UNIVERSITY OF NEVADA, RENO

Network Traffic Fingerprinting using Machine Learning and Evolutionary Computing

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Engineering

by

Ahmet Aksoy

Dr. Mehmet Hadi Gunes / Dissertation Advisor

August 2019

UNIVERSITY OF NEVADA RENO

THE GRADUATE SCHOOL

We recommend that the dissertation prepared under our supervision by

AHMET AKSOY

entitled

Network Traffic Fingerprinting using Machine Learning and Evolutionary Computing

be accepted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Mehmet Hadi Gunes, Ph.D. – Advisor

Sushil J. Louis, Ph.D. – Committee Member

Emily M. Hand, Ph.D. – Committee Member

Shamik Sengupta, Ph.D. – Committee Member

M. Sami Fadali, Ph.D. – Graduate School Representative

David Zeh, Ph.D. – Dean, Graduate School

August 2019

Abstract

The Internet has become essential to our daily life, especially with a multitude of IoT devices. However, the end hosts connected to the Internet are prone to be compromised. An essential measure for protecting attacks on end hosts is through the detection of system characteristics and isolation of vulnerable devices by restriction of communications to the device. Network traffic fingerprinting provides the ability to remotely and automatically gather information about the hosts within a network.

Fingerprinting can help perform network management and the detection and isolation of vulnerable hosts. It is essential to automate this process to perform fingerprinting more efficiently. It also provides the ability to adapt to changes in the behavior of host devices and software.

Network practitioners rely on some classifier tools for fingerprinting, but they rely on an expert to select features/attributes and generate machine learning models. Hence, existing approaches need to be manually updated for each new Operating System (OS) or IoT device introduced to the network.

This dissertation addresses automated methods and tools for performing fingerprinting of OSes and IoT devices. It also presents SILEA, a new inductive learning algorithm along with improvements to its numerical feature quantization to further improve classification accuracy.

SILEA is a covering-method inductive learning algorithm that reliably extracts IF-THEN rules from a collection of examples/instances. The algorithm eliminates exhaustive feature selection by reducing the number of features to be considered for each necessary iteration of rule extraction. We also use a genetic algorithm (GA) to determine the maximum number of clusters to be considered for each numeric feature for quantization and observe their contribution to classification accuracy. Once the number of clusters for each numeric feature is determined, we run the k-means algorithm for each feature with the number of clusters that are pre-determined by GA to obtain as optimal ranges for numeric features as possible.

We analyze the TCP/IP packet headers to automate OS classification. We utilize a GA to determine the relevant packet header features, which helps reduce the classification complexity and increases accuracy by eliminating noisy features from the data. We use several machine learning algorithms to generate a set of rules and models that can differentiate OSes. We also investigate an automated system, called OSID, for classifying host OSes by analyzing the network packets that they generate without relying on human experts.

We introduce another automated system, called SysID, for the classification of IoT device characteristics based on their network traffic. The system uses any single packet that is originated from the device to detect its kind. We utilize a GA to determine relevant features in different protocol headers, and then deploy various machine learning algorithms to classify host device types by analyzing features selected by GA. SysID allows a completely automated classification of IoT devices using their TCP/IP packets without expert input. SILEA, OSID and SysID codes and trained models are available at `https://github.com/netml/`.

Dedicated to my family,

advisor and friends…

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Dr. Mehmet Hadi Gunes, for the continuous support of my Ph.D. study and related research, for his patience, motivation, and immense help. I could not have imagined having a more helpful, understanding, and patient advisor and mentor for my Ph.D. study.

Besides my advisor, I would like to thank the rest of my thesis committee; Dr. Sushil J. Louis, Dr. Emily M. Hand, Dr. Shamik Sengupta and Dr. M. Sami Fadali for their insightful comments and encouragement, but also for the hard questions which incented me to widen my research from various perspectives. I would also like to thank Dr. Murat Yuksel for all the tremendous support and assistance he has given me on learning about and in pursuing an academic career.

I thank my fellow labmates, Dr. M. Abdullah Canbaz, Khalid Bakhshaliyev, Paulo Regis, Jay Thom and Steven Fisher for the stimulating discussions, the sleepless nights we spent working together before deadlines and for all the fun we had in the last five years. I also thank our department's administrative assistants, Lisa Cody, and Heather Lara, for their help throughout this journey.

Ahmet Aksoy (left) and Dr. Mehmet Sabih Aksoy (right) at the University of Wales, Cardiff in South Wales, UK in 1991.

Last but not the least, I would like to thank my family: my parents Dr. Mehmet Sabih, Aysel Aksoy; my sisters Yumna, Hamide and my brother Cuneyt Aksoy for supporting me spiritually throughout writing this thesis and my life in general. I thank my father, a professor of computer science, for his support all these years. His research and insights have been a tremendous help for me to shape my goals and to be where I am today. I could not have asked for a more understanding and supportive family.

This thesis is based in part upon work supported by the National Science Foundation under grant numbers CNS-1321164, EPS-IIA-1301726, and CICI-1739032. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

Abstract **i**

Acknowledgements **iv**

Contents **vi**

List of Tables **ix**

List of Figures **xi**

1 Introduction **1**

2 Related Studies **7**

2.1 Inductive Learning 7
2.2 Clustering using Genetic Algorithms 9
2.3 Genetic Algorithms for Feature Subset Selection 11
2.4 Operating System Fingerprinting 12
2.4.1 Active Fingerprinting 13
2.4.2 Passive Fingerprinting 14
2.4.3 Hybrid Fingerprinting 16
2.4.4 Mobile OS Fingerprinting 17
2.5 IoT Device Classification 17
2.5.1 Active & Passive Fingerprinting 17
2.5.2 Device Authentication 19
2.5.3 Time-domain Based Fingerprinting 19
2.5.4 Wireless Network Fingerprinting 20
2.5.5 Mobile Device Fingerprinting 20

3 SILEA - a System for Inductive LEArning **22**

3.1 Proposed Algorithm 23
3.1.1 Data Initialization 24
3.1.2 Feature Selection Procedure 26
3.1.3 Rule Extraction Procedure 28
3.2 Illustrative Problem 30
3.3 Experimental Results 35

