POZNAN UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTING AND TELECOMMUNICATION
Institute of Computing Science

Master's thesis

CLASSIFICATION-DRIVEN SYMBOL EXTRACTION IN BINARY FILE REVERSE ENGINEERING

Michał Kapała

Supervisor
Marcin Borowski, PhD

POZNAŃ 2023

Abstract

Decompilation and static analysis of binary files are, alongside debugging, the core part of software reverse engineering. Common applications of binary reverse engineering include malware detection and analysis, vulnerability discovery, algorithm and network protocol analysis, toolchain provenance recovery and more. The main purpose of static analysis is to provide information about an application's control and data flows.

One of the most natural and common means to explain an application's purpose and behaviour is the semantic labelling of code symbols, such as function or variable names. During the compilation process symbol information is inherently lost, unless explicitly generated or embedded by the compiler. Real-world applications are usually stripped from structured symbol data, which cannot be recovered by a decompiler. As a result of decompilation semantically meaningful names are replaced with auto-generated labels, and as such the recovered code symbols become more difficult or nigh impossible to understand for analysts. Solutions such as Debin [1] and DIRE [2] were developed to predict unknown symbol labels from decompiled code. The predictions of both systems are solely based on external training data sets, which renders restoring the original symbol names of unseen binaries difficult.

This thesis proposes a different approach to symbol name recovery, combining extraction and supervised classification of unstructured text found in binaries with decompiled binary information to find potential symbol names and predict original function names. The author contributes dubRE, a set of tools, data sets and machine learning models aiming to evaluate classifier performance for the tasks of symbol name identification and cross-reference path relevance evaluation. The most accurate model trained for function name classification was Gaussian Naive Bayes with 62% accuracy while the random forest method appeared as the best cross-reference path classifier with an F1 score of 0.705.

Keywords – symbol extraction, reverse engineering, static analysis, machine learning

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cyber security is a rapidly growing industry. According to forecasts, the European cybersecurity market worth is prognosed to double between 2021-2027 [16]. With the new technologies and IT products emerging every day comes a demand for both security specialists and tools which keep various threats at bay. Antivirus scanners, browsers, video game anti-cheat systems, digital wallets - all are subject to application security analysis.

One of the most common ways of software analysis is reverse engineering which can be defined as rediscovering the design of an application. Many techniques exist in the industry to help inspect and track the behaviour of software products, including restoration of source code also known as decompilation. It is particularly useful for malware analysis, application penetration testing and vulnerability discovery among other use cases. Code produced by decompilation tools is not without a flaw - it is usually much harder to read than regular application code due to a lack of semantic code names and thus requires substantial knowledge of computer science.

Artificial intelligence was proven effective in automated binary file analysis. Recent research showed methods to restore the semantic meaning of decompiled code. He et al. [1] proposed a method of structuring binary code into higher representation of assembly code to then use classification and structured prediction models, which learn from open source applications, to predict variable and function names in unknown executable files. Lacomis et al. [2] adjusted that approach with deep learning neural network architecture that learns both structural and lexical representations of decompiled code to improve the accuracy of predictions for the code element names.

1.1 Motivation

Traces of debug information can be found in stripped binaries. Complex applications such as web servers, video games, physics engines, or libraries often contain significant amounts of plain text data used for error information display, logging, or internal function calls which include the original names of functions, variables, types, and other code elements. Applications of these kinds can be the target of real-world reverse engineering tasks, e.g. for the sake of vulnerability detection or toolchain provenance recovery [6].

While symbol information contained in text data is relatively easy to uncover for human researchers, manually finding and applying it in decompiled code can become a tedious and time-consuming task as large binaries can include hundreds of thousands of strings. An automated

method of debug symbol extraction from text data could provide vital and original information about the binary at a relatively low cost of human time and effort.

Moreover, text data in executable files can be referenced from functions, which ties it to a certain program execution context. Decompilers can provide tools to traverse such references both between a string of text and a function or between two functions. That enables the hypothetical automated method to learn text data usage patterns in executable code. This fact is important in cases where debug symbols describe code elements that do not reference the text data directly but are found in close proximity to the described target in reference graphs.

The above approach is less invasive and more accurate than predictions made based on external data sets - only positively labelled parts of text data would be applied to decompiled code. The limited symbol data is extracted from the original binary rather than all the predictions guessed or synthesised from external samples. While the prediction strategy presented by Debin and DIRE can be applied to all of the decompiled code used as input, this method would have a significantly more narrow area of impact, which could be considered either a drawback or a perk depending on the researcher's needs.

The presented techniques could be extended for other code symbols such as variable names or data types using debug symbols and Run-Time Type Information (RTTI) as truth sources. However, extraction of every type of code symbol is a fairly complex task, and a classification system for multiple code symbol types would likely require a much more sophisticated approach than the proposed one, along with significantly larger data sets needed to achieve satisfying accuracy levels. This thesis focuses on extracting function name information from string data as function names are easy to manually recognise and label, as well as they provide vital information about the global control flow of an application. Other elements such as variable or type names are out of scope for this work.

1.2 Research goals

Considering the above foreword, this thesis presents experimental research attempting to answer the below questions:

RQ1: What are the current methods and possibilities for automated semantic labelling of decompiled code?
RQ2: What are the downsides of state-of-the-art techniques for symbol information restoration in stripped binaries?
RQ3: Is automated extraction-based function naming possible to achieve in debug symbol-stripped binary files using machine learning techniques?

1.3 Structure of the thesis

This thesis consists of 7 chapters - chapter 2 explains what software analysis is and how its techniques can be employed for security-oriented research on software; it also introduces the reader to concepts of artificial intelligence and machine learning. Chapter 3 reviews the current approaches to text representation in machine learning, automated analysis of software, and recent research on techniques for providing semantically meaningful code symbols in decompiled code. Chapters 4, 5 and 6 propose a method for extraction and classification of code artifacts from binary files as opposed to prediction of code symbol names and evaluate an experimental implementation. Lastly, chapter 7 summarises the key points of the thesis.

Chapter 2

Domain knowledge

This chapter serves as an introduction to the subjects of software reverse engineering and machine learning. It explains the concepts present in both domains important from the perspective of this thesis.

2.1 Software reverse engineering

This section briefly presents the concepts of software reverse engineering and executable binary files, along with the two main approaches to binary file analysis.

2.1.1 Definition of reverse engineering

Reverse engineering is a process that can relate to many industries and fields of science. As the term suggests, it is the reversed process of engineering - a starting point is a human-made object and the product is the understanding of its design and behaviour. An important aspect of reverse engineering is the lack or inaccessibility of sufficient external knowledge on the researched subject.

Software reverse engineering aims to rediscover the internal structure, control and data flow of a program without its source code or any human-readable representation. Among the motivations for gaining such knowledge could be the need for interaction with a system, reproduction of its behaviour, access to internal data, or emulation of the system. Common industrial applications for software reverse engineering are malware analysis, application vulnerability discovery and firmware diagnostics [6].

Two main approaches to software reverse engineering are dynamic and static analysis. Dynamic analysis relies on interaction with a program in a runtime environment while static analysis employs binary code structuring techniques in order to restore a program's representation in a human-readable form.

2.1.2 Dynamic analysis

Dynamic analysis focuses on the inspection and analysis of the program's properties while it is being executed [17]. Debugging, emulation and communication over a network are all means to interact and get information about the state of an application at a given point in time. Dynamic approach is particularly useful for a program's internal data flow analysis as data is often computed or obtained at runtime.

While dynamic analysis is used for control flow analysis, it can only give information about one execution path of a linear program. That fact becomes a serious limitation in reasoning about a system with a large number of branching points in complex systems as more and more debugging sessions are needed to get information about how the program can behave. In contrast, static analysis produces a description of all possible execution paths which does not require as much time and effort to analyse. Usually, both methods are used as complementary to help reverse engineers understand the application logic [17].

The ability to debug or manipulate the internals of a running process can be restricted in order to protect sensitive code or data of the running application. Various techniques can be applied to detect or block process debugging, ranging from minor modifications to a program's code [18] to complex external anti-debugging systems running alongside the application.

2.1.3 Static analysis

In the context of software reverse engineering, static analysis stands for automated techniques employed to reason about an application without executing it [19]. Prime examples of static analysis techniques are disassembly and decompilation of executable files - a process reverse to compilation aiming to create source code-like representations from application files. Other examples of static analysis methods include binary code fingerprinting [20], binary file visualisation [21] or even file hashes used to check data integrity.

Decompilers are sophisticated tools using static analysis to structure binary code and transform it into human-readable code, for instance in C programming language. Such code representations can be successfully used to reason about both data and control flow in unprotected binaries. For compiled binaries Common tools for binary code decompilation are Ghidra and *Interactive Disassembler* (IDA); the latter was used in this thesis to provide binary information for classifier training.

Similarly to the case of dynamic analysis, there exist mechanisms to prevent software users from analysing files. A widespread protection technique against static analysis tools is binary code obfuscation. Obfuscated code is intended to look unstructured to automatic tools and often cannot be correctly interpreted without inverting the process. Obfuscated binaries were not used as data in this thesis.

Binary code static analysis can be used to create application security tools such as malware detectors [22] [23], vulnerability detectors, secret scanners and more. Furthermore, static analysis tools can introduce optimisations to reverse engineering process, e.g. by improving the quality of code structuring during the decompilation process [24] or by restoring symbol information in stripped binaries [1] [2].

2.1.4 Executable files

Executable files contain binary code which can execute directly in the environment of an operating system. They store machine code and data needed to run the application in binary format. During the execution of a program, the executable's contents are loaded into the memory of a process as its initial state. Two examples of popular executable formats are Executable and Linkable Format (ELF) for Unix-based operating systems and Portable Executable (PE) for Windows systems. Executable binaries are a vital part of any operating system - from system utilities, libraries and

services through antivirus systems to user applications, all use a common format to store data on disk and load it into memory at runtime. A deep understanding of executable file internals is crucial for software reverse engineering.

Executables are produced from source code written in a compiled programming language, for instance, C or C++. A compiler performs several steps to translate the code into machine code and produces a file in the target format. Executable files are compiled either to platform-native machine code or translated into an intermediate representation of bytecode which can be interpreted by a programming language runtime and compiled to machine code later by a Just-In-Time (JIT) compiler. An example of such a solution is the .NET platform where the executables store Intermediate Language (IL) bytecode on disk. Although static analysis tools exist for such files, they are out of the scope of this thesis.

