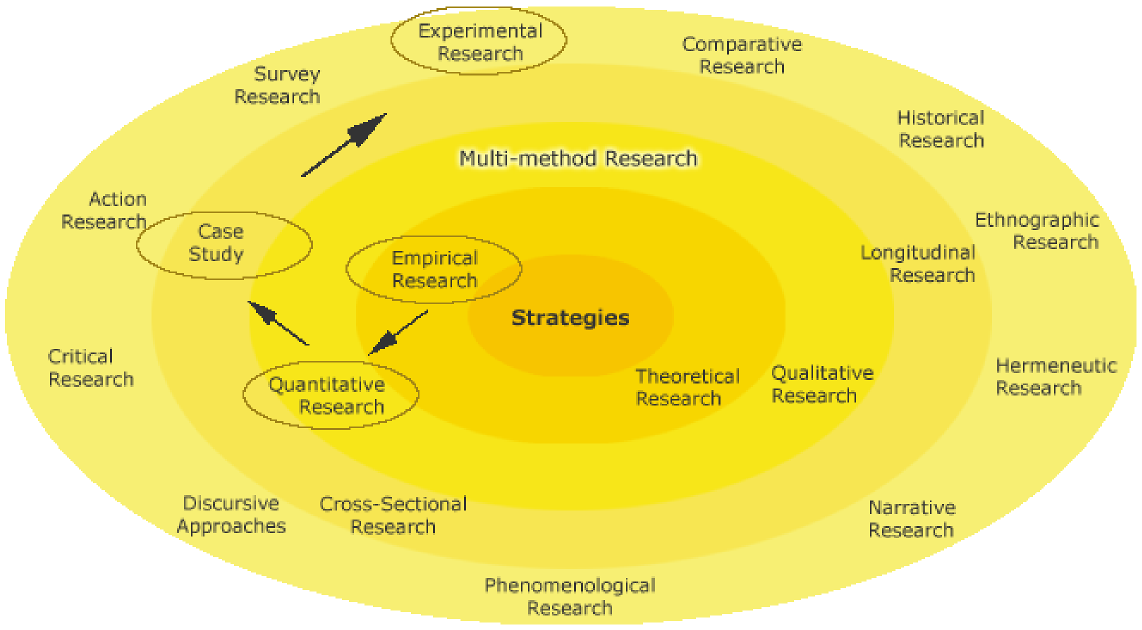


Figure 10: Steps to choose a research methodology (University of Jyväskylä 2010)

In the broad context, this thesis falls under empirical research, as the research is not only based on theory and ideas that are purely analytical, but is based on evidence from the real world (Balnaves and Caputi 2001). Next, because we are interested in comparing and displaying the classifier performance using data in numerical instead of narrative form, quantitative research is the logical option

(Given 2008).

This thesis could be a case study, as the intention is to study the case in depth, with only a portion of the population. The case could be a binary file, but as the classifiers need a lot of samples for training, there might be too many cases for a case study. The boundary for the number of cases is not clear, but some research has suggested an upper limit of 10 to 60 cases. (Given 2008) In this thesis the number of cases would be in tens of thousands.

Exploring the research methods in the outer ring of Figure 10, experimental research was found the most fitting. It composes of things to be measured (in this case, classifiers), a hypothesis to be tested (which classifier performs the best) and a systematic analysis of the data (evaluation of classifier performance) (Denning 1980). The following sections will cover how the experiment will be executed.

4.2 Experimental environment

The complete feature set consists of 293 features: the byte frequencies (256 features), endianness signatures made by Clemens (2015) (4), 31 function epilog and prolog signatures from angr framework Shoshitaishvili et al. (2016) for amd64, arm, armel, mips32, powerpc, powerpc64, s390x and x86 that De Nicolao et al. (2018) also used, and two signatures for powerpcspe made for this thesis. These features are extracted from only the code sections of the binary, and saved to a CSV file. The full binary is not used, because data sections can alter the byte frequencies or include byte sequences that match a signature for another architecture, which can lead into false classifications (De Nicolao et al. 2018). The possible negative effect of using full binary for training is tested as a part of this thesis.

The acquired data set consists of 23 architectures and about 105 000 executables. Hurd-i386 architecture was left out, as it uses i386 architecture and is considered a port because of the different kernel it utilizes. AVR and Cuda samples were left out as well as there was no source for sufficiently reliable and representative set of binary files for the architectures. Not every Debian version available for every architecture was downloaded due to resource constraints. Also, only about 67 000 executables were used to train and test the classifier due to resource constraints. The summarized details of the data set are presented in Tables 2 and 3. The data set was downloaded to a server provided by the University of Jyväskylä[1] because of the amount of disk space required. The

1. We acknowledge grants of computer capacity from the Finnish Grid and Cloud Infrastructure (persistent identifier urn:nbn:fi:research-infras-2016072533)

server had a four core CPU (Intel(R) Xeon(R) CPU E7- 8837 2.67GHz) and 16 GB of DDR3 and was running CentOS 7.4 (kernel 3.10.0-957.5.1.el7.x86_64).

Table 2: Cumulative data set statistics

Type	# of files	Total size
.iso	1 600	1.8 TB
.deb	79 000	36 GB
executable	105 000	28 GB
code section	105 000	17 GB

Table 3: Data set statistics

Architecture	# of samples	Wordsize	Endianness	File info
alpha	3000	64	Little	ELF 64-bit LSB executable, Alpha (unofficial)
amd64	2994	64	Little	ELF 64-bit LSB executable, x86-64
arm64	2997	64	Little	ELF 64-bit LSB executable, ARM aarch64
armel	2994	32	Little	ELF 32-bit LSB executable, ARM
armhf	2994	32	Little	ELF 32-bit LSB executable, ARM
hppa	3000	32	Big	ELF 32-bit MSB executable, PA-RISC (LP64)
i386	2994	32	Little	ELF 32-bit LSB executable, Intel 80386
ia64	3000	64	Little	ELF 64-bit LSB executable, IA-64
m68k	3000	32	Big	ELF 32-bit MSB executable, Motorola 68020
mips	2997	32	Big	ELF 32-bit MSB executable, MIPS, MIPS-II
mips64el	2998	64	Little	ELF 64-bit LSB executable, MIPS, MIPS64 rel2
mipsel	2994	32	Little	ELF 32-bit LSB executable, MIPS, MIPS-II
powerpc	2110	32	Big	ELF 32-bit MSB executable, PowerPC or cisco 4500
powerpcspe	3000	32	Big	ELF 32-bit MSB executable, PowerPC or cisco 4500
ppc64	2552	64	Big	ELF 64-bit MSB executable, 64-bit PowerPC or cisco 7500
ppc64el	2997	64	Little	ELF 64-bit LSB executable, 64-bit PowerPC or cisco 7500
riscv64	3000	64	Little	ELF 64-bit LSB executable
s390	2997	32	Big	ELF 32-bit MSB executable, IBM S/390
s390x	2994	64	Big	ELF 64-bit MSB executable, IBM S/390
sh4	3000	32	Little	ELF 32-bit LSB executable, Renesas SH
sparc	2997	32	Big	ELF 32-bit MSB executable, SPARC32PLUS, V8+ Required
sparc64	2676	64	Big	ELF 64-bit MSB executable, SPARC V9, relaxed memory ordering
x32	3000	32	Little	ELF 32-bit LSB executable, x86-64

The number of samples used is higher than Clemens (2015) and De Nicolao et al. (2018) used and is more evenly distributed. Therefore it should give more accurate results when evaluating classifier performance, as imbalance of classes in the data set can cause suboptimal classification performance (Japkowicz 2003). Also, only code sections over 4000 bytes were selected to eliminate the negative effect of too small sample, even though Clemens (2015) opted to not use this method. He acknowledged it would avoid many of the features having close to zero matches, but argued it is a more realistic scenario in the field. Regardless, only large enough samples were used to train the classifiers in this thesis as there was enough data to do so and because as seen in Clemens (2015) results, all classifiers are near 90% accuracy at that point, so it removes one random variable when evaluating the classifiers.

To validate Clemens (2015) results, Weka framework was used with only the architectures and features used by the author. The list of classifiers and their settings can be seen in Table 4.

Table 4: Classifiers and settings used by Clemens (2015)

Model	Weka name	Parameters
1 nearest neighbor (1-NN)	IBk	-K 1 -W 0 -A "weka.core.neighboursearch.-LinearNNSearch -A "weka.core.-EuclideanDistance -R first-last""
3 Nearest neighbors (3-NN)	IBk	-K 3 -W 0 -A "weka.core.neighboursearch.-LinearNNSearch -A "weka.core.-EuclideanDistance -R first-last""
Decision tree	J48	-C 0.25 -M 2
Random tree	RandomTree	-K 0 -M 1.0 -V 0.001 -S 1
Random forest	RandomForest	-I 100 -K 0 -S 1 -num-slots 1
Naive Bayes	NaiveBayes	N/A
BayesNet	Bayesnet	-D -Q weka.classifiers.bayes.net.-search.local.K2 – -P 1 -S BAYES -E weka.classifiers.bayes.net.-estimate.SimpleEstimator – -A 0.5
SVM (SMO)	SMO	-C 1.0 -L 0.001 -P 1.0E-12 -N 0 -V -1 -W 1 -K "weka.classifiers.functions.-supportVector.PolyKernel -E 1.0 -C 250007"
Logistic regression	SimpleLogistic	-I 0 -M 500 -H 50 -W 0.0
Neural net	MultilayerPerceptron	-L 0.3 -M 0.2 -N 100 -V 0 -S 0 -E 20 -H 66

Default values were used for most of the classifiers in Weka, as our experiment's iterations have shown the tweaking of the parameters did not have a significant effect on most of the classifiers (Amancio et al. 2014). Different parameters were used for neural net when training the classifier on the complete data set of this thesis, because the parameters used by Clemens (2015) were specific to his data set. The parameters used to train the neural network for this data set were "-L 0.3 -M 0.2 -N 100 -V 20 -S 0 -E 20 -H 66 -C -I -num-decimal-places 10", which were acquired by manual parameter tuning.