4 Operating System Classification Performance of TCP&IP Protocol Headers 39
4.1 Methodology 41
4.1.1 Data Initialization 42
4.1.2 Feature Selection 44
4.1.3 Rule Extraction 45
4.2 Experimental Results 45
4.2.1 IP Protocol Headers 46
4.2.2 ICMP Protocol Headers 49
4.2.3 TCP Protocol Headers 50
4.2.4 UDP Protocol Headers 53
4.2.5 HTTP Protocol Headers 54
4.2.6 DNS Protocol Headers 55
4.2.7 SSL Protocol Headers 57
4.2.8 SSH Protocol Headers 58
4.2.9 FTP Protocol Headers 59
4.2.10 Discussion 60

5 Operating System Fingerprinting via Automated Network Traffic Analysis 61
5.1 Methodology 63
5.1.1 Data Collection 63
5.1.2 Feature Selection 65
5.1.3 OS Classification 66
5.2 Experimental Results 67
5.2.1 OS Genre Results 68
5.2.2 Linux OS Results 69
5.2.3 Mac OS Results 71
5.2.4 Windows OS Results 73
5.2.5 Feature Subset Selection 74
5.2.6 Discussion 77

6 Operating System Identification using Network Packet Headers 78
6.1 OSID: Operating System Identifier 80
6.1.1 Data Initialization 81
6.1.2 Feature Selection 81
6.1.3 OS Identification 84
6.2 Experimental Results 85
6.2.1 Data Collection 86
6.2.2 Individual Protocol Classification Results 87
6.2.3 Individual Packet Classification Results 98
6.2.4 Network Flow Classification Results 102
6.2.5 TCP SYN results 103
6.3 Discussion 106

7 Automated IoT Device Identification from Network Traffic 109
7.1 Methodology . . . 110
7.1.1 Feature Selection . . . 111
7.1.2 IoT Device Classification . . . 114
7.2 Experimental Results . . . 114
7.2.1 Data . . . 115
7.2.2 One-level Classification . . . 115
7.2.3 Two-level Classification . . . 118
7.2.4 Feature Subset Selection . . . 122
7.3 Discussion . . . 124

8 Automated Quantization of Numerical Features using K-means and Genetic Algorithms 126
8.1 K-means Clustering . . . 128
8.2 Genetic Algorithm Optimization . . . 129
8.3 Classification . . . 132
8.4 Experimental Results . . . 133
8.4.1 Data . . . 133
8.4.2 Classification Accuracy . . . 134

9 Conclusion 139

10 Future Work 142

Bibliography 144

List of Tables

3.1 Algorithm Complexities 30
3.2 Example Set 31
3.3 *PotentialList* - Potential rules extracted from the 1st example . . . 32
3.4 *PotentialList* - Potential rules extracted from the 1st and 2nd examples 32
3.5 *Blacklist* - Attribute name(s) which belong to more than one class 33
3.6 *PotentialList* - Potential rules extracted from the 1st, 2nd and 3rd examples 33
3.7 *PotentialList* - Potential rules selected from all the examples with $n_c = 1$ and from the 1st example with $n_c = 2$ 34
3.8 *SelectedList* - All selected rules from the dataset 34
3.9 Accuracy for the Balloons dataset 35
3.10 Accuracy for the Hayes-Roth dataset 36
3.11 Accuracy for the Hepatitis dataset 36
3.12 Accuracy for the Iris dataset 37
3.13 Accuracy for the Lenses dataset 37
3.14 Average accuracy 38

4.1 Number of packets 44
4.2 Number of selected features 46
4.3 Selected IP Features 47
4.4 Selected ICMP Features 50
4.5 Selected TCP Features 52
4.6 Selected UDP Features 53
4.7 Selected HTTP Features 55
4.8 Selected DNS Features 56
4.9 Selected SSL Features 58

5.1 Number of selected features for OS identification 75
5.2 Number of selected features for Linux version identification . . . 75
5.3 Number of selected features for Mac OS version identification . . 75
5.4 Number of selected features for Windows version identification . 76
5.5 Classifier settings 76

6.1 Selected IP Features 89
6.2 Selected ICMP Features 91

6.3 Selected SSL Features 94
6.4 Selected TCP Features 96
6.5 Protocols Selected for Flow Analysis 101
6.6 OSID Confusion Matrix 103
6.7 p0f Confusion Matrix 105

7.1 Confusion Matrix 116
7.2 Number of selected features 122
7.3 Selected Features 123

List of Figures

4.1 IP Accuracy . . . 47
4.2 ICMP Accuracy . . . 49
4.3 TCP Accuracy . . . 51
4.4 UDP Accuracy . . . 53
4.5 HTTP Accuracy . . . 54
4.6 DNS Accuracy . . . 56
4.7 SSL Accuracy . . . 57
4.8 SSH Accuracy . . . 59
4.9 FTP Accuracy . . . 59

5.1 Instances . . . 64
5.2 Classifiers . . . 66
5.3 Classification accuracy of IP, TCP and UDP protocols for OS genre 68
5.4 Classification accuracy of IP, TCP and UDP protocols of Linux versions . . . 70
5.5 Classification accuracy of IP, TCP and UDP protocols for Mac OS versions . . . 72
5.6 Classification accuracy of IP, TCP and UDP protocols for Windows versions . . . 73

6.1 Integrated Validation and Training . . . 83
6.2 Classifiers . . . 84
6.3 IP Performance . . . 88
6.4 ICMP Performance . . . 90
6.5 SSL Performance . . . 92
6.6 TCP Performance . . . 95
6.7 UDP Performance . . . 97
6.8 IP & TCP Performance . . . 98
6.9 IP & TCP & HTTP Performance . . . 99
6.10 IP & TCP & SSL Performance . . . 99
6.11 IP & UDP Performance . . . 100
6.12 IP & UDP & DNS Performance . . . 101
6.13 TCP SYN Packet Performance . . . 104