During software development, compilers emit symbol files that store information on variable and function names from the source code and map them to the corresponding addresses in the compiled binary so that they are available to programmers during debugging. Widely used symbol file formats are DWARF for ELF executables and Program Database (PDB) for PE format. During the compilation of software intended for production use symbol information is stripped and cannot be fully recovered without the access to application's source code.

2.2 Machine learning

The purpose of this section is to introduce the reader to artificial intelligence and its main subdomains. It provides an overview of classification problems followed by descriptions of the classification models and evaluation metrics relevant to the context of the conducted research.

2.2.1 Overview of artificial intelligence subfields

Artificial intelligence (AI) and machine learning (ML) both are extensive fields of computer science. With the rise of large language models (LLMs) and rapid growth in their popularity [25], the two terms are often incorrectly treated as synonymous [26]. This chapter aims to provide insight into different areas of AI research and help the reader distinguish between some of its branches, as well as introduce classification methods and evaluation metrics used in this thesis.

Artificial intelligence is an umbrella term for many different forms of systems capable of automatic learning and reasoning about data. While similar in the general idea, AI systems differ in data requirements, training methods and learning algorithms used. Firstly, a distinction between AI systems can based on input data used for training, providing categories of supervised, unsupervised and reinforcement learning. Secondly, such systems can be differentiated by the type of employed algorithm and the internal representation of knowledge. Lastly, a classification of AI systems can be made based on their applications in research and industry.

Similarly to AI, the definition of machine learning is hard to formulate strictly. T. Mitchell defines the field of ML as 'concerned with the question of how to construct computer programs that automatically improve with experience' [27] which borrows its concepts from multiple fields of science such as statistics, biology, information theory, philosophy and other domains [27]. A more pragmatic view of machine learning is to perceive it as a set of methods for pattern detection

that allow for decision-making on future data, such as prediction-making [28]. Machine learning is widely used across different fields of science and found applications in major industries as a reliable and affordable technique for solving highly complex problems, such as analysis in business intelligence [29], medical diagnosis [30], image recognition [31] and many more. Example use cases for ML in computer science include natural language processing [32] [33], malware detection [23] and software analysis [21] [34].

2.2.2 Supervised learning

Systems trained using data containing values that are to be later predicted fall under the category of supervised learning. Supervised learning methods are commonly used for classification and regression problems to predict a continuous numeric value (regression) or a discrete value representing a class, sometimes referred to as a label (classification). The feature to be predicted is indicated to the prediction model during its training and included in the sample data. After training, the model is provided with a different data set without the target feature in order to make a prediction for every sample. To measure the model's performance the real values of samples (which need to be included in the testing set) are compared with the obtained predictions.

The behaviour of classifier algorithms can be modified using multiple parameters which affect the results. Such a process is called hyperparameter optimisation and is commonly performed to improve model performance, based either on an empirical approach and experimental parameter suites or automated techniques such as Bayesian optimisation [35]. To evaluate a trained classifier, an independent testing set ought to be used, which may result in additional computational overhead, particularly expensive for large data sets. Cross-validation is a resampling-based evaluation technique operating exclusively on the training data to estimate how well a model would perform tested on independent data. There exist several variants of cross-validation [36], one of which is k-fold cross-validation. In this scenario, the data is divided into k equally-sized subsets. One of those folds is chosen as the testing set with the remaining data treated as the validation set. The model is then trained and tested in a k-iterations loop, with each iteration using a different testing set. The result of k-fold cross-validation is the average performance across all iterations [36].

2.2.3 Unsupervised learning

Unsupervised methods, as opposed to supervised learning, provide unlabelled data during model training, which results in an inability to assess the results since there is no universal result validation technique [37]. A possible interpretation of unsupervised learning could be the set of methods for pattern recognition learning, especially useful for the processing of unstructured data characterised by extremely high levels of noise [38].

Unsupervised learning is employed for the tasks of exploratory data analysis [37] with example applications of unsupervised learning including clustering and data dimensionality reduction [38]. The common techniques used by unsupervised models to solve the above problems are factor analysis (FA), principal component analysis (PCA), and independent component analysis (ICA) [38] [39].

2.2.4 Reinforcement learning

Reinforcement learning can be viewed as a middle ground between supervised and unsupervised learning - in this approach, the agent model is supposed to discover a policy for decision-making that best satisfies a goal. The learning process is indirect, providing the agent with knowledge through repeated interactions with the surrounding environment and analysis of feedback data. During training, the agent is given the initial state of the environment with a finite set of possible interactions. For each of the taken actions, a score is calculated by a reward function to evaluate the agent's decision in the context of a new environment state. The reward function's purpose is to promote certain decisions and discriminate against others in order to achieve the end goal, for instance by introducing time constraints to the environment. The agent's only goal is to maximise the reward score by finding the best available behavioral policy. Agent's training ends with a certain state of the agent or environment being reached or after a set limit of time units (e.g. performed actions) is exceeded [40].

A common problem in reinforcement learning algorithms is balancing the agent's decision-making between the exploitation of the best-known policy and the exploration of new policies which may result in higher cumulative reward in the long run. The dilemma is still an unsolved puzzle to mathematical research [26]. An example subfield of reinforcement learning is Q-learning, commonly used in robotics [41], video games [42] and autonomous vehicle research [43].

2.2.5 Classification problems

Prediction of future data based on historical data constitutes a vast set of problems solved by machine learning. Such prediction tasks are divided into classification problems, where the predicted value set is discrete (finite) and represents possible classes to be assigned by the ML model and regression problems with a continuous (infinite) set of possible prediction values. A special case of a classification problem is binary classification with only 2 possible output classes by convention labelled as P (positives) and N (negatives). The result of a classification model test is commonly represented in the form of a square confusion matrix of C length where C is the size of input class set and m_{ij} is the number of samples belonging to class i and assigned to class j [44].

$$CM = \begin{bmatrix} m_{11} & m_{12} & ... & m_{1C} \\ m_{21} & m_{22} & ... & m_{2C} \\ ... & ... & ... & ... \\ m_{C1} & m_{C2} & ... & m_{CC} \end{bmatrix} \tag{2.1}$$

Confusion matrix for multi-class classification

In case of binary classification the confusion matrix includes 4 elements where TC represents correct assignments to class C and FC represents incorrect assignments to C where $C \in \{P, N\}$.

$$CM = \begin{bmatrix} TP & TN \\ FP & FN \end{bmatrix} \tag{2.2}$$

Confusion matrix for binary classification

Commonly used measures for evaluation of binary classification results include accuracy, precision, recall and F1-score among other indicators. Accuracy (ACC) is the simplest metric, measuring the percentage of correct predictions out of all predictions made on the testing sample set. Given the training data set balance in terms of sample class ratio, accuracy is suitable for evaluating the overall performance of a classifier.

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.3}$$

Accuracy metric for binary classification

Precision ($PREC$) represents the percentage of correct positive predictions out of all positive predictions. It measures the quality of positive predictions made by a classifier.

$$PREC = \frac{TP}{TP + FP} \tag{2.4}$$

Precision metric for binary classification

Recall ($RECALL$) measures the percentage of correct positive predictions out of all real positives in the test set. The metric measures how well a classifier predicts positive samples, which is essential in classification tasks where type II errors (false negatives) are highly undesirable [45].

$$RECALL = \frac{TP}{TP + FN} \tag{2.5}$$

Recall metric for binary classification

F1-score ($F1$) is a general performance metric unaffected by training set class imbalance (contrary to accuracy) which is useful for evaluation performed using limited real-world data sets. It is calculated as the harmonic mean of precision and recall.

$$F1 = 2 \cdot \frac{PRECISION \cdot RECALL}{PRECISION + RECALL} \tag{2.6}$$

2.2.6 Classifier families

Supervised learning classification models can be divided into categories based on the type of algorithm used to make predictions. Below are presented the selected types and examples of classifiers used throughout the experimental research on dubRE.

Bayesian classifiers rely on the concept of the Bayesian network, a probabilistic model based on Bayes' theorem of conditional probability. A Bayesian network can be represented as a direct acyclic graph (DAG) [46] [47] [48].

$$P(A|B) = \frac{P(A) \cdot P(B|A)}{P(B)} \tag{2.7}$$

Conditional probability (Bayes' theorem)

Naive Bayes classifiers learn from conditional probability of feature values given a class occurrence, assuming strong conditional independence between pairs of features [48] [49]. The Gaussian Naive Bayes variant assumes a normal distribution of feature values.

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{2.8}$$

Gaussian Naive Bayes classifier [50]

Decision tree classifiers are binary trees that represent a reasoning process. The nodes of a decision tree represent branch split conditions and its leaves indicate the output prediction class. During training the nodes are split recursively, adjusting the split conditions to fit the training data. A sample's label is chosen by answering a series of questions (evaluating conditions) regarding the input features, starting from the root of the tree [51]. Decision trees are used as a base classifier type in ensemble models such as random forests. An example illustration of a simple decision tree is presented in Fig. 2.1.

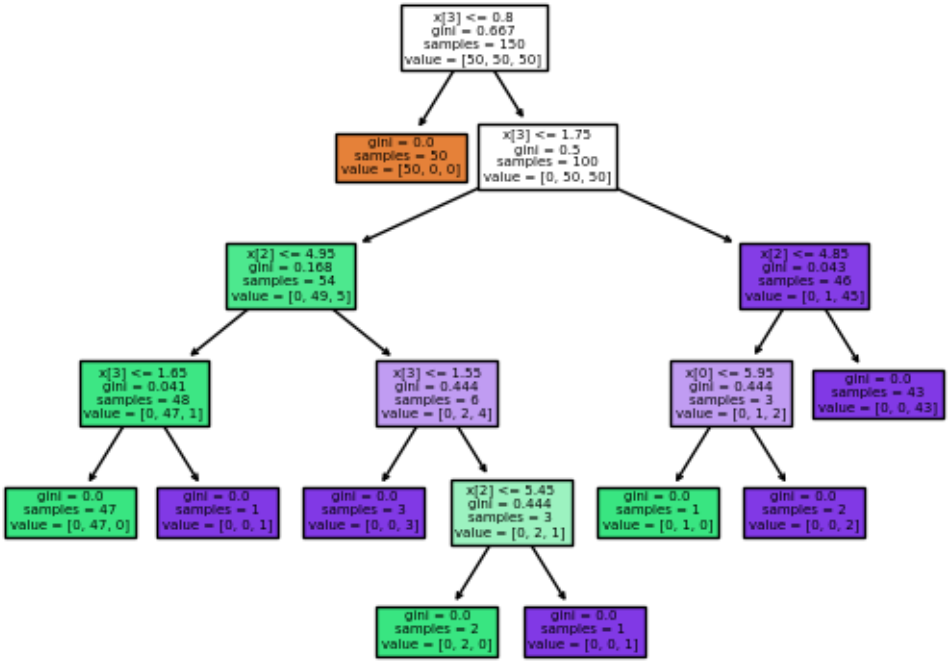

FIGURE 2.1: A visualisation of a decision tree (scikit-learn) [3]

Linear classifiers (also referred to as function classifiers [26] [52]) are non-probabilistic models based on optimisation theory and statistical estimation [26]. Support vector classifiers rely on multidimensional spaces to represent features and hyperplane separation of those as the learning objective [26]. Linear SVM classifier was found effective in text classification [53]. Logistic regression relies on regression analysis utilising the logit function. Example fields of application for logistic regression classifiers are medical industry [54] and spam detection [26].