To validate De Nicolao et al. (2018) results, scikit-learn by Pedregosa et al. (2011) was used. De Nicolao et al. (2018) used only logistic regression classifier but compared to the Clemens (2015) logistic regression, they added L1 regularization. In this thesis the logistic regression classifier will be implemented in both scikit-learn and Keras, a high-level neural networks API by Chollet et al. (2015) to see if the used framework has an effect on the classification accuracy. The version used for Keras was 2.2.4, scikit-learn version 0.20.0 and Weka version 3.8.3. Both scikit-learn and Keras use Tensorflow as the back end. The machine used to train the classifiers is a standard computer with a six-core Intel(R) Core(TM) i7-8700k 5.00GHz CPU with 12 threads, GTX 970 graphics card, 16 GB of DDR4 and a Windows 10 operating system.

In addition, Azure Machine Learning was used to test the classification performance on full binaries. It provides several classifiers for both multi-class classification as well as binary classifiers, which can be used in multi-class cases with "One-vs-All" plugin provided by Azure. The classifiers will be chosen based on what classifiers will perform the best on other frameworks. An example of the workflow in Azure is presented in Figure 11. In the example, the classifier is trained on all of the training data and tested against a separate test set containing 500 full binaries for each architecture.

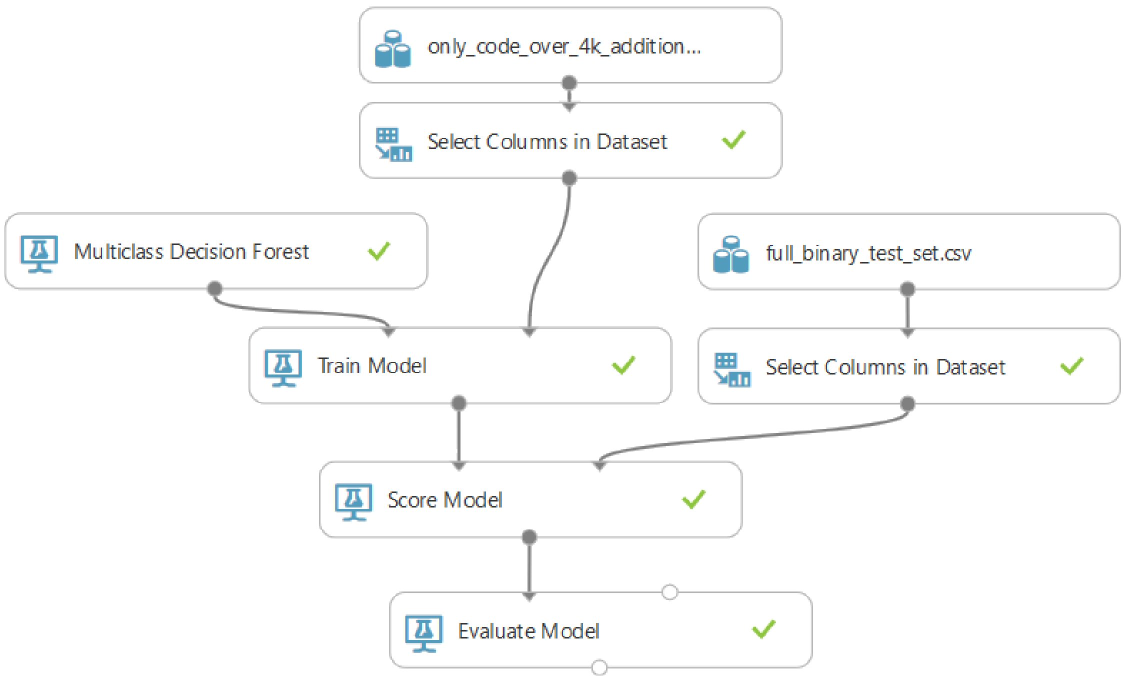

only_code_over_4k_addition...
Select Columns in Dataset
Multiclass Decision Forest
full_binary_test_set.csv
Train Model
Select Columns in Dataset
Score Model
Evaluate Model

Figure 11: Classifier and dataflow setup used in Azure Machine Learning

In this thesis, the performance of classifiers will be evaluated in different environments. First, classifiers' performance is compared with 10-fold cross validation using only the features extracted from code sections. Also, the effect of features on classification performance is evaluated by calculating performance measures with the whole feature set, BFD and endianness signatures and BFD only. These results are compared to the ones presented by Clemens (2015).

Next, to see if the sample size has an effect, the classifiers will be tested against a test set of code sections with varying size as done by Clemens (2015) and De Nicolao et al. (2018). If the performance is good with only the fragments of the executables, same classifiers could be used in environments where only a part of the executable is present, such as forensics. The code fragments were taken with random sampling to remove any bias that could come from using only the beginning of the code sections (Clemens 2015). On the other hand, this will decrease reliability as acquiring exactly the same data is unlikely to happen.

To see how well classifiers perform classifying full binaries, the classifiers trained with only code sections of binaries are tested against a separate test set consisting of 500 full binaries per architecture. The addition of data sections in the full binaries will have an effect on the byte frequency distribution, but according to De Nicolao et al. (2018), it did not have an effect on classification accuracy. In real life case, it might not always be possible to extract code sections from binaries, so this could reflect on how well the classifiers would perform in real life situations. Finally, the classifiers are trained using full binaries to see if they perform better or worse compared to classifiers trained with only code sections when classifying full binaries.

5 Results

This chapter goes through the results of the experimental evaluation. Classifier performance is tested in different situations, including training and testing with only code sections of the binaries, testing with different sample sizes as well as testing performance of classifying full binaries. Also, classifiers are trained and tested with full binaries to see if it has an effect on classifier performance.

5.1 Creation of new signatures

During the testing of classifier performance, few architectures caused the classifiers the most false matches. For example, the binary code for powerpc an powerpcspe are essentially the same, the only difference being the presence of SPE instructions in powerpcspe. Therefore, custom signatures had to be created to be able to accurately distinguish between the powerpc and powerpcspe architectures. The signatures were created by comparing the instructions between the two architectures and finding unique values that only appear in one of the architectures. The two signatures composed of many floating point and integer operations that appeared only in powerpcspe architecture. The many different floating point operations were so similar in byte representation, that it was possible to create a single signature for them. The same applied to the integer operations. The results can be seen in Table 5.

Table 5: F1 score with random forest trained and tested with full binaries in Weka with and without powerpcspe signatures

Architecture	F1 score		
	Without signature	With	Improvement
powerpc	0.868	0.894	2.99%
powerpcspe	0.888	0.906	2.02%

The two additional powerpcspe signatures increased the F1 score of random forest implemented in Weka by about two percentage units, which corresponds to about two to three percentages performance increase. The data set consisted of thousand full binaries for each architecture for training and the same amount for testing. The confusion matrix with only powerpc and powerpcspe architectures without the additional signatures can be seen in Figure 5.1 and confusion matrix with the additional signatures can be seen in 5.1.