7.1 Validation/Training . . . 111
7.2 Overall classification accuracy . . . 117

7.3 Classifiers 119
7.4 Device genre classification accuracy 119
7.5 Hue devices classification accuracy 120
7.6 TP-LinkPlug devices classification accuracy 120
7.7 EdimaxPlug devices classification accuracy 121
7.8 D-Link devices classification accuracy 121
7.9 Smarter devices classification accuracy 122

8.1 Device genre classification performance 132
8.2 EdimaxPlug classification performance 136
8.3 Hue classification performance 136
8.4 Smarter classification performance 137
8.5 TP-LinkPlug classification performance 137
8.6 Classification accuracy with and without GA 138

Chapter 1

Introduction

Network management requires knowledge of devices attached to a network. A vulnerable device connected to the network could be utilized for insider attacks. Hence, network administrators identify devices that are connected to the network in order to monitor and secure the network. To this end they may perform local or remote network measurements to analyze networked devices [1]. Detecting the Operating System (OS) versions of connected systems helps determine vulnerabilities of particular OSes. Knowing vulnerable systems, network administrators can take necessary actions to secure not only a particular system but also the whole network. OS identification is also helpful for network administrators to manage and monitor a large number of hosts within a network. As the various Internet of Things (IoT) devices are introduced into a network, the network administrators need a better understanding of the devices that are connected as well. In particular, IoT devices lack computational capabilities of typical networked devices, and hence need closer monitoring to prevent/detect intrusion/malware. Securing IoT devices is not only crucial for the device itself but also for other network components. Identification of network devices is valuable for network managements as well as network security [2]. Also, fine grained access controls could be implemented for critical systems to improve system security [3].

Machine learning techniques have been beneficial in terms of knowledge extraction from examples in an automatic way [4]. One of the most preferred machine learning techniques undoubtedly is inductive learning [5]. Inductive learning is the process of reaching general rules from specific examples/instances [6]. The reasons for inductive learning algorithms to be preferred are their simplicity, speed, and accuracy [5]. The categories of inductive learning can be listed as divide-and-conquer methods and covering methods [7]. Decision tree-based approaches of divide-and-conquer methods are efficient but not always reliable in terms of the generality of the rules they generate. Covering-method algorithms provide more flexibility and generality but have higher complexity.

In this dissertation, we present a new covering-method inductive learning algorithm called SILEA [8], which efficiently generates a set of IF-THEN rules. It employs a feature selection technique similar to that of Sequential Forward Selection [9] to decrease the number of attribute-value pairs that are to be considered. Sequential Forward Selection prioritizes certain attributes over the others by an objective function [10]. Similarly, SILEA ensures that these biases are made on attributes with higher metric values than other attributes. The metric used for this purpose is the entropy measure, which allows SILEA to eliminate an enormous amount of possible combinations for each iteration. Also, the extraction process that SILEA employs eliminates unnecessary comparisons for rule extraction. It extracts all possible rules for each combination and selects the most classifying ones among them. It excludes rules which might become redundant due to the existence of more classifying rules. This approach assures the extraction of the most general rules for each combination of attributes considered.

In covering-method inductive learning, the quantization of numerical features plays a significant role in providing generality to the rules extracted. Quantization is a method that allows inductive learning algorithms to process numerical attributes by determining ranges for data points. Assume a feature containing the ages for every employee in a company. Depending on the number of employees and their age distribution, the age feature in the dataset can contain many unique values. Rather than extracting a specific rule for every unique value in the dataset, which could introduce over-fitting, quantization provides the ability to determine ranges of similar values of employee ages which help both reduce the number of rules extracted and provide generality to the rules extracted. In this dissertation, we employ an automated clustering technique using genetic algorithms (GA), and we test the automated quantization method with the SILEA [8] algorithm to further increase the classification accuracy.

Especially with the Internet of Things (IoT), everything is getting connected to the Internet. With a plateau of devices attached to a network, local network management is becoming more challenging. Management and security of a medium to a large network requires an understanding of devices connected to the network. As various OSes run on these devices, identifying and patching vulnerable systems is crucial. Network managers adopt multiple security mechanisms to protect the network from malicious activities. An essential step in securing a network is to be aware of the devices that are attached to it. It is crucial to detect devices that use old or insecure versions of OSes due to their vulnerabilities to remote attacks [11]. Detecting the OS versions of connected systems helps determine vulnerabilities of particular OS versions [12]. Knowing vulnerable systems, network administrators can take necessary actions to secure not only the particular system but also the whole network. OS identification is also helpful for network administrators to manage and monitor a large number of hosts within a network [13].

OS kernels provide a mechanism for transferring network packets from a source to a destination. Although protocol standards exist for packet header information that OSes implement, there still are ambiguities in the specification and developers have differing defaults for protocol fields. Such unique implementations in packet headers provide the ability to identify OSes based on the TCP/IP headers [14]. OS fingerprinting is the process of remotely detecting a system's OS through packet signatures generated by it. Detecting OS is essential for determining system vulnerabilities, improving cybersecurity, and identifying potential attacks. OS detection also helps network administrators better manage and monitor large numbers of hosts [13]. Such detection can also be extended for the classification of host roles [15].

Researchers perform OS identification considering various TCP/IP protocols. TCP, TCP SYN and ICMP protocols are among the most informative TCP/IP protocols for OS identification. Various approaches in machine learning such as neural networks, Naive Bayesian classifiers, as well as more straightforward approaches such as Bayes' rule and statistical tools have been used in OS identification. The information extracted from the network packets to perform OS identification is acquired either actively or passively. Active identification approaches perform OS detection by probing the system and analyzing the responses. Passive identification approaches perform OS detection by sniffing and analyzing the information extracted from the packets of the system. There are several active probing approaches for OS identification [16]–[22] as well as passive approaches [11], [20], [23]–[26].

In this dissertation, we also present an entirely machine learning-dependent OS classification approach and perform single-packet OS classification on network packets that originate from host devices [14]. Specifically, we perform OS classification on a single sniffed packet by extracting and checking the packet's protocol header information [27].