Multi-layer perceptron (MLP) is an artificial neural network. Neural networks are composed of interconnected layers of artificial neurons, with each layer passing signals to the next adjacent layer. The first input layer encodes the features, the middle hidden layers transform the input with both linear summation and non-linear activation function, and the last single-node layer represents the predicted output [4] as shown in Fig. 2.2.
Similarly to linear classifiers, lazy classifiers (also called instance-based classifiers) are non-probabilistic models [26] which defer computation of its representation until a prediction is made and only computes it for the local space of a given point [55]. A prime example of a lazy classifier is k-Nearest Neighbors classifier. Feature vectors are represented as points in multidimensional space and the prediction is made based on the closest k points in that space [56].

Ensemble classifiers are composed of a set of independent base classifiers cooperatively trained on a data set [57]. Predictions made by the collective of estimators are then generalised by a

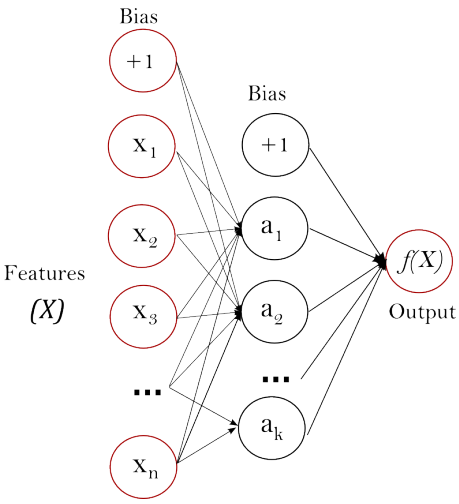

FIGURE 2.2: One-hidden layer MLP (scikit-learn) [4]

fusion method, such as voting, to reduce the error caused by extreme predictions. [57]. Random forest is an ensemble model using randomised decision trees as the base classifier. To generalise the collective opinion of the base estimator set, random forests use either a voting system for base classifier instances or prediction averaging [58]. Random forest has been proven effective as a general-purpose classifier and thus is used across various domains of classification problems [59]. Adaptive boosting relies on repeatedly fitting base estimators on modified versions of data [60]. During training the group of base estimators is ranked and weighted in many iterations based on their performance [61]. The ensemble can utilise any weak (base) classifier, for instance decision trees, Bayesian networks, or logistic regression.

2.2.7 Natural language processing

Natural language processing (NLP) is a field of science focused on automated analysis of human languages [62]. There exist countless industrial applications for NLP systems, e.g. information retrieval, automated text translation, chatbots, speech synthesis and recognition, question answering, sentiment analysis, or spam filtering [62] [63]. In recent years, machine learning-driven text analysis has become a trend [64].

Text data is not suitable for NLP systems in raw form. In order to process text with a classification system, the data needs to be transformed into a numerical representation understandable for the target system. Word embedding is a transformation of text into a distributed representation preserving the semantic and syntactic information of the text [65], such as multi-dimensional vector space.

Pennington et al. [66] distinguished two main categories of word vector-based NLP models: global matrix factorisation methods and local context window methods. The first group utilises statistical information about the training text corpus to generate a low-dimensional language representation, for instance, organised in a word co-occurrence matrix [66]. An example word-word co-occurrence matrix for the text 'Fool of a Took' is presented below as Table 2.1.

	Fool	of	a	Took
Fool	0	1	0	0
of	1	0	1	0
a	0	1	0	1
Took	0	0	1	0

TABLE 2.1: An example of a word co-occurrence matrix

Shallow context window-based methods rely on constrained local contexts within a text corpus, e.g. sentence-wide, to make predictions about a text continuation or missing content. Two proposals of such methods were made by Mikolov et al. [5] where words are represented as points in a continuous multi-dimensional vector space. A continuous *bag-of-words* (CBOW) model predicts a word from a context by summing all the word representations in the context. A continuous *skip-gram* model performs the reverse task - the prediction of surrounding context for a given word in the same vector space (see Fig. 2.3) [5].

The two above architectures are based on simple data structures - a bag of words (BOW) and skip-grams. A bag of words would be represented as a set of unique words, each paired with respective occurrence numbers. Skip-grams are a special case of n-grams; n-grams are n-element sequences of tokens, such as letters or words, in a strict order, e.g. derived from the original text. Skip-grams extend n-grams with wildcard tokens representing unknown values [67].

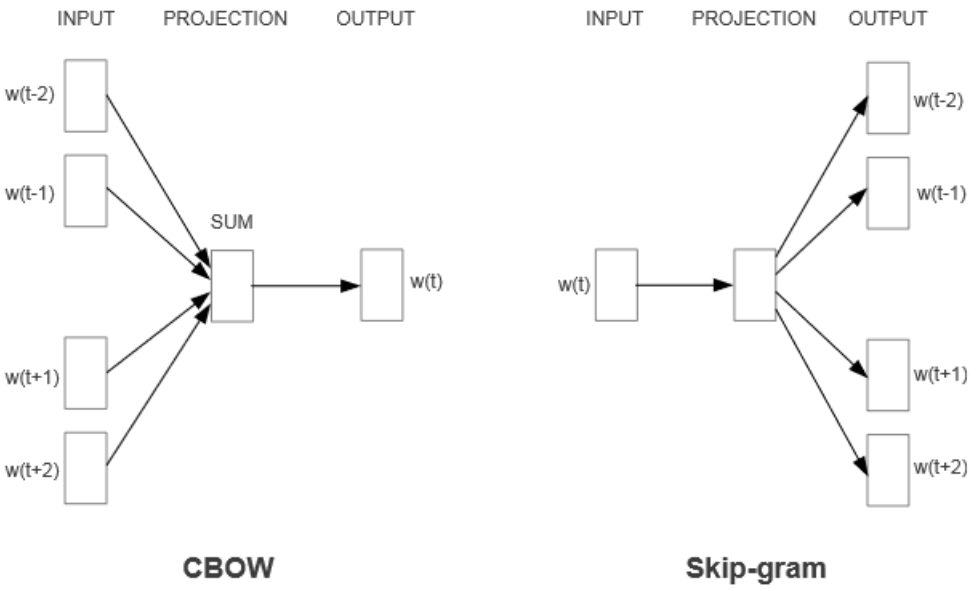

FIGURE 2.3: CBOW and continuous skip-gram architectures (Mikolov et al.) [5]

Chapter 3

Literature review

This chapter presents an overview of the current state of research on machine learning in the fields of natural language processing (NLP), specifically putting emphasis on word embedding techniques. It further discusses the role of machine learning in security-oriented software analysis and reviews state-of-the-art recovery techniques for debugging information in the case of decompiled, unobfuscated and stripped binaries.

3.1 Machine learning in natural language processing

In recent years, machine learning-driven analysis of text documents has been adopted for diverse commercial purposes. Industrial applications of natural language processing include the tasks of fake news identification [68], sentiment analysis [69], document classification [70] and, last but not least, large language models (LLMs) that just recently have stimulated rapid growth in NLP research [71].

The NLP-based prediction models can be broadly divided into generative and discriminative [63]. The generative approaches are typically employed to synthesise text output while discriminative models are text data classifiers functioning on various levels of granularity [63]. Examples of generative methods include Bayesian networks and generative adversarial networks (GAN) [72]. SVM and logistic regression could make illustratory representantives of text classifiers [63] [73]. Before a model can use non-numerical data, the process of feature extraction is performed to transform raw data into its numerical representation. In the context of NLP, this step is called word embedding.

Mikolov et al. (2013) [5] proposed two architectures for word embedding based on continuous vector representations of words, implemented as *word2vec* model. It relies on log-linear single-layer neural networks for learning distributed representations: the continuous bag of words (CBOW) and continuous *skip-grams*. The CBOW model performs with higher accuracy for syntactically-oriented tasks and predicts the next word given a context while the skip-gram model is characterised by higher semantic accuracy and used to predict the context given a word [5].

Pennington et al. (2014) [66] suggested the word representations obtained using the statistics of a data corpus, namely the metrics of factorised global word co-occurrence matrices, could yield results better than the prepared with pure window-based methods. The authors introduced *GloVe* (Global Vectors), an unsupervised method able to noticeably outperform the aforementioned techniques [66].

Bojanowski et al. (2017) [74] presented *fastText* model as an extension of the word2vec technique. In this approach, each word is represented as a sum of its character n-grams which allows for preservation of the word's morphology. As a result, words of similar inner structure share similar vector representations. The model was evaluated for word similarity tasks using text data sets in 7 languages, outperforming CBOW and skip-gram models 9 out of 10 times [74].

In the context of this thesis, the most relevant conclusion is that the fastText model, contrary to word2vec and GloVe, can be successfully used in text classification to extract text features of morphologically rich languages, i.e. those that contain important context-sensitive information embedded within the structure of a word in addition to interconnections between words. Since a function name often consists of multiple words bound together and their context varies depending on the arrangement and internal structure of the name, they can be treated as words comparable to those of a morphologically rich language.

3.2 Machine learning in software analysis

Artificial intelligence-powered tools steadily gain popularity across various domains, including the software security industry. Many developers expect to soon adopt or already work with AI tools on daily tasks [75]. It comes as no surprise that software security analysts are interested in improving their productivity too. In fact, there are so many areas for the application of machine learning tools for binary code analysis alone that Xue et al. [6] proposed a taxonomy to guide through them (Fig. 3.1).