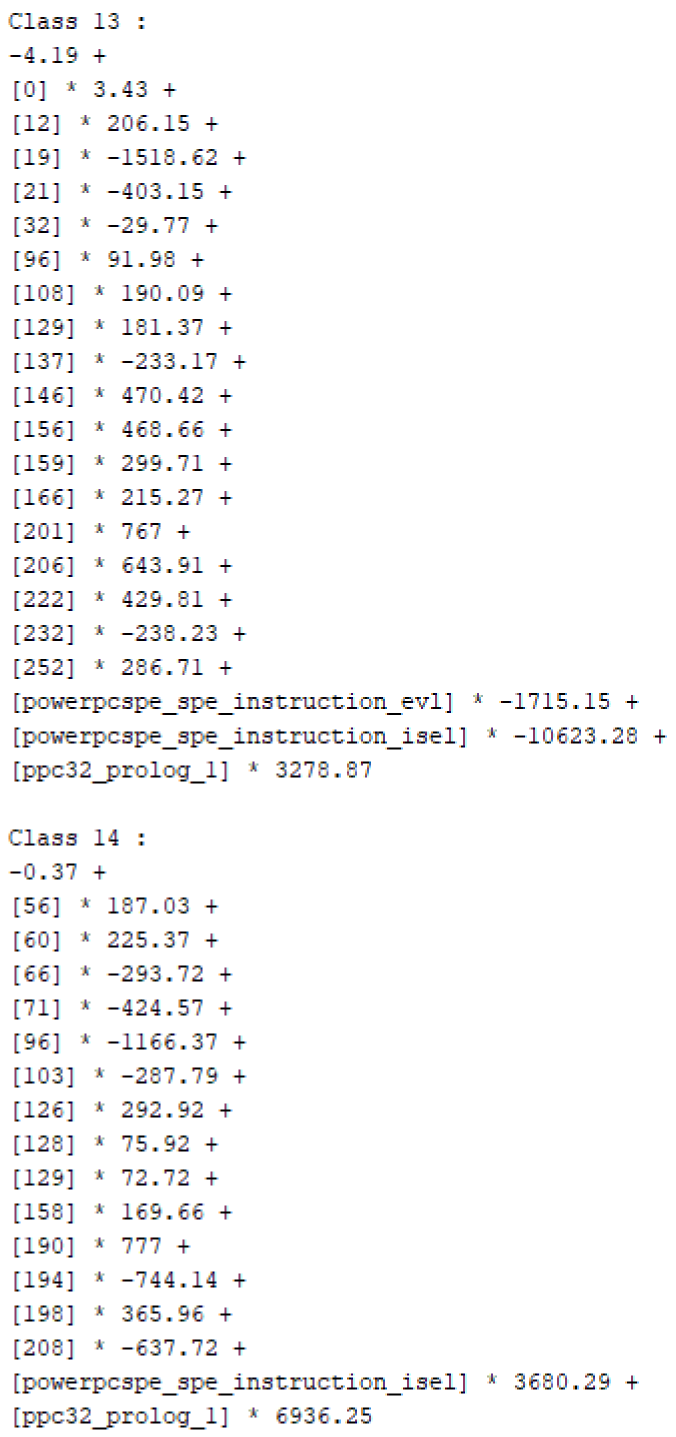
Table 6: Confusion matrix of powerpc and powerpcspe architectures without the additional signatures

	Predicted: powerpc	Predicted: powerpcspe
Actual: powerpc	802	200
Actual: powerpcspe	39	963

Table 7: Confusion matrix of powerpc and powerpcspe architectures with the additional signatures

	Predicted: powerpc	Predicted: powerpcspe
Actual: powerpc	843	160
Actual: powerpcspe	36	965

The additional signature helped to to correctly classify about 40 more powerpc samples. The effect of the signatures can be also seen by running the same data set on a logistic regression classifier and inspecting the coefficients for powerpc and powerpcspe. They are presented in Figure 12.



```
Class 13 :
-4.19 +
[0] * 3.43 +
[12] * 206.15 +
[19] * -1518.62 +
[21] * -403.15 +
[32] * -29.77 +
[96] * 91.98 +
[108] * 190.09 +
[129] * 181.37 +
[137] * -233.17 +
[146] * 470.42 +
[156] * 468.66 +
[159] * 299.71 +
[166] * 215.27 +
[201] * 767 +
[206] * 643.91 +
[222] * 429.81 +
[232] * -238.23 +
[252] * 286.71 +
[powerpcspe_spe_instruction_evl] * -1715.15 +
[powerpcspe_spe_instruction_isel] * -10623.28 +
[ppc32_prolog_l] * 3278.87

Class 14 :
-0.37 +
[56] * 187.03 +
[60] * 225.37 +
[66] * -293.72 +
[71] * -424.57 +
[96] * -1166.37 +
[103] * -287.79 +
[126] * 292.92 +
[128] * 75.92 +
[129] * 72.72 +
[158] * 169.66 +
[190] * 777 +
[194] * -744.14 +
[198] * 365.96 +
[208] * -637.72 +
[powerpcspe_spe_instruction_isel] * 3680.29 +
[ppc32_prolog_l] * 6936.25
```

Figure 12: Coefficients in logistic regression when using the additional powerpcspe signatures. Class 13 is powerpc while class 14 is powerpcspe.

The negative coefficients in powerpc (class 13) means the relationship between the class and the feature is negative, i.e., if the two signatures are found from a binary, it is less likely to be a powerpc binary. Looking at class 14 representing powerpcspe, one of the signatures correlates positively on it, meaning that if the signature is found from a binary, it is more likely to be a powerpcspe binary.

5.2 Classifier performance using code sections

Different classifiers were trained and tested using different feature sets in Weka using the settings presented in Table 4. BFD corresponds to using only byte frequency distribution, while BFD+endian adds the endian fingerprints introduced by Clemens (2015). The complete data set includes the new architectures as well as the new signatures for powerpcspe. The performance metrics are weighted averages, i.e., sum of the metric through all the classes, weighted by the number of instances in the specific class. The results are compared to the results presented by Clemens (2015). The results can be seen in Table 8. The results obtained with different parameters than Clemens (2015) used are marked with an asterisk.

Table 8: Classifier performance with different feature sets, also compared with some metrics presented by Clemens (2015)

Classifier	Precision	Recall	AUC	F1 measure	Accuracy				
					Complete	BFD+endian	BFD+endian Clemens (2015)	BFD	BFD Clemens (2015)
1-NN	0.983	0.983	0.991	0.983	0.983	0.911	0.927	0.895	0.893
3-NN	0.994	0.994	0.999	0.994	0.993	0.957	0.949	0.902	0.898
Decision tree	0.992	0.992	0.998	0.992	0.992	0.993	0.980	0.936	0.932
Random tree	0.966	0.966	0.982	0.966	0.965	0.953	0.929	0.899	0.878
Random forest	0.996	0.996	1.000	0.996	0.996	0.992	0.964	0.904	0.904
Naive Bayes	0.991	0.991	0.999	0.991	0.990	0.990	0.958	0.932	0.925
BayesNet	0.992	0.992	1.000	0.992	0.991	0.994	0.922	0.917	0.895
SVM (SMO)	0.997	0.997	1.000	0.997	0.997	0.997	0.983	0.931	0.927
Logistic regression	0.989	0.988	0.998	0.989	0.988	0.997	0.979	0.939	0.930
Neural net	0.995*	0.994*	1.000*	0.994*	0.994*	0.919	0.979	0.940	0.940

The results are inline with the results presented by Clemens (2015) even with the different data set. The largest differences to the results presented by Clemens (2015) were BayesNet's sensibly higher accuracy of 99.4% compared to the accuracy reported by Clemens (2015) of 92.2% and neural net's sensibly lower accuracy of 91.9% compared to 97.9% when using both BFD and endianness features. However, in our experiments the complete data set with added architectures and signature increased the accuracy of all classifiers when compared to the results presented by Clemens (2015).

5.3 Effect of test sample size on classification performance

To see the effect of the size of the classified binary, samples of code sections of varying sizes were taken and the classifiers trained using the whole code sections were tested against them. Small (128 bytes or less) sample sizes could be encountered in forensics, where, for example, only a portion of malware code is successfully extracted from a smartphone or an IoT device while bigger samples of thousands of bytes could already be full binaries encountered in IoT device firmware images. The results can be seen in Figure 13

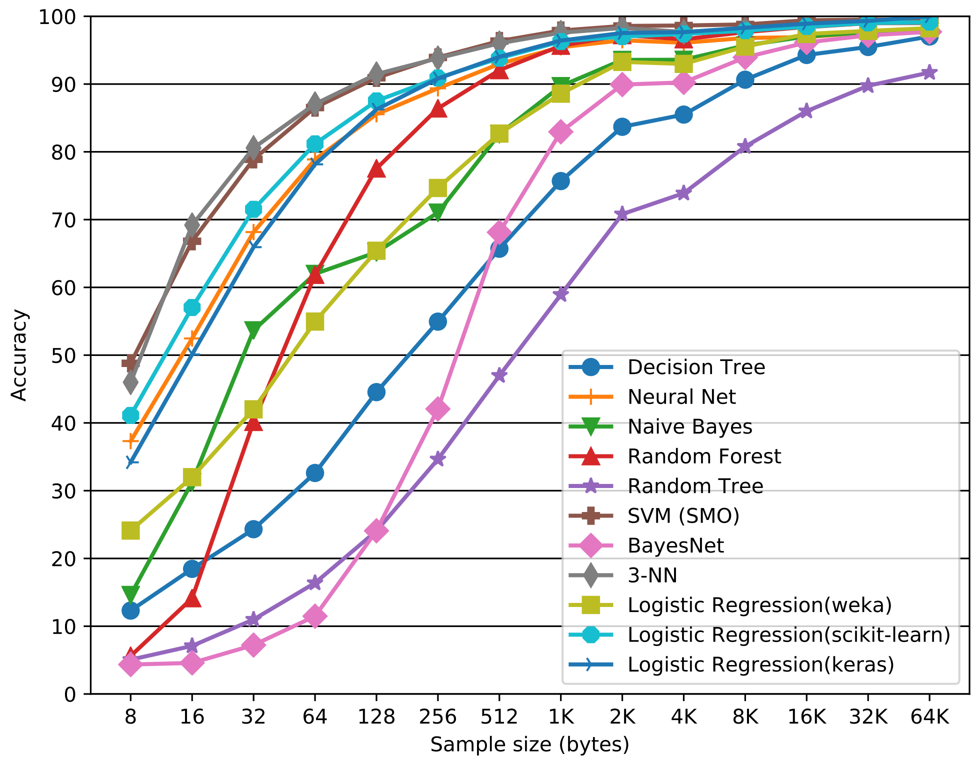


Figure 13: Impact of the sample size on classifier performance

SVM performed the best with almost 50% accuracy even with the smallest sample size of 8 bytes and achieving 90% accuracy at 128 bytes along with 3 nearest neighbors. Logistic regression implemented in scikit-learn and Keras were close in performance achieving 90% accuracy at 256 bytes. Logistic regression implemented in Weka falls off compared to them requiring 2048 bytes to reach 90% accuracy. Compared to the results presented by Clemens (2015), the classifiers that performed the best in the experiments of this thesis also performed well in Clemens (2015) research. Also, in our experiment not all the classifiers achieved 90% accuracy at 4000 bytes as Clemens (2015) measured, with decision tree and random tree achieving 85% and 75% accuracies at 4000 bytes.