Different types of network packets yield different features that can be used for classification [14]. For each type of network packet, and for each classifier that we use, a GA evolves a subset of packet-type dependent features relevant to OS classification. The evolved feature subsets have fewer features, and the lower number of features enable faster system classification. Furthermore, these evolved features generally lead to improved classification accuracy and lower classifier complexity. We conjecture that the GA eliminates features that do not contribute to the accuracy or add noise and thus decrease accuracy. The reason why we used GA is due to the large search space of feature combinations in the protocols. For instance, for the TCP protocol header, we obtained around 61 features which make 2^{61} unique combinations to consider. We tried a hill-climber, but it did not yield as high classification results as GA. Although the presented approach is applicable to both active and passive fingerprinting, since we collected packets passively, the results presented in this dissertation are for passive fingerprinting.

Feature subset selection is computationally expensive but cheap; distributed computing power makes this a non-issue in practice. Collecting the data needed for training is a different issue. Our approach's accuracy depends on the quality and quantity of the data collected.

The network administrators also need a better understanding of the devices that are connected to a network. The use of the Internet of Things (IoT) devices is increasing rapidly. Since in many cases, the IoT devices do not have the necessary computation power, such devices require closer monitoring in order to prevent intrusion or detect malware. Therefore, the security of IoT devices is beneficial both for the devices themselves and for other components within the same network. For instance, Mirai botnet utilized over 400,000 IoT devices in launching the first DDoS attack with over 1 TB traffic volume [28].

In this dissertation, we also present an IoT device fingerprinting system, System IDentifier (SysID) that can identify the device type from a single packet with high accuracy [29]. We utilize machine learning and GA to learn the unique features of each IoT device without expert supervision. Header field information in different protocols contains unique signatures and provides clues for the classification of devices. We utilize a GA to select features that are as unique as possible for each IoT device compared to other devices in the training data. GA reduces the number of selected features to improve system efficiency in a large search space along with classification accuracy. It also helps eliminate features that negatively affect classification accuracy.

Chapter 2

Related Studies

2.1 Inductive Learning

There are many divide-and-conquer and covering-method inductive learning algorithms that induce knowledge from examples. Inductive learning forms a knowledge base from a given set of examples where each example contains several attribute values along with a class [30].

ID3 is a divide-and-conquer algorithm introduced by Quinlan in 1987 which uses training examples to generate a decision tree [31]. It selects nodes according to their entropies to construct a tree more efficiently [32]. ID3, however, lacks in terms of the generality of the trees it generates [33]. Many covering-method algorithms have used this vulnerability to their advantage by introducing algorithms that focus more on the generality of the rules they induce.

C4.5 is another well known divide-and-conquer algorithm proposed by Quinlan in 1993 [34]. This improved algorithm, unlike ID3, can handle continuous attributes. Another improvement was pruning. ID3 is more sensitive to noise and to prevent the tree from over-fitting the data, C4.5 prunes the tree by eliminating the sections of the tree that contribute little to data classification [35].

AQ is a covering-method algorithm introduced in 1969 by Michalski [36]. AQ was initially introduced to solve the boolean function satisfiability problem. However, it was later adapted to solve the covering problem. The algorithm has been applied to several problems, such as the generation of individuals within an evolutionary computation network. The algorithm has been improved several times but has not been used widely mainly due to its high complexity [37].

CN2 is a covering algorithm introduced in 1989 by Peter Clark and Tim Niblett [38]. CN2 algorithm takes advantage of ID3's noisy data handling approach along with the flexibility of AQ family algorithms. In CN2, rule forming procedure is terminated by the use of a heuristic function based on an estimate of the noise observed in the data. As a result, CN2 may generate rules which do not necessarily classify all the training examples but perform well on new data.

RULES3 is also a covering algorithm introduced in 1995 by Pham and Aksoy. RULES3 is an automatic rule extraction system that was released as an advancement to its predecessors, RULES1, and RULES2. RULES3, in addition to its predecessors, introduces two new features: 1) it allows the user to set the precision of rules and 2) it provides more generality to the rules it generates [6].

RULES3-Plus was released in 1997 by Pham and Dimov [32] to overcome the issue of the exhaustive searching procedure that RULES3 employs [6]. RULES3-Plus provides two advantages over its predecessor, 1) it adopts a more efficient rule searching procedure and 2) it uses the h-measure metric for selecting attributes [6], [32].

Several algorithms, based on RULES3 and RULES3-Plus, have been implemented. These algorithms have introduced many improvements to the base algorithms. For instance, RULES4 can extract rules incrementally [37]. RULES5 employs a new method to handle continuous attributes and to extract rules [4]. RULES6 also introduces a new method for continuous attribute handling along with a noise-tolerant rule extraction technique [5]. In addition to handling continuous attributes, RULES-F can generate accurate and compact fuzzy models that allow it to handle continuous classes as well [39]. RULES3-EXT can perform attribute re-ordering and fire rules partially if the extracted rules are unable to classify new examples [40]. RULES-TL uses transfer learning by collecting knowledge from agents in different domains, which helps reduce the search time [41]. RULES-IT algorithm is the incremental version of the RULES-TL algorithm that also transfers rules from different domains to improve its accuracy [42].

2.2 Clustering using Genetic Algorithms

In this paper [43], authors use a genetic algorithm (GA) to find optimum clustering. [44] also employs GA to further improve the k-means clustering algorithm by overcoming its major limitation of being stuck with the local optima. In the GA implementation, the centers of the clusters are encoded in a chromosome rather than a partition which helps reduce the length of the chromosome as the number of data points in most cases exceeds the number of clusters.

Authors in [45], however, find a global optimum for a partition. They also employ GA and k-means. To avoid expensive crossover operation, they use a hybrid approach of using GA with a gradient descent algorithm.

This study [46] is an improvement to [45] where although both approaches always converge to global optima, the latter approach can run faster.

This paper [47] introduces an improved genetic k-means algorithm that can dynamically detect k value for the k-means algorithms. The proposed approach tries to reduce the number of clusters needed as much as possible while trying to ensure a large separation of clusters. [48] also proposes an approach for automatically detecting the number of optimal clusters using a quantum-inspired GA.