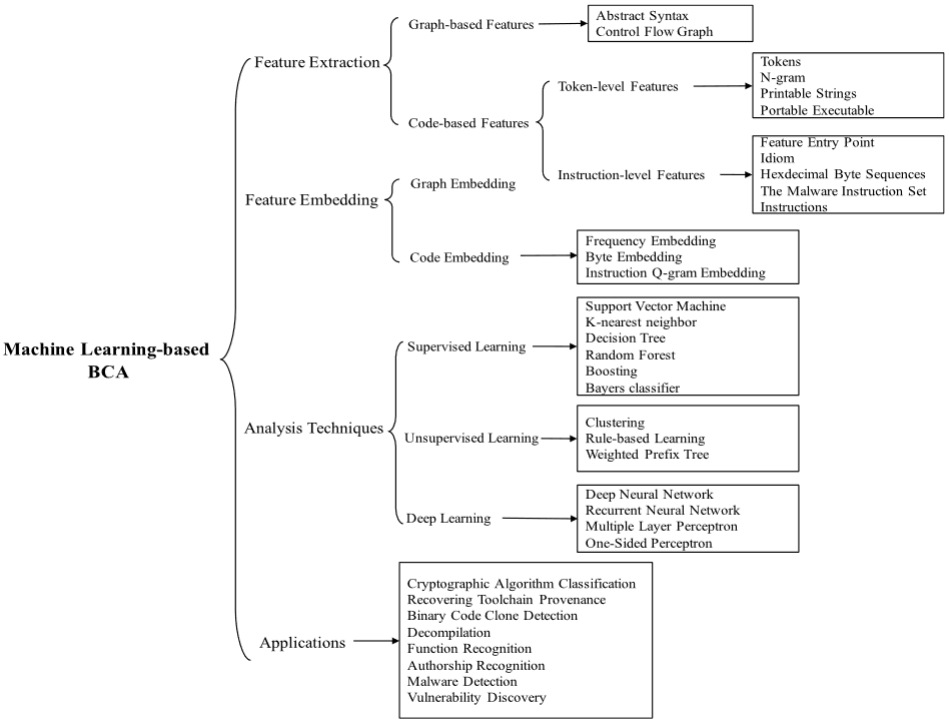

FIGURE 3.1: The taxonomy of machine learning-based BCA framework (IEEE) [6]

From widespread supervised classifiers used for identification of binary architecture [26], through clustering malware prototypes [23] to self-organising incremental neural networks for binary visualisation analysis [22], security-oriented research employs various techniques and paradigms for automated analysis of software. While fully automated malware detection tools or vulnerability scanners [76] are already a reality, there still exist sophisticated cases and tasks that require manual reverse engineering of files by trained application security specialists. Disassemblers and decompilers such as Ghidra or IDA are a vital part of software reverse engineering. While the capabilities of these tools are large, they cannot produce semantically meaningful code out of binaries stripped from debug symbols. In the lack of the original code symbols such as variable and function names, decompilers substitute them with automatically generated labels that do not hold semantic value to a researcher.

When available, reliable code symbols greatly increase the readability of decompiled code and as a consequence, the speed of the binary file reverse engineering process. For that reason, significant research has been conducted to employ machine learning to reconstruct the code symbols of stripped binaries. He et al. [1] proposed Debin as an automated tool to predict variable and function names. The system is based on a lifted representation of disassembly code in the form of Binary Analysis Platform Intermediate Language (BAP IL). With architecture-agnostic BAP IL representation, binary classification of source code-level elements, such as variables, names, or types is performed using extremely randomised trees, a variant of random forest classifier. As the next step, Debin creates a dependency graph to form a conditional random field (CRF) for structured prediction to obtain symbol names. The model uses Maximum a Posteriori (MAP) inference to propagate the predicted symbol names across the constructed dependency graph and lastly encodes the predicted names in DWARF debug symbol format. Debin was trained on 8100 binaries obtained from open-source C codebases. Quality of name predictions expressed as average F1-score reached 64.2% [1].

Lacomis et al. [2] engineered DIRE, a deep learning-based prediction system for identifiers in decompiled code, similar to Debin in general principle. Instead of transforming disassembly to BAP IL, DIRE retrieves a function's abstract syntax tree (AST) and its corresponding decompiled code from the Hex-Rays decompiler. DIRE follows encoder-decoder architecture; it consists of two encoder networks, with each producing two representations of code elements and identifiers: a structural encoder for AST input and a lexical encoder for decompiled code tokens. Lexical encoder utilises a long short-term memory (LSTM) model for text token vectorisation while the structural encoder is a graph-gated neural network (GGNN) for AST node processing. The final step of the encoding process is merging the representations obtained from both encoders into two single representations of linear transformations for the LSTM decoder, which then predicts the names of the supplied code symbols. The full architecture of DIRE's model is shown in Fig. 3.2. DIRE was trained on a large set of over 3 million functions extracted from almost 165,000 open-source C binaries and achieved an overall accuracy of 74.3%. DIRE was also contrasted with Debin in tests with both systems trained on 1% of DIRE's data set - overall accuracy reached 32.2% for DIRE and 2.4% in case of Debin [2].

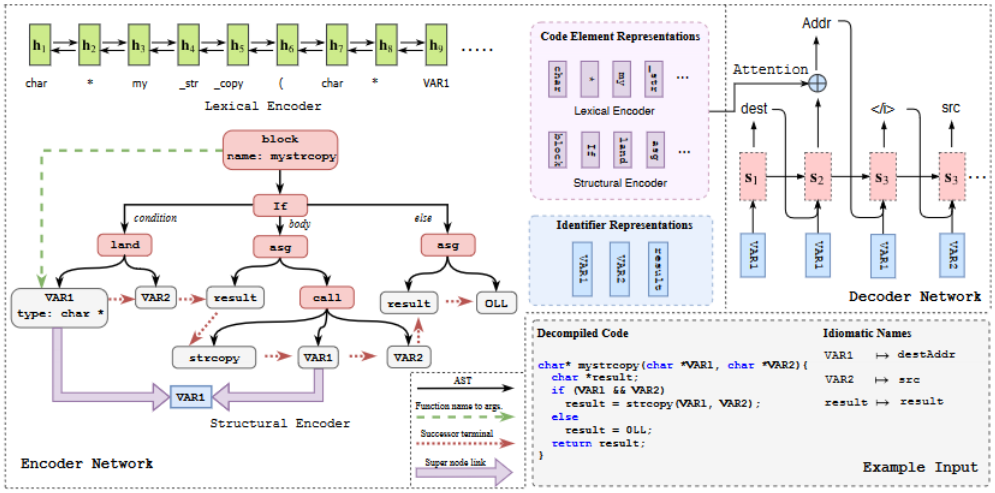

FIGURE 3.2: Overview of DIRE's neural architecture (Lacomis et al.) [2]

While both of the above prediction models can provide impressive results, there are two major drawbacks to this approach. Firstly, incorrect predictions of symbol names introduce false information to decompiled code, which, applied at the rate of over 25%, can mislead a researcher reading the code. Secondly, in case of complex or unusual names not commonly found in open source applications the prediction accuracy can drastically drop as the binaries could not be the typical target of reverse engineering effort [2]. The following chapters of this thesis present an experimental approach to automated symbol extraction from text data using binary classification of text and cross-reference dependencies between binary code and text data, including problem modelling, preparation of data sets for both classifier models, training and evaluation of different machine learning models, and analysis of evaluation results. Similarly to DIRE, the Hex-Rays decompiler was chosen as the underlying source of data for the training data set.

Chapter 4

Methodology

This chapter describes data set preparation process, as well as the approach to classifier training, testing and result evaluation.

4.1 Problem model

The goal of this research is to extract original function names found in text data of PE executables compiled from C or C++ source code and apply the found information to decompiled code. In order to achieve that task two binary classifiers are needed: a text classifier that labels a text fragment (from now called a token) as a function name along with a reference classifier that predicts the token's relation to a potentially described function. Moreover, a text tokenisation method should preserve special function names such as operators, constructors, destructors, or namespace-prefixed names to increase the number of recognised function symbols. As the solution includes the classification of text features, a method for feature extraction needs to be chosen. In addition, a feature set needs to be selected for both classifiers. To label the data set and evaluate classifier results a source of truth about the real function symbol names must be available.

4.1.1 The concept of cross-reference paths

Cross-references are decompiler objects containing information about a function-function or function-data use, for instance in terms of functions the caller has a downward cross-reference on the callee and the callee has an upward cross-reference on the caller. Function-function cross-references form graph-like structures which can be traversed.

Cross-reference path is a working term for a path in a cross-reference graph which in the case of this research always stems from a function to a string. It can include other functions but never data other than the ending string. The length of a cross-reference path is the number of intermediary functions between the two ends, i.e. the length of a cross-reference path with a function directly referencing a string is zero. As cross-reference paths were used as a feature source, given the high complexity of analysed binaries the maximum analysed length of a cross-reference path was limited to 2. For simplicity of the analysis, only paths with upward-only or downward-only directions of cross-references were considered.

4.1.2 Data sources

The general concept of data assembling was to use open-source C and C++ code and compile it with the emission of symbol files. The information about text and function data can be retrieved from a decompiler after automatic analysis of the binaries. The truth about function names can be retrieved from symbol files using a custom parser.

In this research, Microsoft's Visual C++ compiler was chosen as the target with the associated PDB symbol file format. The target platforms of the used applications were x86 and x64 architectures. The prepared data set consisted of 14 binaries and included real-world examples of applications commonly targeted with reverse engineering efforts. IDA Pro was the tool of choice for its state-of-the-art disassembler and decompiler utilities. The target format for the data was SQLite database.

After the source executables were compiled they were decompiled using IDA and saved as IDA Database files (IDB). PDB files were parsed into intermediate JSON file format with a custom parser [77] and then transformed into a relational table representing knowledge about a single binary. IDA plugins were prepared to extract function, string and cross-reference path data from IDB files. Additional Python scripts were prepared for data transformations and labelling. After the manual labelling of samples and ensuring data integrity, the single-binary databases would be merged into a single database file.

4.1.3 Feature engineering

The first classifier's task was to recognise token candidates for a function name. The chosen solution was to only tokenise strings into word-like tokens with prior structuring of function name-like patterns. The text feature was then transformed to a single-word vector representation by a fastText word embedding model trained on the text data from prepared binaries to be finally provided as input to the classifier.

For every function in the binary, its cross-reference paths of a maximum length of 1 were extracted. Additionally, downward paths to strings of length 2 were extracted for experimental purposes. The computational cost of extracting upward paths of length 2 and above was empirically tested to be too high for those connections to be evaluated. The features extracted from cross-reference paths were the direction of the path (upward/downward) and its length (0-2). These features provide information on the call (usage) contexts of a given function that can be reflected inside its name, e.g. indicating a callback function.

Function size metrics were extracted and calculated, namely the numbers of referee and referrer functions, the number of directly referenced strings and the number of assembly instructions. Path classifier's training script would query function size data, cross-reference path features and the referenced token as input features for the model. Such function information lets the model relate common function names to their size and importance inside a binary, e.g. getter and setter functions are often short and have many callers. All seven of the above features were provided as input for the cross-reference path classifiers.