5.4 Logistic regression on different frameworks

De Nicolao et al. (2018) chose logistic regression from all the different classifiers available. It has couple of parameters that affect the classification performance. De Nicolao et al. (2018) used grid search to identify the best value for C, which stands for inverse of regularization strength and found 10000 to give the best results. The data set affects the results, so grid search was ran on the scikit-learn model on the data set introduced for this thesis. C values of 10000, 1000, 100, 10, 1, 0.1 were tested and 1000 gave the best results. Using Keras, C value of 0.0000001 provided the best accuracy. Table 9 shows the results of logistic regression 10-fold cross-validation on code sections on all three different platforms.

Table 9: Logistic regression 10-fold cross validation performance on different platforms

Classifier	Precision	Recall	AUC	F1 measure	Accuracy
Weka	0.989	0.988	0.998	0.989	0.988
scikit-learn	0.998	0.998	0.998	0.998	0.996
Keras	0.998	0.998	0.998	0.998	0.997

Logistic regression implemented in scikit-learn and Keras give better results compared to Weka with F1 measures of 0.998 compared to 0.989 in Weka. Scikit-learn and Keras give similar results except Keras giving 99.7% accuracy compared to scikit-learn's 99.6%.

5.5 Classifier performance on full binaries

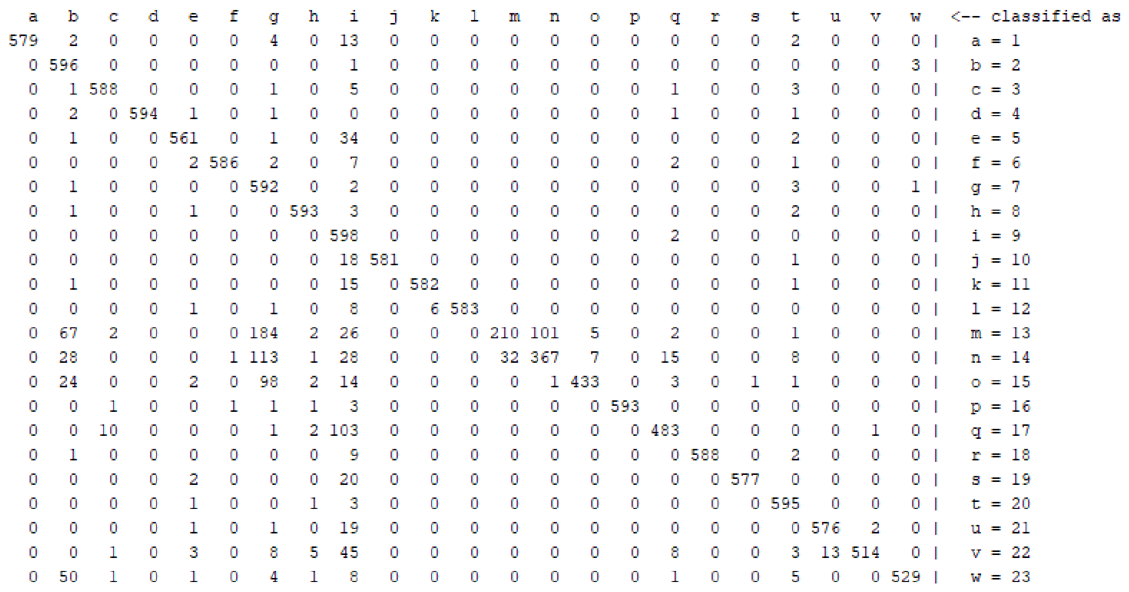
De Nicolao et al. (2018) tested their classifier performance on full binaries with both code and data sections. In this thesis all the different classifiers used by Clemens (2015) and De Nicolao et al. (2018) will be tested against full binaries using a separate test set consisting of 500 binaries for each architecture. The classifiers are still trained using only the code sections, while in Section 5.6 the classifiers are also trained using the full binaries. The results can be seen in Table 10.

Table 10: Classifier performance on full binaries

Classifier	Precision	Recall	AUC	F1 measure	Accuracy
1-NN	0.871	0.742	0.867	0.772	0.741
3-NN	0.876	0.749	0.892	0.773	0.749
Decision tree	0.845	0.717	0.865	0.733	0.716
Random tree	0.679	0.613	0.798	0.619	0.613
Random forest	0.912	0.902	0.995	0.892	0.901
Naive Bayes	0.807	0.420	0.727	0.419	0.420
Bayes net	0.886	0.844	0.987	0.840	0.844
SVM (SMO)	0.883	0.733	0.971	0.766	0.732
Logistic regression (Weka)	0.875	0.718	0.978	0.728	0.718
Logistic regression (scikit)	0.913	0.780	0.780	0.794	0.579
Logistic regression (Keras)	0.921	0.831	0.831	0.839	0.676
Neural net	0.841	0.452	0.875	0.515	0.451
(De Nicolao et al. 2018)	0.996	0.996	0.998	0.996	-

Random forest performed the best having the highest performance measures achieving 0.901 accuracy and 0.995 AUC. The logistic regression implemented in scikit-learn did not perform as well as evaluated by De Nicolao et al. (2018) with reported averages of 0.99 in all performance measures except accuracy that was not presented. The time to classify all the binaries in the test set took only a couple of seconds on all algorithms except the Nearest Neighbor algorithms, which took approximately 15 minutes. This is because Nearest Neighbor is a lazy classifier and the model is only built when data needs to be classified, as explained in Section 3.6.2.

Figure 14 shows the confusion matrix of the best performing classifier, random forest. The columns represent the class frequencies predicted by the model while the rows present the true class frequencies. Everything off from the diagonal is a misclassification. The alphabets represent the 23 classes in the alphabetical order seen in Table 3. Looking at the confusion matrix, i386 (g) and m68k (i) caused over 70% of the misclassifications.



```
   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o   p   q   r   s   t   u   v   w   <-- classified as
 579   2   0   0   0   0   4   0  13   0   0   0   0   0   0   0   0   0   0   2   0   0   0 |   a = 1
   0 596   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   3 |   b = 2
   0   1 588   0   0   0   1   0   5   0   0   0   0   0   0   0   1   0   0   3   0   0   0 |   c = 3
   0   2   0 594   1   0   1   0   0   0   0   0   0   0   0   0   1   0   0   1   0   0   0 |   d = 4
   0   1   0   0 561   0   1   0  34   0   0   0   0   0   0   0   0   0   0   2   0   0   0 |   e = 5
   0   0   0   0   2 586   2   0   7   0   0   0   0   0   0   0   2   0   0   1   0   0   0 |   f = 6
   0   1   0   0   0   0 592   0   2   0   0   0   0   0   0   0   0   0   0   3   0   0   1 |   g = 7
   0   1   0   0   1   0   0 593   3   0   0   0   0   0   0   0   0   0   0   2   0   0   0 |   h = 8
   0   0   0   0   0   0   0   0 598   0   0   0   0   0   0   0   2   0   0   0   0   0   0 |   i = 9
   0   0   0   0   0   0   0   0  18 581   0   0   0   0   0   0   0   0   0   1   0   0   0 |   j = 10
   0   1   0   0   0   0   0   0  15   0 582   0   0   0   0   0   0   0   0   1   0   0   0 |   k = 11
   0   0   0   0   1   0   1   0   8   0   6 583   0   0   0   0   0   0   0   0   0   0   0 |   l = 12
   0  67   2   0   0   0 184   2  26   0   0   0 210 101   5   0   2   0   0   1   0   0   0 |   m = 13
   0  28   0   0   0   1 113   1  28   0   0   0  32 367   7   0  15   0   0   8   0   0   0 |   n = 14
   0  24   0   0   2   0  98   2  14   0   0   0   0   1 433   0   3   0   1   1   0   0   0 |   o = 15
   0   0   1   0   0   1   1   1   3   0   0   0   0   0   0 593   0   0   0   0   0   0   0 |   p = 16
   0   0  10   0   0   0   1   2 103   0   0   0   0   0   0   0 483   0   0   0   0   1   0 |   q = 17
   0   1   0   0   0   0   0   0   9   0   0   0   0   0   0   0   0 588   0   2   0   0   0 |   r = 18
   0   0   0   0   2   0   0   0  20   0   0   0   0   0   0   0   0   0 577   0   0   0   0 |   s = 19
   0   0   0   0   1   0   0   1   3   0   0   0   0   0   0   0   0   0   0 595   0   0   0 |   t = 20
   0   0   0   0   1   0   1   0  19   0   0   0   0   0   0   0   0   0   0   0 576   2   0 |   u = 21
   0   0   1   0   3   0   8   5  45   0   0   0   0   0   0   0   8   0   0   3  13 514   0 |   v = 22
   0  50   1   0   1   0   4   1   8   0   0   0   0   0   0   0   1   0   0   5   0   0 529 |   w = 23
```

Figure 14: Confusion matrix of random forest classification results on full binaries. Columns show the predicted class frequencies while rows show the true class frequencies.