One of the important issues in the k-means algorithm is the initial selection of centers. It is possible to obtain different results as the initial centers change. This paper [49] addresses this issue, and the authors employ a GA that evolves the centers to identify the right partitions for a range of values.

Authors of [50] address both the issue of the initial selection of centers and determining the number of clusters within the data. They automatically find the number of clusters to be used with the k-means algorithm using GA. After determining the centers, they feed them to the k-means algorithm to increase accuracy. [51] also automates the selection of the number of clusters to use with the k-means algorithm. Similarly, [52] tries to optimize both the initial centers and partitions from the data. It employs a two-stage GA approach where the probabilities for selection and mutation operations are determined through the use of GA.

There is also research trying to eliminate the numeric data only limitation. [53] proposes a clustering approach based on the GA and the k-means algorithm paradigm, which can work with both numeric and categorical data.

2.3 Genetic Algorithms for Feature Subset Selection

Genetic algorithms (GA) can be used to select feature subsets from a dataset. Feature subset selection is the process of selecting a smaller set of features based on an optimization criteria [54], [55]. Feature subset selection helps increase classification accuracy by selecting features that contribute to the classification the most and increase efficiency by reducing the number of features to process [56].

There exist many work on feature selection using GA. Most of these work depends on wrappers where different machine learning algorithms are used to evaluate subsets of features selected by GA [57]. However, there is work on applying GA to clustering using filter methods as well [58]. Due to classification accuracy and simplicity, SVMs and K Nearest Neighbor (KNN) are among the most preferred algorithms. Researchers analyzed how different values for population size, mutation, and crossover effect the accuracy [59].

Studies such as [57] and [60] do not consider feature interaction. Elimination of feature interaction potentially creates an issue of elimination of features that might yield better classification accuracy together as opposed to being used individually. There are traditional approaches to GA feature selection as well. For example, in [57], the authors combine both SFFS and GA to select features. The results are claimed to be improved with such hybrid approaches. However, this introduces extra computation overhead.

It is also possible to use GA along with a classifier to improve the original classifier's accuracy. Kelly and Davis [61] use GA and KNN algorithms in conjunction to improve accuracy. GA helps increase KNN's accuracy by searching a weight vector. Their results show that KNN, along with GA, performs better than KNN alone [62].

2.4 Operating System Fingerprinting

Operating system (OS) fingerprinting techniques are often classified as active and passive approaches [63]. Both active and passive fingerprinting approaches have received much attention over the last few years, and many tools have been developed for these approaches. In active fingerprinting, the target device is directly probed, and the OS of the target device can be detected based on its response. In passive fingerprinting, OS detection can only be performed by analyzing sniffed packets from the target device. There are also hybrid approaches that try to overcome the limitations of active and passive fingerprinting by combining both. For instance, Sinfp uses signatures acquired from active fingerprinting to perform passive fingerprinting [64].

As noted earlier, active fingerprinting tools can be blocked by firewalls and IDSs. Such tools also require numerous probes to classify OSs accurately. Even though there is work on trying to reduce the number of probes required, e.g., [65], we cannot guarantee that the system will not block packets generated by these tools before it receives enough information to identify the system accurately. Often, such tools are not even aware of whether it is the firewall or the actual system that they are scanning [20].

Passive fingerprinting is more limited than active fingerprinting since passive systems cannot choose the type of packet and therefore, the type of information to use for classification. Since passive fingerprinting techniques sniff packets, they are limited to the information that the packets provide.

2.4.1 Active Fingerprinting

Active fingerprinting tools can request specific types of information that are useful to distinguish OSes. Therefore, active fingerprinting tools usually yield high accuracy. Since such active measurements involve probing the system, they could interfere with the operation of the system or be blocked from probing the system. Additionally, system responses depend on the probe packets [66], [67].

Veysset et al. perform a temporal response analysis-based OS detection [16]. Their system employs an active fingerprinting approach to elicit the TCP SYN packet responses, which is then compared to the known signatures to detect the OS of the target system.

Similarly, Arkin performs active OS classification by analyzing the ICMP replies from target systems [17].

SYSNSCAN tries to determine the distinguishing information among different TCP implementations of OSes to determine the OS of a target system [18]. In addition to incorporating many of the existing techniques, SYNSCAN also depends on features such as congestion control, congestion window size, don't fragment bit, default MSS value, IP identification field, TTL value, etc.

Greenwald et al. derive effective probe communications for OS detection by evaluating fingerprinting probes to reduce the number of packets to be exchanged with the target system [65]. In active fingerprinting, multiple probes to a target system may be needed to deduce the OS of the target device. It is possible that tools such as IDS at the target network can detect and prevent responses to such probes. By minimizing the probes to a target system, the authors try to evade such mechanisms.

Additionally, machine learning techniques have been adopted for performing OS classification [21]. Authors employ neural networks and statistical tools with DCE-RPC endpoints and Nmap [19] signatures to improve detection analysis. They try to initially detect the OS type (Windows, Linux, Mac) of packets using neural networks, and then Nmap's signature database to further classify the specific OS version, for example as Windows 7. The proposed system consists of two modules where one of them performs Windows OS classification using DCE-RPC, and the other oner performs Linux-based OS classification using Nmap. Authors have relied on features such as ACK flag responses: S, S++, O; DF flag response (yes/no); response flag: ECN-Echo, URG, ACK, PSH, RST, SYN, FIN; Options field and window size.

Nmap is an active OS fingerprinting tool introduced by Gordon Lyon [20]. It has received multiple improvements over the years, which gave it the ability to classify various OSs. However, since it sends up to 16 probes to be able to make a decision, it becomes easily detectable and blockable.

Xprobe2 is another active fingerprinting tool [22] and mostly uses ICMP protocol probes to perform fingerprinting. Xprobe2 can perform partial matching and using ICMP enables Xprobe2 to distinguish similar OSs such as different Windows versions.