4.1.4 Data labelling

Text tokens were manually labelled positives if the token was found in PDB data, named a common library function or was used in a context unambiguously indicating a function, for instance with a call operator. Cross-reference paths were labelled positives if the target function name was included in the string's text. Then token paths would be generated and the positive function-token path (or paths) manually labelled, with the remaining ones labelled as negatives.

Debug symbol data served as the source of truth for token labelling and was also used to balance the token data set with positives. PDB symbol files contain function names associated with section-relative virtual addresses of the functions. The base virtual address was extracted from binaries and then supplied to the PDB parser which would calculate the correct address and save the function data to a JSON file. Function names in PDB files can be stored in a decorated form [78] which could not be understood by the classifiers - a custom tool for name undecoration was developed.

4.1.5 Data model

A relational database model was developed to unify storage and access to data originating from multiple source files and formats. The database structure which represents data originating from a single executable is presented below (Fig. 4.1). For clarity, only the table relations used by dubRE were highlighted.

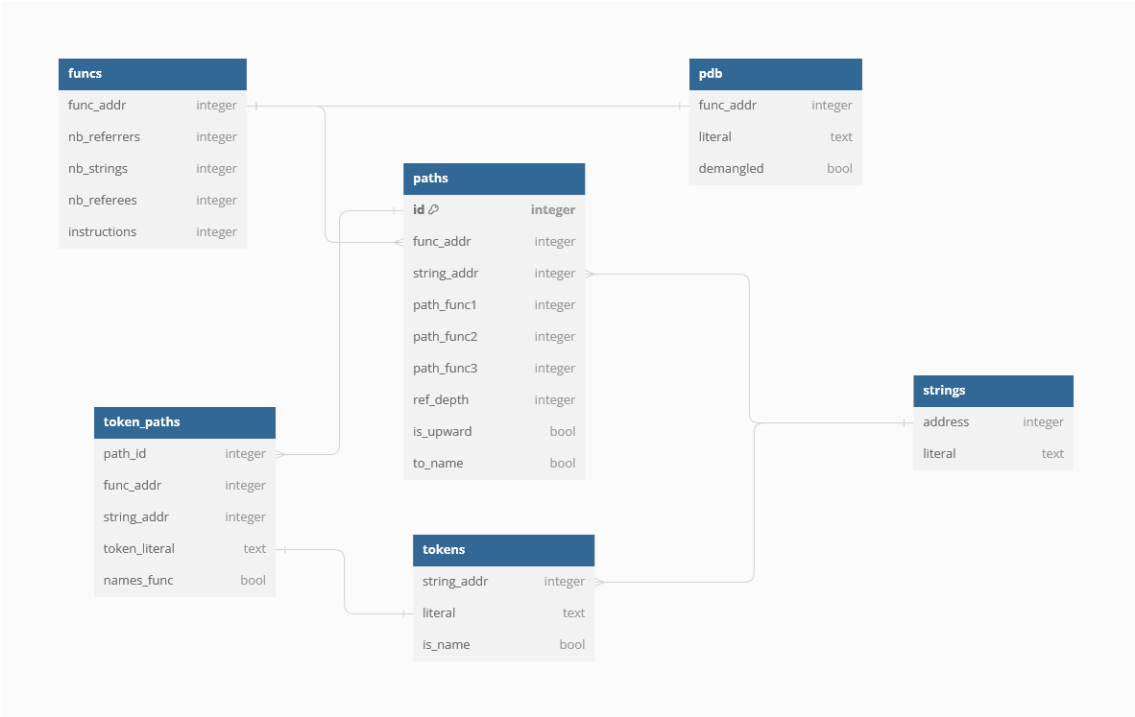

FIGURE 4.1: Binary-scoped database schema

Function debug symbols (*pdb* table) serve as the source of truth for data labelling and are used in the function name data set. Function symbol information relevant to this thesis was extracted from PDB files, namely their names (*literal*) and offsets in the binary (*func_addr*). As symbol names in PDB format can be stored in raw or decorated form [78], an additional column was provided to indicate if a function name was processed by a name demangler (*demangled*).

Function information (*funcs* table) was extracted from IDB files to provide features describing the target functions of cross-reference paths. Every row consists of a function's offset in the binary (*func_addr*) and 4 features for cross-reference path classifiers: the number of all functions that call a given function (*nb_referrers*), the number of all functions called by the function (*nb_referees*), the number of all directly referenced strings (*nb_strings*) and the size of the function measured as the number of its assembly instructions (*instructions*).

String data (*string* table) was extracted from IDB files to produce a set of words for function name classifiers. Every row consists of raw text data (*literal* column) and its offset within the binary (*address*).

Text tokens (*tokens* table) are unique words obtained from text data. The names of operators, constructors, destructors and namespaces were structured to preserve them before tokenisation. Additionally, file and directory paths were structured to reduce the number of tokens in the data set. A token row includes the offset of the original strings (*string_addr*), the token text (*literal*) and a binary label that identifies the text as a function name (*is_name*), assigned manually. Token text is used as the only feature for function name classifiers, vectorised by fastText word-embedding model during feature extraction (see 4.3).

Function-string cross-reference paths (*paths* table) contain cross-reference paths starting at a given function and ending at a string. Only the paths within the adopted cross-reference graph proximity limits were extracted and evaluated (see 4.1.1). A cross-reference path is defined by its unique identifier (*id*), a target function (*func_addr*), up to 2 intermediate callees or 1 intermediate caller (*path_func1* and *path_func2*; *path_func3* was unused) and the target string (*string_addr*). Features extracted from cross-reference paths were their lengths (measured as the number of intermediate functions - *ref_depth*) and the direction of the path (upward or downward - *is_upward*). Each row was manually labelled (*to_name*) to indicate if the string contains the target function's name to partially automate the labelling process of function-token cross-reference paths.

Function-token cross-reference paths (*token_paths* table) contain all the text candidates (tokens) for a function name narrowed by the given function's cross-reference paths. They contain a unique identifier of a cross-reference path (*path_id*), a token from the referenced target string (*token_literal*), and a label indicating if the token is the function's name (*is_name*). Additionally, redundant function and string addresses were supplied from *paths* table for ease of manual labelling process.

The data model was extended to enable the storage of data from multiple binaries. The diagram below describes the database created from all single-binary databases (Fig. 4.2). A column containing the name of the source of data was added to every table (*binary*) - all previous primary keys were extended. In addition, a new unique identifier *id* was introduced for *paths* table to resolve conflicts across different databases - conflicting identifiers were renamed to *local_id*.

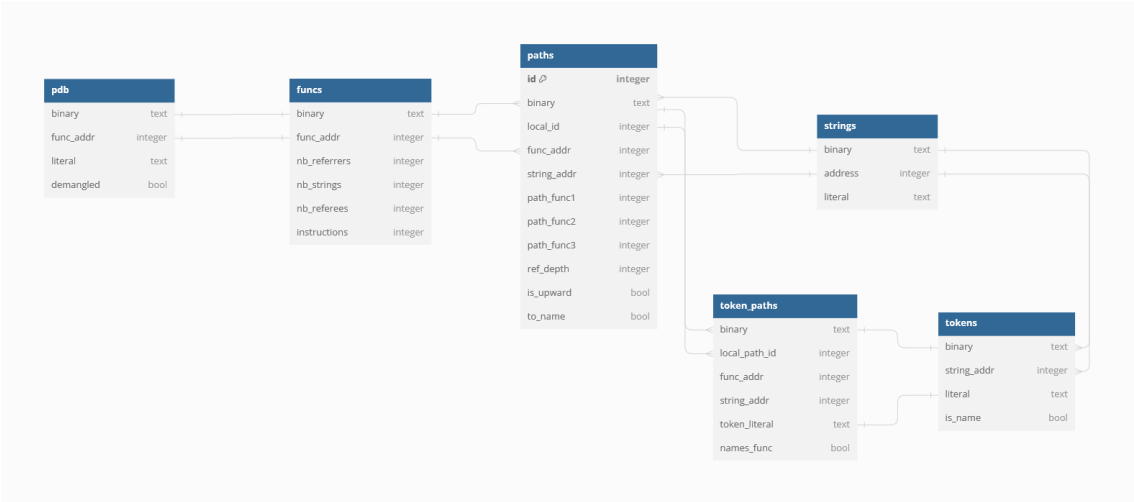

FIGURE 4.2: Aggregated database schema

4.1.6 Data set characteristics

The research was performed on a set of 14 binaries implemented in C and C++ programming languages, compiled from 10 GitHub repositories. The projects chosen for the data set were video games, game engines, compression tools, web servers and user-defined libraries. The file size of the compiled binaries varied from 0.5MB up to 200 MB.

The proportion of function name samples in the data set is presented in Table 4.1. The number of negative unique tokens was significantly larger than that of positives, however the distribution was evened with PDB data. The resulting set consisted of nearly 120,000 samples.

Labelled samples	68 935
Labelled positives	9 589
Labelled negatives	59 346
Balancing PDB positives	49 757
Total	118 692

TABLE 4.1: Statistics of function name token set

The proportion of function-to-string cross-reference path samples in the data set is presented in Table 4.2. The cross-reference path set was extremely unbalanced - the positives constituted only 0.32% of the data. *Positive functions* refers to the number of functions that had at least 1 positive cross-reference path.

Labelled samples	665 518
Labelled positives	2 188
Labelled negatives	663 330
Positive functions	1 994

TABLE 4.2: Statistics of function-string cross-reference path set

As a consequence of function-to-string cross-reference path imbalance, the percentage of positive function-token samples is even smaller (0.25%) due to strings consisting of multiple tokens (Table 4.3).

Labelled samples	828 526
Labelled positives	2 071
Labelled negatives	826 456
Positive functions	1 994

Table 4.3: Statistics of function-token cross-reference path set

4.2 Classifier parameters

All evaluated classifiers are *scikit-learn* implementations. No hyperparameter tuning was performed - all models were evaluated with default settings defined by the classifier implementations, used for both kinds of classifiers with the exception of the logistic regression paths classifier, for which the default limit of iterations was increased due to warnings encountered during training. This approach allows for a thorough evaluation of the classifiers; a comparison of fine-tuned models could introduce a bias, resulting in classifiers outperforming the competition only due to their parameter value selection during the optimisation process.

4.2.1 Gaussian Naive Bayes

Gaussian Naive Bayes classifiers were trained with the default parameters presented below (Table 4.4).