To test if the classifier implementation has an effect, random forest was also trained in Azure Machine Learning platform. The used parameters were "Number of decision trees: 100, Maximum depth of the decision trees: 32, Number of random splits per node: 128, Minimum number of samples per leaf node: 1". The classifier was trained using the whole data set, and tested against the full binary test set. Also, decision jungle was tested as it is closely related to random forest as explained in Section 3.6.3. The parameters used to train decision jungle were: "Number of decision DAGs: 100, Maximum depth of the decision DAGs: 32, Maximum width of the decision DAGs: 128, Number of optimization steps per decision DAG layer: 2048". All the parameters were empirically tested and selected. Increasing the number of decision trees or DAGs increased the accuracy to some point. Other research has seen same kind of behavior with random forest (Shafieian, Zulkernine, and Haque 2015). The results can be seen in Table 11. Results from Weka were added to make it easier to compare the results.

Table 11: Accuracy, precision and recall for random forest and decision jungle classifiers trained in Azure Machine Learning platform. Performance results from random forest implemented in Weka added for comparison.

Classifier	Accuracy	Precision	Recall
Random forest (Azure)	0.974	0.974	0.974
Random forest (Weka)	0.901	0.912	0.902
Decision jungle (Azure)	0.964	0.964	0.964

Random forest classifier implemented in Azure performed much better compared to the Weka implementation. Random forest in Azure had over 6 percentage units better accuracy, precision and recall than Weka implementation.

5.6 Training classifiers with full binaries

To see how the training of classifiers with full binaries affects the classifier performance on full binaries, some of the best performing classifiers from previous experiments were chosen and trained with a training set consisting of 1000 full binaries for each architecture and tested against a test set of 1000 full binaries for each architecture. The results can be seen in Table 12. The accuracies measured classifying full binaries with classifiers trained with code sections are presented for easier comparison.

Table 12: Classifier performance on full binaries trained with full binaries. Random forest and random jungle missing AUC and F1 measures as Azure platform does not provide them. Accuracies achieved in classifying full binaries with classifiers trained with code sections added for comparison.

Classifier	Precision	Recall	AUC	F1 measure	Accuracy	Accuracies from Table 10 and 11
Random forest (Weka)	0.987	0.987	1.000	0.987	0.986	0.901
Random forest (Azure)	0.992	0.992	-	-	0.992	0.974
Decision jungle (Azure)	0.989	0.989	-	-	0.989	0.964
Logistic regression (Weka)	0.983	0.983	0.999	0.983	0.982	0.718
Logistic regression (scikit)	0.985	0.984	0.984	0.984	0.971	0.579
Logistic regression (Keras)	0.990	0.989	0.989	0.989	0.980	0.676
SVM (SMO)	0.975	0.975	0.997	0.975	0.974	0.732

Classifiers haven't been trained with full binaries in previous research (Clemens 2015; De Nicolao et al. 2018) and the results look very good compared to the performance of classifiers trained with only code sections when classifying full binaries. All the tested classifiers achieved over 97% accuracy with random forest implemented in Azure performed the best with 99.2% accuracy. The accuracy increased compared to the results presented earlier, with logistic regression implemented in scikit-learn achieving over 40 percentage units better accuracy.

6 Discussion

This chapter discusses the results presented in the previous chapters as well as the limitations and possible future research ideas. It also answers the third research question, namely "RQ3: What method for identifying architecture and endianness has the best performance overall and in which cases existing methods outperform the others?".

6.1 Classification performance

When evaluating classifier performance using only the code sections and the complete data and feature sets, all the classifiers achieved over 98% accuracy and no significant differences could be observed from the results. Support vector machine achieved the best accuracy at 99.7% while random tree performed the worst at 96.6% accuracy. Every addition of features increased the accuracy as expected, with the exception of neural net when adding endianness feature. This could be due to the sensitivity of the parameters to the used data, so the same parameters Clemens (2015) used did not work that well with this data set.

When comparing the results presented by Clemens (2015) to the results obtained in this thesis in Table 8, the results are inline, and in most cases the classifiers presented in this thesis outperform the classifiers demonstrated by Clemens (2015). This could be explained by better strategy of data collection, pre-processing, and data set balancing and not having the CUDA and AVR samples. However, this is challenging to verify at this stage because only metadata of the data set used by Clemens (2015) is available. The increase in performance could also be explained by only using binaries over 4000 bytes in size for training, because as shown by Clemens (2015), increasing sample size had a positive effect on classifier performance, and the same can be seen from the results.

Implementing logistic regression in Keras or scikit-learn did not have a considerable difference, but they performed better than logistic regression implemented in Weka, achieving 99.7% and 99.6% accuracies compared to Weka's 98.8%. The better performance could be explained by the addition of L1 regularization in both scikit-learn and Keras.

The architecture identification when using full binaries creates more diversity between the different classifiers. Random forest performed the best achieving 90.1% accuracy, compared to the second best classifier, Bayes net at 84.4%. When comparing the results from logistic regression implemented in Weka and Keras, the difficulty of comparing classifiers can be seen. Keras implementation has higher precision, recall and F1 measure while Weka implementation performs better in AUC and accuracy. To reflect on previous research, De Nicolao et al. (2018) reported over 99% performance measures on average. The accuracies achieved in this thesis did not come close to such high accuracies. The big difference in the results could be explained by the use of a larger test set in this thesis, which comprised of 500 binaries for each architecture, totaling to 11 500 binaries, compared to the dataset of De Nicolao et al. (2018) with a test set of about 3000 binaries for 8 architectures.

When the classifiers were trained with full binaries, the classification performance on full binaries improved dramatically. For example, logistic regression implemented in scikit-learn achieved 97.1% accuracy classifying full binaries compared to the model trained with code sections with only 57.9% accuracy. All the classifiers performed well achieving over 97% accuracy with random forest implemented in Azure achieving the best performance at 99.2%.

Even though good accuracies were achieved classifying full binaries with classifiers trained with full binaries, identifying architecture is still more reliable when classifying only the code sections, but it requires extracting the code sections from the binary. If the header information is missing, it is not straightforward to extract code sections from the binary, but there exists research on how to deal with this issue. For example, the automatic identification of code sections was researched by De Nicolao et al. (2018) and was successful: over 98% accuracy on all 12 architectures tested.

When testing classification accuracy on different sample sizes, SVM, neural net, and the logistic regression implemented in scikit-learn and Keras performed the best achieving 90% accuracy at 256 bytes. Random forest was close getting to 90% accuracy at 512 bytes. The good performance of SVM could have been expected, as said in Section 3.6.4, it only requires a small training sample size. All classifiers achieved 90% accuracy at 2000 bytes except decision tree and random tree, which needed 8000 bytes and 32 000 bytes respectively.

To answer the third research question, namely "RQ3: What method for identifying architecture and endianness has the best performance overall and in which cases existing methods outperform the others?", random forest showed the best performance overall, only losing to SVM when classifying small sample sizes and when classifying binaries using code sections for both training and testing.

Out of the three frameworks used, Weka had to worst documentation, but it had many classifiers implemented and ready to use. Keras and scikit-learn seemed to have a bigger community behind them and they provided extensive documentation.

Using byte frequencies to identify the architecture has its limitations. If a file includes the same code for different architectures, architecture identification using the method presented in this thesis will most likely fail. This could be prevented by using for example sliding windows as done by cpu_rec (Granboulan 2018). Also, to introduce new architectures to the classifiers, there should be a large number of samples with large enough code sections to achieve good accuracies. Producing optimally and automatically large number of binary samples for less popular and known architectures is left as a future work.

6.2 Future work

Future research could include combining the strengths of different classifiers. Seeing that there does not seem to be a single classifier that would perform the best in every situation, multiple classifiers could be used together in a Multiple Classifier System (MCS) (Woźniak, Graña, and Corchado 2014). For example, Shabtai et al. (2009) did research on classification of malicious code and came to the conclusion that different classifiers behave differently with different features, and very high accuracies could be achieved by training multiple classifiers on various types of features. The individual decision of the classifiers would be weighed and combined into the final classification. Also, if it is known whether the binary that is being classified is full binary or if the code sections have been successfully extracted, the classifier choice should be based on this, as this thesis showed that classifiers trained with full binaries perform much better than ones trained with code sections when classifying full binaries.