2.4.2 Passive Fingerprinting

Passive fingerprinting tools merely sniff packets originating from the host. Such systems can perform identification as long as the host device generates network traffic. The disadvantage of passive identification tools is that they are limited in the type of information they can access to perform identification. If the information provided in the network packets are not distinguishing enough, they might not be able to narrow down the search space.

p0f was introduced by Michal Zalewski [20] and is widely used. p0f extracts header information from TCP SYN packets which then are compared to a database of signatures for OS classification.

Spitzner performs passive OS classification on pre-determined signatures such as TTL, window size, DF and TOS [24].

Lippmann et al. determine the accuracy of passive OS classification based on TCP/IP packets along with evaluating open-source tools for OS classification [11]. They also evaluate suitable classifier techniques to increase classification accuracy.

Beverly developed a Naive Bayesian classifier for passive fingerprinting [68]. The presented machine learning-based approach is compared to rule-based inference tools such as p0f's signature database and HTTP UserAgent data in terms of its classification accuracy.

Chen et al. analyze and identify a series of TCP/IP header features to examine the effectiveness of such features for their contribution to OS classification, including mobile devices [25]. They utilize features such as the stability of the clock frequency, presence of TCP timestamp option, and the default set of TCP window size scale.

Mavrakis develop a machine learning-based system that uses TCP/IP headers and decision-tree learning [69]. It is shown that the UserAgent data outperformed p0f's signature database. Since they use p0f's signatures, the selected features are similar to those of p0f, including MSS, WS, and iTTL. They also consider features such as options layout and IP version.

Unlike OS classification from TCP/IP traffic analysis, Chang performs OS classification based on DNS logs [70]. The author used a chi-squared test to extract features from DNS logs to distinguish different OSes and used the hamming distance for classification.

Some tools focus on performing OS fingerprinting using IPv6 packets [26] where they analyze data obtained from passive measurements to perform a comparison between IPv4 and IPv6 data.

2.4.3 Hybrid Fingerprinting

There are also hybrid systems such as SinFP that try to reap the benefits of both approaches [64]. SinFP also introduced methods such as using signatures collected from one system to perform classification on another. They also perform active and passive OS fingerprinting with IPv6. Like p0f, SinFP relies on TCP SYN packets for fingerprinting.

Ettercap detects man-in-the-middle attacks and can also perform OS fingerprinting [71]. Like p0f, Ettercap uses TCP SYN packet information for OS classification. This tool is not a pure passive fingerprinting tool since it sends SYN packets to the system and checks responses.

Another hybrid approach was introduced in [72]. In this study, the authors use the answer set programming where the problem of OS classification is solved through automated reasoning.

[63] formalizes the problem of OS fingerprinting as a diagnosis problem. Diagnosis problem helps detect components within a system which helps determine the incompatibility of what is observed and how the system is supposed to function [73]. They employ both active and passive fingerprinting approaches to extract information from previous observations and to be able to request information on demand.

2.4.4 Mobile OS Fingerprinting

There is some work in Mobile OS classification as well. In [25], the authors try to improve the classification of Mobile OSs by introducing new features. Their approach implements Bayes' rule to perform classification.

2.5 IoT Device Classification

Similar to OS identification, researchers have developed approaches to fingerprint IoT devices.

2.5.1 Active & Passive Fingerprinting

Nmap is an active OS fingerprinting tool that is also capable of device fingerprinting [74]. It takes advantage of different implementation of network stack by different vendors to detect the device type. Nmap generates up to 16 probes to detect the OS version or the device type.

A couple of passive fingerprinting approaches focus on network packet features. P0f performs passive fingerprinting and leverages the TCP SYN packets' header information for OS detection, as well as device identification [20]. Gao et al. perform wavelet analysis on packet traffic to distinguish access points [75].

In [76], authors introduce GTID, which is a passive fingerprinting technique used to fingerprint wireless devices. The authors rely on the wired backbone observations such as hardware compositions (e.g., processor, DMA controller and memory) and hardware variations (e.g., clock skew) of the devices to perform device type identification and device identification, respectively. They utilize statistical approaches to capture the uniqueness of devices. The main limitation of this work is that since the approach relies on the timing of packets and since this timing is possible to be lost in switches and routers due to buffering, the proposed approach is not very suitable for classifying devices across the Internet.

The most similar study to ours is [77], where the authors propose an automated system for device fingerprinting. Miettinen et al. collect n packets after a new IoT device initiates its setup phase. This n is determined when a decrease in the packet rate is observed. They use 23 features to fingerprint a device where 19 are binary values representing the absence or existence of the following protocols; link layer (i.e. ARP, LLC), network layer (i.e. IP, ICMP, ICMPv6, EAPoL), transport layer (i.e. TCP, UDP), application layer (i.e., HTTP, HTTPS, DHCP, BOOTP, SSDP, DNS, MDNS, NTP), data payload, and IP options (i.e. Padding, RouterAlert). Additional four features are integer values to count for packet size, destination IP counter, source port counter, and destination port counter. They remove consecutive identical packets and use the first 12 unique vector packets and generate a 23x12 matrix as a fingerprint for each device. Then they use the RandomForest algorithm to generate a classifier for each device.

2.5.2 Device Authentication

In [78], authors propose a device authentication protocol named S2M (speaker-to-microphone). The proposed protocol extracts an acoustic fingerprint for each device by utilizing the frequency response of their speakers and microphones. S2M helps authenticate users by checking for a match with the extracted fingerprints. Although the authors claim that many IoT devices contain microphones and speakers, the limitation of the approach presented occurs with simpler IoT devices that may not contain such hardware.

An object authentication framework is proposed in [79]. The proposed framework uses transfer learning to distinguish between an attack and a legitimate change of behavior by analyzing the effect of the physical environment on IoT devices.