Parameter description	**Value**
Variance smoothing	1E-9
Prior class probabilities	None

Table 4.4: Parameters used for Gaussian Naive Bayes classifiers [7]

4.2.2 K-Nearest neighbors

K-Nearest neighbors classifiers were trained in 3 variants (1-NN, 3-NN and 5-NN). Each variant used the same default values for all other parameters to measure how the number of neighbors affects the results (Table 4.5).

Parameter description	Value
Number of neighbors (k)	1/3/5
Algorithm	Chosen dynamically
Sample weights	Equal
Power parameter (p)	2
Leaf size	30
Distance metric type	Minkowski
Metric parameters	None

Table 4.5: Parameters used for k-nearest neighbors classifiers [8]

4.2.3 Decision trees

Decision tree classifiers were trained with the default parameters presented below (Table 4.6). Gini impurity function was used to measure the quality of tree node splits.

Parameter description	Value
Split quality estimation function	Gini impurity
Maximum tree depth	Unlimited
Minimum samples for a node split	2
Minimum samples at a leaf	1
Sample weight	Equal
Splitting strategy	Best splits
Number of features considered in splitting	Unlimited
Class weights	Equal
Cost-Complexity Pruning	Disabled
Maximum number of leaves	Unlimited
Bootstrap samples for tree building	Yes
Minimal impurity decrease for node splits	0

Table 4.6: Parameters used for decision tree classifiers [9]

4.2.4 Random forest

Random forest classifier was trained with the default number of 100 decision trees; similarly to the decision tree classifiers, the Gini impurity function was used to measure the quality of tree node splits. The parameters used in the training of both random forest classifiers are listed below (Table 4.7).

Parameter description	**Value**
Number of decision trees	100
Split quality estimation function	Gini impurity
Minimum samples at a leaf	1
Minimum samples for a node split	2
Sample weight	Equal
Number of features considered in splitting	$\sqrt{nb\_features}$
Maximum number of leaves	Unlimited
Minimal impurity decrease for node splits	0
Bootstrap samples for tree building	Yes
Class weights	Equal
Cost-Complexity Pruning	Disabled

Table 4.7: Parameters used for random forest classifiers [10]

4.2.5 Adaptive boosting

AdaBoost classifiers were trained with the default parameters of 50 decision tree base estimators. The boosting algorithm, along with its learning rate, was not modified. All the parameters used for both AdaBoost classifiers are presented below (Table 4.8).

Parameter description	**Value**
Weak classifier type	Decision trees
Number of weak classifiers	50
Learning rate	1
Boosting algorithm	SAMME.R

Table 4.8: Parameters used for adaptive boosting classifiers [11]

4.2.6 Linear SVM

Linear SVM classifiers were trained with the default parameters presented below (Table 4.9).

Parameter description	**Value**
Penalty type	L2
Loss function	Squared hinge
Optimisation problem formulation	Primal
Tolerance for stopping criteria	0.0001
C	1
Multi-class strategy	OVR
Intercept calculation	Enabled
Intercept scaling	1
Class weights	Equal
Maximum number of iterations	1 000

Table 4.9: Parameters used for linear SVM classifiers [12]

4.2.7 Logistic regression

Logistic regression classifiers were trained with the default parameters presented below (Table 4.10). The maximum number of iterations was exceptionally increased for the paths classifier from 100 to 200 due to warnings about the exceeded number of maximum iterations encountered during training.

Parameter description	**Value**
Penalty type	L2
Optimisation problem formulation	Primal
Tolerance for stopping criteria	0.0001
C	1
Multi-class strategy	OVR
Intercept calculation	Enabled
Intercept scaling	1
Class weights	Equal
Solver algorithm	LBFGS
Maximum number of iterations	100*
Warm start	Disabled

Table 4.10: Parameters used for logistic regression classifiers [13]

4.2.8 Multi-layer perceptron

Multi-layer perceptron classifiers were trained with the default parameters presented below (Table 4.11). Both classifiers comprised of 1 input layer, 1 hidden layer of 100 neurons, and 1 output layer.

Parameter description	**Value**
Number of hidden layers	1
Hidden layer size	100
Activation function	rectified linear unit
Solver algorithm	ADAM
α (L2 regularisation)	0.0001
Batch size	200
Learning rate	0.001 (constant)
Optimisation tolerance	0.0001
Intercept scaling	1
Class weights	Equal
Solver algorithm	LBFGS
Maximum number of iterations	200
Shuffle samples per iteration	Enabled
Warm start	Disabled
Early stopping	Disabled
β_1 (ADAM)	0.9
β_2 (ADAM)	0.999
ϵ (ADAM)	1E-8
Maximum iterations without tolerance (ADAM)	10

TABLE 4.11: Parameters used for MLP classifiers [14]

4.3 Classifier training and testing approach

Data sets for both classifiers were divided into training and testing sets with 80% of samples used for training and 20% for testing. The splitting method preserved the proportions of positive and negative samples. While the token data set could be balanced with debug symbol-supplied function name positives, there was no additional source of positives for the function-to-token cross-reference path set. Due to extreme class imbalance accuracy should not be considered as a valid measure for path classifiers.

To perform feature extraction of the text token feature, *fastText* word embedder was chosen. The motivation behind this choice was that contrary to many word embedding models, e.g. *word2vec* or *GloVe*, the fastText model is based on character n-grams rather than word-grained architectures like the popular techniques based on continuous bag of words (CBOW) or skip-grams [5] [74]. This makes it suitable for the task of feature extraction for morphologically rich function name classification as the model is able to spot sub-word nuances in single-word data [74]. The embedder's model was trained with training token data and used for every classifier training session. All tokens are represented as single-element word vectors. Before classifier fitting features were normalised

by removing the mean and scaling to unit variance. The parameters used to train the fastText model are presented below (Table 4.12).

Parameter	**Value**
Model	CBOW
Vector size	1
Initial learning rate	0.025
Context window size	3
Minimum number of word occurences	1
Loss function	NS
Threshold for downsampling higher-frequency words	0.001
Number of negative words to sample	5
Number of iterations	5
Vocabulary frequency sorting	Descending
Number of threads	12
Minimum length of n-grams	3
Maximum length of n-grams	6
Number of buckets for n-gram hashing	2000000

Table 4.12: Parameters used for fastText model training [15]

5-fold cross-validation was performed before every classifier training with estimated accuracy and standard deviation for function name classifiers and F1 score and standard deviation for cross-reference paths classifiers. For both models accuracy, precision, recall, and F1 score were calculated. It is worth noting that accuracy should not be taken into consideration when analysing results of cross-reference path classifiers as the data set contained less than 1% of positively labelled samples.

The best-performing 3 classifiers of each type were tested together in all combinations in a data processing pipeline where data was first labelled by the function name classifier, then based on the prediction samples were rejected or passed for classification to the cross-reference path classifier. The data utilised for testing was the set used for path classifier testing. Due to the text token feature being present in both model types and both models being evaluated at once, the testing set was inspected for inclusion of text data from function name classifiers' training set. The text tokens shared with the training set for function name classifiers constituted 84.25% of the processing pipeline's testing set. Such a large similarity to the training set likely caused function name classifiers of the pipeline to perform significantly better than they would on never-before-seen data. As a consequence, the overall performance of the pipeline in the test was improved, thus the results of the joint classifier pipeline should be interpreted only as a rough and optimistic estimate of how the two models could perform together in a real-world application scenario of automated labelling for decompiled code. To obtain reliable results, the test should be repeated using additional data largely independent of both classifiers' training sets, which unfortunately was not available during the research.

Chapter 5

Results

This chapter presents the results of classifier evaluation. Both types of classifiers were tested independently. Three best-performing classifiers of each type were chosen and evaluated together in a data processing pipeline to simulate the target environment.

5.1 Function name classifiers

Function name classifiers aim to learn the representations of function names and choose function name candidates from the text data found in binaries. Single-word vector representation was used as the only input feature.

5.1.1 Cross-validation results

The results of 5-fold cross-validation for function name classifiers which was performed during training are presented below (Table 5.1). Since the data set was equalised in terms of class occurrences, average accuracy and its standard deviation (σ) are presented to provide insight into prediction quality.

	Accuracy	σ
AdaBoost	0.616	0.004
Decision tree	0.590	0.003
Gaussian Naive Bayes	0.615	0.004
1-NN	0.591	0.003
3-NN	0.589	0.003
5-NN	0.588	0.004
Logistic regression	0.503	0.002
Linear SVM	0.503	0.002
Neural network	0.617	0.004
Random forest	0.591	0.003

Table 5.1: Results of cross-validation for function name classifiers

5.1.2 Test results

The results of function name classifier performance validation with an independent testing data set are presented below (Table 5.2).

Classifier	TP	TN	FP	FN	Accuracy	Precision	Recall	F1
AdaBoost	9 434	5 254	6 576	2 475	0.619	0.589	0.792	0.676
Decision tree	6 873	7 246	4 584	5 036	0.595	0.600	0.577	0.588
Gaussian Naive Bayes	10 188	4 496	7 334	1 721	0.619	0.581	0.855	0.692
1-NN	6 995	7 164	4 666	4 914	0.596	0.600	0.587	0.594
3-NN	7 135	6 930	4 900	4 774	0.592	0.593	0.599	0.596
5-NN	7 263	6 897	4 933	4 646	0.596	0.596	0.610	0.603
Logistic regression	5 646	6 267	5 563	6 263	0.502	0.504	0.474	0.488
Linear SVM	5 647	6 264	5 566	6 262	0.502	0.504	0.474	0.488
Neural network	9 323	5 356	6 474	2 586	0.618	0.590	0.783	0.673
Random forest	6 983	7 154	4 676	4 926	0.596	0.599	0.586	0.593

Table 5.2: Test results for function name classifiers

5.2 Cross-reference path classifiers

Function-to-token cross-reference path classifier is responsible for the assessment of a function name occurrence in the context of a function's data reference inside of a binary. The prediction provides information on the relevance of a function name to function code in a decompiled binary. A function can be renamed with the found name in case of a positive prediction. The feature vector used as input included a vectorised function name, function-token path length and direction, the function's size in instructions and direct cross-reference counts.

5.2.1 Cross-validation results

The results of 5-fold cross-validation for function-token cross-reference paths classifiers which were performed during training are presented below (Table 5.3). Since the data set by its nature is imbalanced in terms of class occurrences, the average F1 score and its standard deviation (σ) are presented to provide insight into prediction quality.