The current feature set might include irrelevant or redundant features, that might not have any influence on the output (Kotsiantis, I. D. Zaharakis, and P. E. Pintelas 2006). Dimensionality reduction methods could be studied to reduce the number of features and hopefully improving accuracy at the same time. For example, representation learning has been utilized to transform the data to improve accuracy while reducing computational complexity and increasing the processing speed (Suthaharan 2014). Increase of processing speed would help especially in case of nearest neighbor algorithms, where the classification takes significantly longer than with any other classifier tested in this thesis.

The feature set used might also have dependencies between the features, and seeing that many classifiers are based on theory that expects independent variables, it might affect the accuracy negatively. Feature construction or transformation could be used to generate new meaningful features, which could lead to more concise and accurate classifiers, better comprehensibility of the classifiers and better understanding of the learned concept. (Kotsiantis, I. D. Zaharakis, and P. E. Pintelas 2006) Also, new features such as function epilog and prolog signatures for the architectures that don't yet exist in the feature set of this thesis could be introduced, which might improve the classification accuracy.

The toolset is limited to download binaries only from the Debian repository. It could be expanded to include samples of architectures such as AVR and CUDA which would be compiled automatically as a part of the process. It would also allow compilation with different compilers to see how they affect the classification performance. However, this approach has a few problems. The source code should contain enough lines of code so the code sections would not be too small and affect the classifier performance negatively. Source code should also be complex enough so the compiled code would consist of many different opcodes.

7 Conclusion

This thesis studied automatically identifying the architecture and endianness of a binary file. In particular, three research questions were addressed, namely "RQ1: What are the current state of the art methods and tools related to identifying binary code architecture and endianness?", "RQ2: How could existing methods and approaches be improved in terms of algorithms, performance, data sets, and support tools?" and "RQ3: What method for identifying architecture and endianness has the best performance overall and in which cases existing methods outperform the others?".

For the first research question, an in-depth literature review of the state of the art methods was performed and from research field Clemens (2015) and De Nicolao et al. (2018) had the most conclusive research on automatically identifying the architecture and endianness by combining the information of byte frequency analysis and signatures. In practice, tools such as binwalk (ReFirmLabs 2018) and cpu_rec (Granboulan 2018) are used, but they are either too slow or dependent on signatures.

The conducted literature review was also used to answer the second research question. The presented classifiers have performed well, but current approaches lack in the form of support tools. A toolset was developed for this thesis to acquire a data set. It downloads the architectures selected by the user from the Debian repository and preprocesses the binaries to CSV format, ready to be used in frameworks such Weka, scikit-learn or Keras. Also, a more comprehensive review of classifier performance on full binaries with both code and data sections was lacking.

To answer the third research question, an experimental evaluation was conducted, measuring different classifiers' performance in different scenarios. First, the overall classification performance was evaluated using 10-fold cross-validation with only the code sections of binaries. All the classifiers performed well achieving over 98% accuracy. Second, the effect of sample size was studied on classifiers trained with the complete feature set. 3 nearest neighbor and SVM classifiers had the best accuracies, requiring only 128 bytes to reach 90% accuracy. Most of the classifiers achieved 90% accuracy at 2000 bytes. Third, the classifier performance was evaluated on full binaries, containing both code and data sections. Random forest had the best performance at 90% accuracy. Finally, classifiers were both tested and trained with full binaries. The accuracies were much better than when classifiers were trained with only code sections, with random forest implemented in Azure achieving 99.2% accuracy. Overall, random forest had the best performance.

This thesis offered a comparison of supervised classifiers when identifying architectures using binary file contents. It also contributes to advancing the field by releasing parts of the implementation and data as open-source [1]. We believe this strongly supports the open-science as well as gives the opportunity to our peers to further improve the results and the research within the field.

1. `https://github.com/kairis/`

Bibliography

Abadi, Martín, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. “Tensorflow: a system for large-scale machine learning”. In *OSDI,* 16:265–283.

Abu-Nimeh, Saeed, Dario Nappa, Xinlei Wang, and Suku Nair. 2007. “A comparison of machine learning techniques for phishing detection”. In *Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit,* 60–69. ACM.

Amancio, Diego Raphael, Cesar Henrique Comin, Dalcimar Casanova, Gonzalo Travieso, Odemir Martinez Bruno, Francisco Aparecido Rodrigues, and Luciano da Fontoura Costa. 2014. “A systematic comparison of supervised classifiers”. *PloS one* 9 (4): e94137.

Amin, Rohan, Julie Ryan, and Johan van Dorp. 2012. “Detecting targeted malicious email”. *IEEE Security & Privacy* 10 (3): 64–71.

Baldoni, Roberto, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi. 2018. “A Survey of Symbolic Execution Techniques”. *ACM Comput. Surv.* (New York, NY, USA) 51 (3).

Balnaves, Mark, and Peter Caputi. 2001. *Introduction to quantitative research methods: An investigative approach.* Sage.

Bapat, Rohan, Abhijith Mandya, Xinyang Liu, Brendan Abraham, Donald E Brown, Hyojung Kang, and Malathi Veeraraghavan. 2018. “Identifying malicious botnet traffic using logistic regression”. In *Systems and Information Engineering Design Symposium (SIEDS), 2018,* 266–271. IEEE.

Bhatia, Nitin, et al. 2010. “Survey of nearest neighbor techniques”. *arXiv preprint arXiv:1007.0085.*

Bhattarai, Sulabh, and Yong Wang. 2018. “End-to-End Trust and Security for Internet of Things Applications”. *Computer* 51 (4): 20–27.

Buczak, Anna L, and Erhan Guven. 2016. “A survey of data mining and machine learning methods for cyber security intrusion detection”. *IEEE Communications Surveys & Tutorials* 18 (2): 1153–1176.

Castillo, Carlos, Debora Donato, Aristides Gionis, Vanessa Murdock, and Fabrizio Silvestri. 2007. “Know your neighbors: Web spam detection using the web topology”. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval,* 423–430. ACM.

Cawley, Gavin C, and Nicola LC Talbot. 2003. “Efficient leave-one-out cross-validation of kernel fisher discriminant classifiers”. *Pattern Recognition* 36 (11): 2585–2592.

Chen, Daming D, Maverick Woo, David Brumley, and Manuel Egele. 2016. “Towards Automated Dynamic Analysis for Linux-based Embedded Firmware”. In *NDSS.*

Chollet, François, et al. 2015. *Keras.* `https://keras.io`.

Clemens, John. 2015. “Automatic classification of object code using machine learning”. *Digital Investigation* 14:S156–S162.

Costin, Andrei, and Aurélien Francillon. 2014. “Dangerous Pyrotechnic’Composition’: Fireworks, Embedded Wireless and Insecurity-by-Design (short paper)”.

Costin, Andrei, Jonas Zaddach, Aurélien Francillon, Davide Balzarotti, and Sophia Antipolis. 2014. “A Large-Scale Analysis of the Security of Embedded Firmwares”. In *USENIX Security Symposium,* 95–110.

Costin, Andrei, Apostolis Zarras, and Aurélien Francillon. 2016. “Automated dynamic firmware analysis at scale: a case study on embedded web interfaces”. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security,* 437–448. ACM.

———. 2017. “Towards automated classification of firmware images and identification of embedded devices”. In *IFIP International Conference on ICT Systems Security and Privacy Protection,* 233–247. Springer.

De Nicolao, Pietro, Marcello Pogliani, Mario Polino, Michele Carminati, Davide Quarta, and Stefano Zanero. 2018. “ELISA: ELiciting ISA of Raw Binaries for Fine-Grained Code and Data Separation”. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment,* 351–371. Springer.

“Debian packages”. 2018. Debian. `https://packages.debian.org/stable/`.

“Debian CDs/DVDs archive”. 2019. Debian. `http://cdimage.debian.org/mirror/cdimage/archive/`.

“Debian Ports”. 2019. Debian. `https://www.ports.debian.org/`.

Denning, Peter J. 1980. “ACM President’s Letter: What is experimental computer science?” *Communications of the ACM* 23 (10): 543–544.

Dua, Sumeet, and Xian Du. 2016. *Data mining and machine learning in cybersecurity.* Auerbach Publications.

Dumais, Susan, and Hao Chen. 2000. “Hierarchical classification of Web content”. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval,* 256–263. ACM.

Dy, Jennifer G, and Carla E Brodley. 2004. “Feature selection for unsupervised learning”. *Journal of machine learning research* 5 (Aug): 845–889.

Enache, Adriana-Cristina, and Valentin Sgarciu. 2014. “Enhanced intrusion detection system based on bat algorithm-support vector machine”. In *Security and Cryptography (SECRYPT), 2014 11th International Conference on,* 1–6. IEEE.

Fawcett, Tom. 2006. “An introduction to ROC analysis”. *Pattern recognition letters* 27 (8): 861–874.

Ferri, César, José Hernández-Orallo, and R Modroiu. 2009. “An experimental comparison of performance measures for classification”. *Pattern Recognition Letters* 30 (1): 27–38.

“firmadyne scraper”. 2018. Visited on October 29, 2018. `https://github.com/firmadyne/scraper`.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2009. *The elements of statistical learning.* 2nd edition. Springer.