2.5.3 Time-domain Based Fingerprinting

Several approaches focus on timing characteristics. Passive and Temporal Fingerprinting performs device fingerprinting based on the application layer protocols' timing [80]. RTF represents fingerprints in a tree-based temporal finite state machine and uses SVM (Support Vector Machines) as a classifier. Similarly, Radhakrishnan et al. model the distribution of packet inter-arrival times (IAT) and use Artificial Neural Network (ANN) to classify devices [81]. Formby et al. focus on the identification of industrial control systems with data response processing times and physical operation times as device signatures [82]. Kohno et al. identify devices based on the distribution of clock skews, which are captured from TCP timestamps [83].

2.5.4 Wireless Network Fingerprinting

Several studies focus on wireless network characteristics. Desmond et al. detect devices connected to a Wireless-LAN by analyzing the timing of 802.11 probe request frames [84]. The authors employ clustering algorithms to generate fingerprints. Nguyen et al. propose a passive fingerprinting technique using radio-metrics as the distinguishing feature of devices to detect identity spoofing [85]. The radio-metrics include radio signal amplitude, frequency, etc. They then employ a non-parametric Bayesian method for device identification. Xu et al. focus on physical, MAC, and upper-layer characteristics to fingerprint wireless devices using a White-List Based Algorithm and Unsupervised Learning [86]. Physical layer fingerprinting of ZigBee devices using Radio Frequency Fingerprinting is performed in [87]. In this work, the author proposes improvements to their previous work after observing that, unlike the normal distribution assumption in their previous work, most of the signals collected from the devices are either multi-modal or non-parametric. The author uses non-parametric methods for feature generation. The authors claim to have increased their accuracy up to 9%.

2.5.5 Mobile Device Fingerprinting

In this study [88], authors utilize sensors in smartphones to extract unique fingerprints across them. The authors utilize the frequency response of the speaker and microphone and the accelerometer calibration errors of devices. One of the advantages of the proposed approach is claimed to be easy access to devices' accelerometer readings by JavaScript. Authors claim that with their approach, the fingerprints extracted can be used to de-anonymize devices' activities on the Internet. It is shown in the paper that the devices can be uniquely identified based on the entropy from the devices' sensors. However, the proposed approach performs device-specific fingerprinting. It may not be able to detect the generalities across devices to perform device-type fingerprinting.

Chapter 3

SILEA - a System for Inductive LEArning

In this section, we present a new inductive learning algorithm called SILEA. In order to reduce the complexity of rule extraction, SILEA employs a feature selection technique similar to that of Sequential Forward Selection [9]. Sequential Forward Selection is a technique that prioritizes certain attributes over others based on an objective function [10]. SILEA uses the entropy measure for prioritizing the features to be selected.

In order to asses SILEA's complexity and accuracy, we compared it with some of the well-known algorithms in the field. SILEA considerably reduces the number of attribute combinations, i.e., from O(n^3) to O(n^2) in the worst case and has a smaller run-time complexity. The algorithm also performs better than the others on averages of all the analyzed datasets.

3.1 Proposed Algorithm

SILEA is an inductive learning algorithm that generates rules from a dataset efficiently and accurately. In order to achieve this, the algorithm employs two important characteristics, namely, feature selection and rule extraction.

The feature selection approach that SILEA employs is similar to the Sequential Forward Selection method. To avoid consideration of every possible combination of attributes, the algorithm in each iteration fixes $n_c - 1$ attributes and takes the combinations of these pre-selected attributes with the remaining ones. To avoid accuracy degradation, it needs to select these attributes judiciously. Otherwise, the algorithm can miss more optimal attribute combinations. That is why SILEA uses the entropy measure to select the attributes in each iteration of the rule extraction. It favors attributes with lower entropy values to assure the selection of more information-gaining attributes, which helps SILEA to generate as general rules as possible for a given iteration.

After deciding on which attributes to consider for rule extraction, SILEA extracts all possible rules from a dataset. For each iteration, SILEA generates all possible rules for a given number of conditions with a single visit of every example in the dataset. Each potential rule that SILEA forms from the selected attributes are considered to be a rule unless it contradicts other examples. A contradiction occurs when the formed attribute combination value(s) belong to more than one class among the examples in the dataset. After the extraction, the rule selection phase of SILEA eliminates obsolete rules. Obsolete rules are rules that can be replaced by higher occurring ones. After selecting the rules, it discards the examples that can be classified by these rules. SILEA stops rule extraction when it can classify all the examples in the dataset with the selected rules.

SILEA induces rules from a set of examples within a dataset as presented in Algorithm 1 through Algorithm 4. Each example in a dataset contains numerous attributes and a class [30]. A single or combination of attributes is considered a condition. The number of attributes within a condition could vary between one and n_a (total number of attributes in an example). After data initialization in Section 3.1.1, feature selection and rule extraction procedures of SILEA are explained in Section 3.1.2 and Section 3.1.3.

3.1.1 Data Initialization

To ease the selection process before feature selection and rule extraction, the ***SortFeatures*** function is executed to sort attributes according to their entropy values from the lowest to the highest (Alg1-Ln1). SILEA quantizes numerical attributes by defining and setting ranges for their values (Alg1-Ln2). For the quantization process, the algorithm executes ***QuantizeAttributes*** function. It finds the ranges of each numerical attribute (Alg2-Ln1) by determining the minimum and maximum values of the attribute (Alg2-Ln2-3) and dividing their difference by the number of quantization levels provided by the user (Alg2-Ln4). Then, for each example (Alg2-Ln5), the range that the corresponding attribute value belongs to is calculated and assigned (Alg2-Ln6-12). Note that we quantize values outside the range as *min* or *max*. After the quantization process, the algorithm goes into a loop which is executed until there are no more unclassified examples left in the dataset (Alg1-Ln6).