	F1	σ
AdaBoost	0.012	0.011
Decision tree	0.573	0.030
Gaussian Naive Bayes	0.010	0.018
1-NN	0.518	0.020
3-NN	0.167	0.022
5-NN	0.144	0.031
Logistic regression	0.000	0.000
Linear SVM	0.000	0.000
Neural network	0.000	0.000
Random forest	0.616	0.033

TABLE 5.3: Results of cross-validation for cross-reference path classifiers

5.2.2 Test results

The results of cross-reference path classifier performance validation with an independent testing data set are presented below (Table 5.4). Although accuracy was provided for the sake of result completeness, the metric is not relevant for result analysis as the data set was highly imbalanced in favor of negative samples (99.75%).

Classifier	TP	TN	FP	FN	Accuracy	Precision	Recall	F1
AdaBoost	5	165 306	0	395	0.998	1.000	0.013	0.025
Decision tree	252	165 168	138	148	0.998	0.646	0.630	0.638
Gaussian Naive Bayes	0	165 301	5	400	0.998	0.000	0.000	0.000
1-NN	237	165 178	128	163	0.998	0.649	0.593	0.620
3-NN	50	165 156	150	350	0.997	0.250	0.125	0.167
5-NN	42	165 300	6	358	0.998	0.875	0.105	0.188
Logistic regression	0	165 306	0	400	0.998	0.000	0.000	0.000
Linear SVM	0	165 306	0	400	0.998	0.000	0.000	0.000
Neural network	0	165 306	0	400	0.998	0.000	0.000	0.000
Random forest	232	165 280	26	168	0.999	0.899	0.580	0.705

TABLE 5.4: Test results for cross-reference path classifiers

5.3 Multi-model pipeline

To envision if the two classifiers can complement each other in a two-step binary classification process needed to label decompiled functions, an experiment was performed where the data is first rejected or passed for further classification by the function name classifier and in the case of the latter labelled by cross-reference path prediction model.

Test results for cooperation between the best-performing function name and cross-reference path classifiers on the cross-reference path testing data set are presented below (Table 5.5). The results do not fully reflect the expected performance on independent data as 84.25% of the text tokens in the pipeline's testing set were found in the training data set of token name classifiers (refer to 4.3). Accuracy was not provided as it should not be considered a reliable metric in result analysis for reasons stated in 5.2.2.

Name classifier	Path classifier	TP	TN	FP	FN	Precision	Recall	F1
AdaBoost	Decision tree	173	165 218	88	227	0.663	0.433	0.523
AdaBoost	Random forest	158	165 286	20	242	0.888	0.395	0.547
AdaBoost	1-NN	161	165 238	68	239	0.703	0.403	0.512
Naive Bayes	Decision tree	194	165 206	100	206	0.660	0.485	0.559
Naive Bayes	Random forest	179	165 283	23	221	0.886	0.448	0.595
Naive Bayes	1-NN	182	165 219	87	218	0.677	0.455	0.544
Neural network	Decision tree	170	165 218	88	230	0.659	0.425	0.517
Neural network	Random forest	155	165 286	20	245	0.886	0.388	0.539
Neural network	1-NN	158	165 240	66	242	0.705	0.395	0.506

Table 5.5: Pipeline evaluation results

Chapter 6

Discussion

This chapter aims to interpret the obtained results and discuss the limitations of the current solution, possible improvements, and areas of future research.

6.1 Classifier evaluation

Function name classifiers achieved accuracy ranging from 50% to 62%. While the results of linear classifiers such as logistic regression and support vector machines are comparable to random guessing, a noticeable difference of approximately 10 percentage points can be observed for other models. The best-performing classifiers are adaptive boosting ensemble of decision trees, Gaussian naive Bayes network and the neural network with similar accuracy and F1 scores. The fact that no classifier exceeded 65% of accuracy may indicate errors in feature engineering or feature extraction as the classifiers were trained with only one text feature, or insufficient sample size to recognise patterns of function names in natural language. Improving the training data set and parameter choice for the fastText word embedding model, changing the text tokenisation technique to return more than a single word from the embedder, increasing the training data set size and fine-tuning the classifiers - all could positively affect the results.

The case of cross-reference path classifier results was quite different from function name classifiers due to significantly larger size and low rate of positive samples. Naive Bayes and linear classifiers did not make any positive predictions which was likely caused by the low number of positives in the data set. Random forest, decision tree, and 1-nearest neighbor models outperformed other classifiers, with random forest reaching an F1 score surpassing 0.7. An important metric for cross-reference classifiers is precision as it measures the model's accuracy for positive predictions. Maximising precision aligns with the goal of reducing the number of false positives that can mislead reverse engineers, improving the trust for positive predictions. Despite the decision tree classifier correctly labelling the highest number of function names, the cost of introducing a significant amount of false positive renamings is considerably higher compared to the more restrained random forest classifier. A surprising observation is the difference in results between the two ensemble classifiers, both incorporating decision trees as the base estimator type in the test. The results indicate random forest to be the most fitting out of all evaluated cross-reference path classifiers. An attempt was made to assess the cooperation of the best function name and cross reference paths classifiers. To no surprise, the best-performing pair were the highest-scoring classifiers, i.e. Gaussian naive Bayes for function names and random forest for cross-reference paths. However, due to the test set data being in large part known to the function name clas-

sifiers, these results can only serve as a hypothesis until validated with a fully independent test set.

The research did not focus on optimisation of the classifier performance but rather served as a proof of concept for a debug symbol extraction method based on binary classification. The best-performing classifiers, Gaussian Naive Bayes for function names and random forest for cross-reference paths, complimented each other in a desirable manner - the first had a 'liberal' tendency indicated by high recall value (0.855) and the majority of labels being positive, while the latter could be described as more 'careful' in predictions, characterised by high precision (approximately 0.9) out of the best classifiers. The resulting combination of the two could identify nearly half of all available function names (44.75%) with minimised false discovery rate (0.114) [79]. Such an approach could provide a significant portion of automatic function name assignments that could be trusted by reverse engineers. Considering the usage of experimental feature vectors and the lack of any hyperparameter tuning, the results obtained with presented classifiers could reasonably be called promising and worthy of future research, being mindful of the mentioned threat to validity for the data processing pipeline test.

6.2 Future work

The achieved results seem sufficient for a practical application of the presented name-extraction technique aiming to enrich the raw output code of a decompiler, although they are far from perfect. As an experimental case study, this research did not evaluate all of the available models, e.g. graph neural networks (GNNs) could be used for cross-reference graph representation learning as a similar solution was adopted by Lacomis et al. for AST processing [2]. Furthermore, arbitrary approaches to text structuring, tokenisation, and representation were based on the assumption of analogy between complex function names and morphologically rich words of natural languages. Evaluation of new models using different word representations could bring valuable insight into the pool of well-performing classification techniques. Moreover, none of the presented classifiers were optimised in terms of training hyperparameters, which could significantly increase the individual performance of the models, including the word embedding fastText model responsible for the representation of text features.

Besides the enhancements to classification methods, the limitations of the data prepared for the research could be lifted to reach higher and more consistent classifier performance, as well as more generalised and accurate model validation. Firstly, a testing set of data for joint classifier pipeline, independent of both training sets, could be prepared to perform reliable evaluation. Secondly, feature vectors used for model training were arbitrarily selected as highly experimental thus some features may contribute noise to the representation, especially in the case of cross-reference path classifiers. Additionally, custom sanitisation and decorated name translation tools were developed and used to process the unstructured text data found in binaries, which could decrease the potential quality of the available information. Tests for various feature combinations along with improvements to the text tokenisation process could appear favorable for the overall quality of the data set. As the data was manually labelled, a verification of class assignment correctness could further increase the quality of the data by removing human errors. Lastly, the data set was assembled from a very limited number of binaries. What is more, the extracted samples happened not to contain text data due to type mismatches produced by the decompiler, which artificially boosts the sample count while providing no value for the classifiers. Expanding the data set with new samples from function name-rich binaries, as well as removing false samples, could improve

the results and increase the reliability of performance validation.

Chapter 7

Conclusion

This thesis studied classification-driven symbol extraction in binary file reverse engineering. The main purpose of this work was to conduct research on the topic of code symbol name recovery in decompiled code, formalised as three questions:

RQ1: What are the current methods and possibilities for automated semantic labelling of decompiled code?
RQ2: What are the downsides of state-of-the-art techniques for symbol information restoration in stripped binaries?
RQ3: Is automated extraction-based function naming possible to achieve in debug symbol-stripped binary files using machine learning techniques?

Firstly, the main aspects of software reverse engineering were covered along with its applications in security-oriented software analysis. The field of artificial intelligence was reviewed to better understand its subdomains, as well as supervised machine learning techniques and the importance of natural language processing methods in text classification. A review of the domain literature was conducted to assess the current methods of text representation for the tasks of natural language processing. Then, an overview of research on ML applications in software analysis was presented, followed by the case study of automated labelling of decompiled code with semantic names. Two recent systems were reviewed as state-of-the-art tools enabling the recovery of debug symbols of stripped binaries, namely Debin (He et al.) [1] and DIRE (Lacomis et al.) [2], to answer the first research question (*RQ1*).

While different in internal architecture, both discussed solutions were prediction systems based on symbol name synthesis predicated on universal external knowledge. As stated by Lacomis et al. [2], the 74.3% prediction accuracy of DIRE could worsen significantly in cases of software not found in the training data set, which relied solely on open-source applications. In addition, both tools make name predictions for the entirety of decompiled code that could result in a significant amount of mislabeled symbols and confuse reverse engineers. Having answered the second research question (*RQ2*), an experimental machine learning environment was prepared to test a data extraction-based method, addressing both issues of the above approach.

Lastly, research was conducted to evaluate the efficiency of the classification of function names extracted from text data found in executable files. Two experimental types of classifiers were presented, utilising the concepts of natural language processing and dependencies between the code and static data of an executable, also known as cross-references. An experimental data set was extracted from open-source C/C++ applications, transformed and labelled. Ten different classifier models were trained and evaluated for comparison. The most effective types of each classifier were tested in a data processing pipeline simulating a real prediction environment. The achieved results reached up to 44.75% of correctly extracted function names while preserving the false discovery rate (FDR) of 11.4% using Gaussian Naive Bayes as the function name classifier and random forest as the cross-reference dependency classification method. In conclusion, the presented technique proved the hypothesis suggested by the last research question (*RQ3*), showing that it can automatically improve decompiled code of any origin in an error-avoiding manner and thus be useful in the process of static analysis. The results of the research, data sets, code and machine learning models were contributed to open source software as *dubRE* [80] to encourage further research within the domain and allow for reproduction of the experiment.

Bibliography

[1] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1667–1680, New York, NY, USA, 2018. ACM.

[2] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A neural approach to decompiled identifier renaming. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, California, November 2019.