Galar, Mikel, Alberto Fernández, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. 2011. “An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes”. *Pattern Recognition* 44 (8): 1761–1776.

Gardner, Matt W, and SR Dorling. 1998. “Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences”. *Atmospheric environment* 32 (14-15): 2627–2636.

Gartner. 2017. “Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016”. Visited on December 21, 2018. `https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016`.

Ghahramani, Zoubin. 2004. “Unsupervised learning”. In *Advanced lectures on machine learning,* 72–112. Springer.

Given, Lisa M. 2008. *The Sage encyclopedia of qualitative research methods.* Sage Publications.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning.* MIT press.

Granboulan, Louis. 2018. "cpu_rec". `https://github.com/airbus-seclab/cpu_rec`.

HaddadPajouh, Hamed, Ali Dehghantanha, Raouf Khayami, and Kim-Kwang Raymond Choo. 2018. "A deep Recurrent Neural Network based approach for Internet of Things malware threat hunting". *Future Generation Computer Systems* 85:88–96.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. "Unsupervised learning". In *The elements of statistical learning,* 485–585. Springer.

Heffner, Craig. 2013. "From China, With Love". Visited on March 16, 2019. `http://www.devttys0.com/2013/10/from-china-with-love/`.

Hemel, Armijn. 2018. "Binary Analysis Next Generation". Visited on March 13, 2019. `https://github.com/armijnhemel/binaryanalysis-ng`.

Hemel, Armijn, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. "Finding software license violations through binary code clone detection". In *Proceedings of the 8th Working Conference on Mining Software Repositories,* 63–72. ACM.

Hosmer Jr, David W, Stanley Lemeshow, and Rodney X Sturdivant. 2013. *Applied logistic regression.* Volume 398. John Wiley & Sons.

Hu, Li-Yu, Min-Wei Huang, Shih-Wen Ke, and Chih-Fong Tsai. 2016. "The distance function effect on k-nearest neighbor classification for medical datasets". *SpringerPlus* 5 (1): 1304.

Hyppönen, Mikko, Linus Nyman, et al. 2017. "The Internet of (Vulnerable) Things: On Hypponen's Law, Security Engineering, and IoT Legislation". *Technology Innovation Management Review.*

Japkowicz, Nathalie. 2003. "Class imbalances: are we focusing on the right issue". In *Workshop on Learning from Imbalanced Data Sets II,* 1723:63.

"jigdo". 2018. Visited on November 25, 2018. `https://www.debian.org/CD/jigdo-cd/`.

Karagiozidis, Alexios. 2018. "Common Attack Vectors of IoT Devices". *Advanced Microkernel Operating Systems:* 27.

Kiang, Melody Y. 2003. “A comparative assessment of classification methods”. *Decision Support Systems* 35 (4): 441–454.

Kim, Jihyun, Jaehyun Kim, Huong Le Thi Thu, and Howon Kim. 2016. “Long short term memory recurrent neural network classifier for intrusion detection”. In *Platform Technology and Service (PlatCon), 2016 International Conference on,* 1–5. IEEE.

Kohavi, Ron, et al. 1995. “A study of cross-validation and bootstrap for accuracy estimation and model selection”. In *Ijcai,* 14:1137–1145. 2. Montreal, Canada.

Kononenko, Igor, and Matjaz Kukar. 2007. *Machine learning and data mining.* Horwood Publishing.

Kotsiantis, Sotiris B, I Zaharakis, and P Pintelas. 2007. “Supervised machine learning: A review of classification techniques”. *Emerging artificial intelligence applications in computer engineering* 160:3–24.

Kotsiantis, Sotiris B, Ioannis D Zaharakis, and Panayiotis E Pintelas. 2006. “Machine learning: a review of classification and combining techniques”. *Artificial Intelligence Review* 26 (3): 159–190.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. 2015. “Deep learning”. *nature* 521 (7553): 436.

Lee, Su-In, Honglak Lee, Pieter Abbeel, and Andrew Y Ng. 2006. “Efficient l˜ 1 regularized logistic regression”. In *AAAI,* 6:401–408.

Li, Tao, Chengliang Zhang, and Mitsunori Ogihara. 2004. “A comparative study of feature selection and multiclass classification methods for tissue classification based on gene expression”. *Bioinformatics* 20 (15): 2429–2437.

Li, Zhen, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. “VulPecker: an automated vulnerability detection system based on code similarity analysis”. In *Proceedings of the 32nd Annual Conference on Computer Security Applications,* 201–213. ACM.

Liaw, Andy, Matthew Wiener, et al. 2002. “Classification and regression by randomForest”. *R news* 2 (3): 18–22.

Lin, Wei-Chao, Shih-Wen Ke, and Chih-Fong Tsai. 2015. “CANN: An intrusion detection system based on combining cluster centers and nearest neighbors”. *Knowledge-based systems* 78:13–21.

Liu, Muqing, Yuanyuan Zhang, Juanru Li, Junliang Shu, and Dawu Gu. 2016. “Security analysis of vendor customized code in firmware of embedded device”. In *International Conference on Security and Privacy in Communication Systems,* 722–739. Springer.

Livadas, Carl, Robert Walsh, David Lapsley, and W Timothy Strayer. 2006. “Usilng machine learning technliques to identify botnet traffic”. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on,* 967–974. IEEE.

McDaniel, Mason, and Mohammad Hossain Heydari. 2003. “Content based file type detection algorithms”. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on,* 10–pp. IEEE.

Mena, Jesus. 2016. *Machine learning forensics for law enforcement, security, and intelligence.* Auerbach Publications.

Mukhopadhyay, Sayan. 2018. “Unsupervised Learning: Clustering”. In *Advanced Data Analytics Using Python,* 77–98. Springer.

Mukkamala, Srinivas, Andrew H Sung, and Ajith Abraham. 2005. “Intrusion detection using an ensemble of intelligent paradigms”. *Journal of network and computer applications* 28 (2): 167–182.

Nataraj, Lakshmanan, Sreejith Karthikeyan, Gregoire Jacob, and BS Manjunath. 2011. “Malware images: visualization and automatic classification”. In *Proceedings of the 8th international symposium on visualization for cyber security,* 4. ACM.

Ng, Andrew Y. 2004. “Feature selection, L 1 vs. L 2 regularization, and rotational invariance”. In *Proceedings of the twenty-first international conference on Machine learning,* 78. ACM.

Nordrum, Amy. 2016. “The internet of fewer things [news]”. *IEEE Spectrum* 53 (10): 12–13.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. “Scikit-learn: Machine Learning in Python”. *Journal of Machine Learning Research* 12:2825–2830.

Rahimi, Ali, and Benjamin Recht. 2008. “Random features for large-scale kernel machines”. In *Advances in neural information processing systems,* 1177–1184.

Rawat, Sanjay, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. “Vuzzer: Application-aware evolutionary fuzzing”. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

ReFirmLabs. 2018. “binwalk”. `https://github.com/ReFirmLabs/binwalk`.

Resende, Paulo Angelo Alves, and André Costa Drummond. 2018. “A Survey of Random Forest Based Methods for Intrusion Detection Systems”. *ACM Computing Surveys (CSUR)* 51 (3): 48.

Rish, Irina, et al. 2001. “An empirical study of the naive Bayes classifier”. In *IJCAI 2001 workshop on empirical methods in artificial intelligence,* 3:41–46. 22. IBM New York.

Robert, Christian. 2014. *Machine learning, a probabilistic perspective.*

Rodriguez, Juan D, Aritz Perez, and Jose A Lozano. 2010. “Sensitivity analysis of k-fold cross validation in prediction error estimation”. *IEEE transactions on pattern analysis and machine intelligence* 32 (3): 569–575.

Russell, Stuart J, and Peter Norvig. 2016. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,

Saberi, Alireza, Mojtaba Vahidi, and Behrouz Minaei Bidgoli. 2007. “Learn to detect phishing scams using learning and ensemble? methods”. In *Proceedings of the 2007 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology-Workshops,* 311–314. IEEE Computer Society.

Schmidt, Mark, Glenn Fung, and Rmer Rosales. 2007. “Fast optimization methods for l1 regularization: A comparative study and two new approaches”. In *European Conference on Machine Learning,* 286–297. Springer.

Shabtai, Asaf, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. 2009. “Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey”. *information security technical report* 14 (1): 16–29.

Shafieian, Saeed, Mohammad Zulkernine, and Anwar Haque. 2015. “CloudZombie: Launching and detecting slow-read distributed denial of service attacks from the cloud”. In *Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), 2015 IEEE International Conference on,* 1733–1740. IEEE.

Shoshitaishvili, Yan, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. “Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware”. In *NDSS.*

Shoshitaishvili, Yan, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, et al. 2016. “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In *IEEE Symposium on Security and Privacy.*

Shotton, Jamie, Toby Sharp, Pushmeet Kohli, Sebastian Nowozin, John Winn, and Antonio Criminisi. 2013. “Decision jungles: Compact and rich models for classification”. In *Advances in Neural Information Processing Systems,* 234–242.

Sikorski, Michael, and Andrew Honig. 2012. *Practical malware analysis: the hands-on guide to dissecting malicious software.* No Starch Press.

Sikos, Leslie F. 2018. *AI in Cybersecurity.* Volume 151. Springer.

Sokolova, Marina, Nathalie Japkowicz, and Stan Szpakowicz. 2006. “Beyond accuracy, F-score and ROC: a family of discriminant measures for performance evaluation”. In *Australasian joint conference on artificial intelligence,* 1015–1021. Springer.

Sokolova, Marina, and Guy Lapalme. 2009. “A systematic analysis of performance measures for classification tasks”. *Information Processing & Management* 45 (4): 427–437.

Suthaharan, Shan. 2014. “Big data classification: Problems and challenges in network intrusion prediction with machine learning”. *ACM SIGMETRICS Performance Evaluation Review* 41 (4): 70–73.

Sutton, Michael, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery.* Pearson Education.

Sutton, Richard S, and Andrew G Barto. 2018. *Reinforcement learning: An introduction.* MIT press.

Suykens, Johan AK, and Joos Vandewalle. 1999. “Least squares support vector machine classifiers”. *Neural processing letters* 9 (3): 293–300.

Thomas, Sam L, Tom Chothia, and Flavio D Garcia. 2017. “Stringer: measuring the importance of static data comparisons to detect backdoors and undocumented functionality”. In *European Symposium on Research in Computer Security,* 513–531. Springer.

Thomas, Sam L, and Aurélien Francillon. 2018. “Backdoors: Definition, Deniability and Detection”. In *International Symposium on Research in Attacks, Intrusions, and Defenses,* 92–113. Springer.

Thomas, Sam L, Flavio D Garcia, and Tom Chothia. 2017. “HumIDIFy: a tool for hidden functionality detection in firmware”. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment,* 279–300. Springer.

University of Jyväskylä. 2010. “Research strategies”. Visited on January 10, 2019. `https://koppa.jyu.fi/avoimet/hum/menetelmapolkuja/en/methodmap/strategies/images/circle_strat.gif`.

Uysal, Alper Kursat. 2016. “An improved global feature selection scheme for text classification”. *Expert systems with Applications* 43:82–92.

Wheeler, David A. 2014. “Preventing Heartbleed”. *IEEE Computer* 47 (8): 80–83.

Wong, Tzu-Tsung. 2015. “Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation”. *Pattern Recognition* 48 (9): 2839–2846.

Woźniak, Michał, Manuel Graña, and Emilio Corchado. 2014. “A survey of multiple classifier systems as hybrid systems”. *Information Fusion* 16:3–17.

Xie, Wei, Yikun Jiang, Yong Tang, Ning Ding, and Yuanming Gao. 2017. “Vulnerability Detection in IoT Firmware: A Survey”. In *Parallel and Distributed Systems (ICPADS), 2017 IEEE 23rd International Conference on,* 769–772. IEEE.

Yousefi-Azar, Mahmood, Vijay Varadharajan, Len Hamey, and Uday Tupakula. 2017. “Autoencoder-based feature learning for cyber security applications”. In *Neural Networks (IJCNN), 2017 International Joint Conference on,* 3854–3861. IEEE.

Zaddach, Jonas, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. 2014. “AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems’ Firmwares”. In *NDSS.*

Zaddach, Jonas, and Andrei Costin. 2013. “Embedded devices security and firmware reverse engineering”. *Black-Hat USA.*

Zhang, Harry. 2004. “The optimality of naive Bayes”. *AA* 1 (2): 3.

Zhang, Zhi-Kai, Michael Cheng Yi Cho, Chia-Wei Wang, Chia-Wei Hsu, Chong-Kuan Chen, and Shiuhpyng Shieh. 2014. “IoT security: ongoing challenges and research opportunities”. In *Service-Oriented Computing and Applications (SOCA), 2014 IEEE 7th International Conference on,* 230–234. IEEE.

Zhu, Ruijin, Baofeng Zhang, Junjie Mao, Quanxin Zhang, and Yu-an Tan. 2017. “A methodology for determining the image base of arm-based industrial control system firmware”. *International Journal of Critical Infrastructure Protection* 16:26–35.

Appendices

A Toolset JSON files

Figure 15: Output JSON from FIRMADYNE scraper

```
[
  {
    "version": "9.0.0/",
    "url": "http://cdimage.debian.org/mirror/cdimage/archive/9.0.0/
       mips64el/jigdo-cd/debian-9.0.0-mips64el-netinst.jigdo",
    "vendor": "debian",
    "file_urls": [
      "http://cdimage.debian.org/mirror/cdimage/archive/9.0.0/
         mips64el/jigdo-cd/debian-9.0.0-mips64el-netinst.jigdo"
    ],
    "architecture": "mips64el/",
    "files": [
      {
        "path": "debian/mips64el/9.0.0/debian-9.0.0-mips64el-netinst
           .jigdo",
        "checksum": "c40a53297590bc6861a47e4ff123978e",
        "url": "http://cdimage.debian.org/mirror/cdimage/archive
           /9.0.0/mips64el/jigdo-cd/debian-9.0.0-mips64el-netinst.
           jigdo"
      }
    ]
  }
]
```

Figure 16: Output JSON from Jigdo downloader

```
{
  "mips64el": [
    {
      "iso_path": "/home/samakair/work/thesis/app/modules/scraper/
         output/debian/isos/mips64el/9.0.0/debian-9.0.0-mips64el-
         netinst.iso",
      "version": "9.0.0"
    }
  ]
}
```

Figure 17: Output JSON from Debian extractor

```
{
  "mips64el": [
    {
      "iso": "/home/samakair/work/thesis/app/modules/scraper/output/
         debian/isos/mips64el/9.0.0/debian-9.0.0-mips64el-netinst.
         iso",
      "version": "9.0.0",
      "deb_path": "/home/samakair/work/thesis/app/modules/scraper/
         output/debian/isos/mips64el/9.0.0/debs/libacl1_2.2.52-3+
         b1_mips64el.deb"
    }
  ]
}
```

Figure 18: (Partial) Output JSON from Debian unpacker

```
{
  "mips64el": [
    {
      "software": "libacl1",
      "controlinfo": "Package: libacl1\nSource: acl (2.2.52-3)\
         nVersion: 2.2.52-3+b1\nArchitecture: mips64el\nMaintainer:
          Anibal Monsalve Salazar <anibal@debian.org>\nInstalled-
         Size: 69\nDepends: libattr1 (>= 1:2.4.46-8), libc6 (>=
         2.4)\nConflicts: acl (<< 2.0.0), libacl1-kerberos4kth\
         nSection: libs\nPriority: required\nMulti-Arch: same\
         nHomepage: http://savannah.nongnu.org/projects/acl/\
         nDescription: Access control list shared library\n This
         package contains the libacl.so dynamic library containing\
         n the POSIX 1003.1e draft standard 17 functions for
         manipulating\n access control lists.\n",
      "package": "/home/samakair/thesis/work/app/modules/scraper/
         output/debian/isos/mips64el/9.0.0/debs/libacl1_2.2.52-3+
         b1_mips64el.deb",
      "filestructure": [
        {
          "hash": "",
          "path": "/",
          "size": "0",
          "date": "2016-02-07"
        },
          .
          .
        {
          "hash": "",
          "path": "/usr/share/doc/libacl1/copyright",
          "size": "761",
          "date": "2009-08-25"
        }
      ],
      "iso": "/home/samakair/thesis/work/app/modules/scraper/output/
         debian/isos/mips64el/9.0.0/debian-9.0.0-mips64el-netinst.
         iso",
      "version": "2.2.52-3+b1"
    }
  ]
}
```

Figure 19: Output JSON from binary extractor

```json
[
  {
    "filesize": 24952,
    "code_sections": [
      ".init",
      ".text",
      ".MIPS.stubs",
      ".fini"
    ],
    "endianness": "little",
    "architecture": "mips64el",
    "deb_package": "acpi_1.7-1+b1_mips64el.deb",
    "filename": "/home/samakair/work/thesis/app/modules/scraper/
       output/debian/isos/mips64el/9.0.0/debs/acpi_1.7-1+
       b1_mips64el.deb_extraction/usr/bin/acpi",
    "wordsize": 64,
    "only_code": "/home/samakair/work/thesis/app/modules/scraper/
       output/debian/isos/mips64el/binaries/1923
       f9252c9cfc169813703625c3f58f.code",
    "only_code_size": 10176,
    "filehash": "1923f9252c9cfc169813703625c3f58f",
    "fileinfo": "/home/samakair/work/thesis/app/modules/scraper/
       output/debian/isos/mips64el/9.0.0/debs/acpi_1.7-1+
       b1_mips64el.deb_extraction/usr/bin/acpi: ELF 64-bit LSB
       shared object, MIPS, MIPS64 rel2 version 1 (SYSV),
       dynamically linked, interpreter /lib64/ld.so.1, BuildID[sha1
       ]=0e298fb929bb982a7bcf69ba3b7cd5537718ed40, for GNU/Linux
       3.2.0, stripped"
  }
]
```

B Revisions

Revision date	Revision comments
14.05.2019	a. Corrected 150 000 to 105 000 on page 35 b. Added acknowledgement on page 35