[3] scikit-learn - Decision tree visualisation. [on-line] `https://scikit-learn.org/stable/auto_examples/tree/plot_iris_dtc.html`. Accessed: August 30th 2023.

[4] scikit-learn - Multi-layer Perceptron. [on-line] `https://scikit-learn.org/stable/modules/neural_networks_supervised.html#multi-layer-perceptron`. Accessed: August 31st 2023.

[5] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[6] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access*, 7:65889–65912, 2019.

[7] scikit-learn - Gaussian Naive Bayes classifier. [on-line] `https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html`. Accessed: August 31st 2023.

[8] scikit-learn - k-Nearest Neighbors classifier. [on-line] `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`. Accessed: August 31st 2023.

[9] scikit-learn - Decision tree classifier. [on-line] `https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html`. Accessed: August 31st 2023.

[10] scikit-learn - Random forest classifier. [on-line] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`. Accessed: August 31st 2023.

[11] scikit-learn - AdaBoost classifier. [on-line] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html`. Accessed: August 31st 2023.

[12] scikit-learn - Linear SVC. [on-line] `https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC`. Accessed: August 31st 2023.

[13] scikit-learn - Logistic Regression classifier. [on-line] `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`. Accessed: August 31st 2023.

[14] scikit-learn - Multi-Layer Perceptron classifier. [on-line] `https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html`. Accessed: August 31st 2023.

[15] FastText Model - hyperparameters (gensim). [on-line] `https://radimrehurek.com/gensim/auto_examples/tutorials/run_fasttext.html#training-hyperparameters`. Accessed: August 31st 2023.

[16] European cybersecurity market forecast. [on-line] `https://www.marketdataforecast.com/market-reports/europe-cyber-security-market`. Accessed: August 31st 2023.

[17] Thomas Ball. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):216–234, 1999.

[18] Michael N Gagnon, Stephen Taylor, and Anup K Ghosh. Software protection through anti-debugging. *IEEE Security & Privacy*, 5(3):82–84, 2007.

[19] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[20] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. Binsign: fingerprinting binary functions to support automated analysis of code executables. In *ICT Systems Security and Privacy Protection: 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings 32*, pages 341–355. Springer, 2017.

[21] KyoungSoo Han, Jae Hyun Lim, and Eul Gyu Im. Malware analysis method using visualization of binary files. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, pages 317–321. 2013.

[22] Irina Baptista, Stavros Shiaeles, and Nicholas Kolokotronis. A novel malware detection system based on machine learning and binary visualization. In *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6. IEEE, 2019.

[23] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of computer security*, 19(4):639–668, 2011.

[24] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*. Citeseer, 2015.

[25] Alec Helbling, Mansi Phute, Matthew Hull, and Duen Horng Chau. Llm self defense: By self examination, llms know they are being tricked. *arXiv preprint arXiv:2308.07308*, 2023.

[26] Sami Kairajärvi. Automatic identification of architecture and endianness using binary file contents. 2019.

[27] Tom M Mitchell. Machine learning, 1997.

[28] Christian Robert. Machine learning, a probabilistic perspective, 2014.

[29] Yasir Shafi Reshi and Rafi Ahmad Khan. Creating business intelligence through machine learning: An effective business decision making tool. In *Information and Knowledge Management*, volume 4, pages 65–75, 2014.

[30] Igor Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine*, 23(1):89–109, 2001.

[31] Hironobu Fujiyoshi, Tsubasa Hirakawa, and Takayoshi Yamashita. Deep learning-based image recognition for autonomous driving. *IATSS research*, 43(4):244–252, 2019.

[32] Daya C Wimalasuriya and Dejing Dou. Ontology-based information extraction: An introduction and a survey of current approaches, 2010.

[33] Anna Stavrianou, Caroline Brun, Tomi Silander, and Claude Roux. Nlp-based feature extraction for automated tweet classification. *Interactions between Data Mining and Natural Language Processing*, 145, 2014.

[34] Rohan Bavishi, Michael Pradel, and Koushik Sen. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*, 2018.

[35] Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. 2015.

[36] Daniel Berrar et al. Cross-validation., 2019.

[37] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, and Jonathan Taylor. Unsupervised learning. In *An Introduction to Statistical Learning: with Applications in Python*, pages 503–556. Springer, 2023.

[38] Zoubin Ghahramani. Unsupervised learning. In *Summer school on machine learning*, pages 72–112. Springer, 2003.

[39] Hagai Attias. Independent factor analysis. *Neural computation*, 11(4):803–851, 1999.

[40] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[41] Chris Gaskett et al. Q-learning for robot control. 2002.

[42] Purvag G Patel, Norman Carver, and Shahram Rahimi. Tuning computer gaming agents using q-learning. In *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 581–588. IEEE, 2011.

[43] Xinwei Chen, Marlin W Ulmer, and Barrett W Thomas. Deep q-learning for same-day delivery with vehicles and drones. *European Journal of Operational Research*, 298(3):939–952, 2022.

[44] Amalia Luque, Alejandro Carrasco, Alejandro Martín, and Ana de Las Heras. The impact of class imbalance in classification performance metrics based on the binary confusion matrix. *Pattern Recognition*, 91:216–231, 2019.

[45] Matthew D Lieberman and William A Cunningham. Type i and type ii error concerns in fmri research: re-balancing the scale. *Social cognitive and affective neuroscience*, 4(4):423–428, 2009.

[46] Ronei M Moraes and Liliane S Machado. Gaussian naive bayes for online training assessment in virtual reality-based simulators. *Mathware & Soft Computing*, 16(2):123–132, 2009.

[47] Todd Andrew Stephenson. An introduction to bayesian network theory and usage. Technical report, Idiap, 2000.

[48] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine learning*, 29:131–163, 1997.

[49] scikit-learn - Naive Bayes. [on-line] `https://scikit-learn.org/stable/modules/naive_bayes.html`. Accessed: August 30th 2023.

[50] Wangchao Lou, Xiaoqing Wang, Fan Chen, Yixiao Chen, Bo Jiang, and Hua Zhang. Sequence based prediction of dna-binding proteins based on hybrid feature selection using random forest and gaussian naive bayes. *PloS one*, 9(1):e86703, 2014.

[51] Carl Kingsford and Steven L Salzberg. What are decision trees? *Nature biotechnology*, 26(9):1011–1013, 2008.

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

[53] Alexandre Hudon, Mélissa Beaudoin, Kingsada Phraxayavong, Laura Dellazizzo, Stéphane Potvin, and Alexandre Dumais. Implementation of a machine learning algorithm for automated thematic annotations in avatar: A linear support vector classifier approach. *Health Informatics Journal*, 28(4):14604582221142442, 2022.

[54] Todd G Nick and Kathleen M Campbell. Logistic regression. *Topics in biostatistics*, pages 273–301, 2007.

[55] Gongde Guo, Hui Wang, David Bell, Yaxin Bi, and Kieran Greer. Knn model-based approach in classification. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings*, pages 986–996. Springer, 2003.

[56] Dietrich Wettschereck, David W Aha, and Takao Mohri. A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. *Artificial Intelligence Review*, 11:273–314, 1997.

[57] Akhlaqur Rahman and Sumaira Tasnim. Ensemble classifiers and their applications: a review. *arXiv preprint arXiv:1404.4088*, 2014.

[58] scikit-learn - Random forest ensemble. [on-line] `https://scikit-learn.org/stable/modules/ensemble.html#random-forests`. Accessed: August 31st 2023.

[59] Gérard Biau and Erwan Scornet. A random forest guided tour. *Test*, 25:197–227, 2016.

[60] scikit-learn - AdaBoost ensemble. [on-line] `https://scikit-learn.org/stable/modules/ensemble.html#adaboost`. Accessed: August 31st 2023.

[61] Raúl Rojas et al. Adaboost and the super bowl of classifiers a tutorial introduction to adaptive boosting. *Freie University, Berlin, Tech. Rep*, 1(1):1–6, 2009.

[62] KR1442 Chowdhary and KR Chowdhary. Natural language processing. *Fundamentals of artificial intelligence*, pages 603–649, 2020.

[63] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5):544–551, 2011.

[64] Diksha Khurana, Aditya Koli, Kiran Khatter, and Sukhdev Singh. Natural language processing: State of the art, current trends and challenges. *Multimedia tools and applications*, 82(3):3713–3744, 2023.

[65] Siwei Lai, Kang Liu, Shizhu He, and Jun Zhao. How to generate a good word embedding. *IEEE Intelligent Systems*, 31(6):5–14, 2016.

[66] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[67] Maarten van Gompel and Antal van den Bosch. Efficient n-gram, skipgram and flexgram modelling with colibri core. *Journal of Open Research Software*, Aug 2016.

[68] Ray Oshikawa, Jing Qian, and William Yang Wang. A survey on natural language processing for fake news detection. *arXiv preprint arXiv:1811.00770*, 2018.

[69] Walaa Medhat, Ahmed Hassan, and Hoda Korashy. Sentiment analysis algorithms and applications: A survey. *Ain Shams engineering journal*, 5(4):1093–1113, 2014.

[70] Aurangzeb Khan, Baharum Baharudin, Lam Hong Lee, and Khairullah Khan. A review of machine learning algorithms for text-documents classification. *Journal of advances in information technology*, 1(1):4–20, 2010.

[71] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Kaijie Zhu, Hao Chen, Linyi Yang, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *arXiv preprint arXiv:2307.03109*, 2023.

[72] Jie Gui, Zhenan Sun, Yonggang Wen, Dacheng Tao, and Jieping Ye. A review on generative adversarial networks: Algorithms, theory, and applications. *IEEE transactions on knowledge and data engineering*, 35(4):3313–3332, 2021.

[73] Shakir Mohamed and Balaji Lakshminarayanan. Learning in implicit generative models. *arXiv preprint arXiv:1610.03483*, 2016.

[74] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146, 2017.

[75] StackOverflow Developer Survey 2023. [on-line] `https://survey.stackoverflow.co/2023/`. Accessed: August 31st 2023.

[76] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[77] PDB parser. [on-line] `https://github.com/michal-kapala/pdb/`. Accessed: August 31st 2023.

[78] MSVC name mangling. [on-line] `https://en.wikiversity.org/wiki/Visual_C%2B%2B_name_mangling`. Accessed: August 31st 2023.

[79] John D Storey. False discovery rate. *International encyclopedia of statistical science*, 1:504–508, 2011.

[80] dubRE. [on-line] `https://github.com/michal-kapala/dubRE`.

Poznan University of Technology
Faculty of Computing and Telecommunication
Institute of Computing Science

Typeset using LaTeX.