Algorithm 1: ***SILEA*** **rule forming procedure**

```
SortFeatures(Examples);
QuantizeAttributes(Examples, NoOfRanges);
$SelectedRules = \emptyset$;
$n_c = 0$ ;                                        // default
$UnclassifiedExamples = Examples$;
while UnclassifiedExamples != $\emptyset$ do
    $n_c = n_c + 1$;
    $Blacklist = \emptyset$;
    $PotentialList = \emptyset$;
    for each Example $\in$ Examples do
        FormedRules = GenerateFormedRules(Example, $n_c$);
        for each Rule $\in$ FormedRules do
            if Rule $\notin$ Blacklist then
                if Rule $\notin$ PotentialList then
                    $PotentialList = PotentialList \cup Rule$;
                    $Rule.occurrence = 1$;
                else if Rule $\in$ PotentialList and Rule.class = Example.class then
                    $Rule.occurrence = Rule.occurrence + 1$;
                else if Rule $\in$ PotentialList and Rule.class != Example.class then
                    $PotentialList = PotentialList - Rule$;
                    $Blacklist = Blacklist \cup Rule$;
```

Algorithm 2: ***QuantizeAttributes****(Examples, NoOfRanges)*

```
for each $Attribute_i \in Examples$ do
    Min = FindMin($Attribute_i$);
    Max = FindMax($Attribute_i$);
    $Range = (Max - Min) / NoOfRanges$;
    for each Example $\in$ Examples do
        if $Example.Attribute_i <= Min$ then
            $Example.Attribute_i = 1$;
        else if $Example.Attribute_i > Max$ then
            $Example.Attribute_i = NoOfRanges$;
        else
            $Example.Attribute_i = \lceil (Example.Attribute_i) / Range \rceil$;
```

3.1.2 Feature Selection Procedure

SILEA tries to extract rules from an example set according to the unique correspondence of attribute-value pairs and their associated classes. It starts with the minimum number of conditions set by the user, and as needed, it keeps incrementing them until it reaches the maximum, which is equivalent to the number of attributes n_a.

An important feature of SILEA for optimizing the performance and accuracy of the classification while increasing efficiency is to prioritize certain attributes over the others. It employs a feature selection technique similar to that of Sequential Forward Selection [9] to dramatically decrease the number of combinations to deal. The criteria for such selection is decided based on the entropy of the attributes [89]. SILEA selects the attributes according to their entropies from the smallest to the highest, in other words, from the most information-gaining to the least. The entropy of i^{th} attribute A_i, i.e., $E(A_i)$ is formulated as:

$$\sum_{j=1}^{|S_i|} \frac{|S_{ij} \in A_i|}{|I|} \left[- \sum_{k=1}^{|C|} \frac{|(S_{ij} \cup C_k) \in I|}{|S_{ij} \in A_i|} log \frac{|(S_{ij} \cup C_k) \in I|}{|S_{ij} \in A_i|} \right]$$

where;

I = set of i^{th} attribute value and i^{th} class value pairs from the dataset,

S_i = set of unique values that the i^{th} attribute can take,

C = set of unique classes in the dataset.

Rule extraction starts with the minimum number of conditions. It can manually be set to any number ranging from 1 to n_a. If all the examples in the dataset cannot be classified by the rules extracted while satisfying the current number of conditions (Alg1-Ln6), it is incremented by one (Alg1-Ln7) and the process

Algorithm 3: ***GenerateFormedRules***(*Example*, n_c)

```
1 $FormedRules = FixedAttributes = \emptyset$;
2 for $i = 1, i{+}{+},$ while $i <= n_c - 1$ do
3     $FixedAttributes = FixedAttributes \cup Example.attribute_i$;
4 for $i = 1, i{+}{+},$ while $i <= n_a - (n_c - 1)$ do
5     $FormedRule = FixedAttributes \cup Example.attribute_i \cup Example.class$;
6     $FormedRules = FormedRules \cup FormedRule$;
7 return $FormedRules$;
```

is repeated until there are no more unclassified examples left. If the number of conditions reaches the maximum value, which is equivalent to n_a, then the remaining unclassified examples are all considered to be rules individually.

In order to minimize the number of combinations when the condition number is greater than one, SILEA follows an approach similar to Sequential Forward Selection (Alg3). This approach eliminates the consideration of every possible combination because, in each iteration, SILEA fixes an attribute from the previous iteration. It selects $(n_c - 1)$ number of attributes with least entropies where n_c=condition number (Alg3-Ln2-4). It then finds all the combinations of these pre-selected attribute(s) with the remaining attribute(s) by appending the remaining attributes one by one to the pre-selected ones (Alg3-Ln5-8). For example, to determine combinations when condition number is 3, the two attributes with least entropy values are selected and then the remaining attributes are appended one by one to form the combinations for rule extraction. Therefore, if the number of attributes within the example set is 5, the combinations to be considered for rule extraction is {(Attr1, Attr2, Attr3); (Attr1, Attr2, Attr4); (Attr1, Attr2, Attr5)}. For each combination, the algorithm goes through the rule extraction mechanism employed by SILEA, which is explained in detail in Section 3.1.3.

3.1.3 Rule Extraction Procedure

For each selected combination of attributes, SILEA extracts every possible rule in a simple yet accurate way. The main idea is that the attribute combinations, along with their classes for each example in the example set, are considered to be potential rules unless a contradiction among examples occurs. A contradiction is when an attribute combination belongs to different classes in the dataset. Two different sets are used throughout the extraction process, *Blacklist* (Alg1-Ln8) and *PotentialList* (Alg1-Ln9). The *Blacklist* contains attribute combinations which cannot form a rule and the *PotentialList* contains attribute combinations with their classes which are likely to be rules. Potential rules in *PotentialList* also contain the number of examples they can classify.

Initially, for each example (Alg1-Ln10), the attribute combination is checked by the algorithm whether or not it exists in the *Blacklist* (Alg1-Ln13). If it exists, then the current combination for the current example is ignored since it cannot form a rule and the next combination is considered. If the combination does not exist in the *Blacklist*, then the algorithm checks whether the combination exists in the *PotentialList* (Alg1-Ln14-22). Three cases can occur in this case. The first case is that the combination does not exist in the *PotentialList* (Alg1-Ln14). It is then added to the *PotentialList* and its occurrence value is set to 1 (Alg1-Ln15-16). The second case is that it exists in the *PotentialList* and the rule combination has the same class as the one in the *PotentialList* (Alg1-Ln17). In this case, the current combination can still be considered as a potential rule and is therefore kept in the *PotentialList* and its occurrence value is incremented by 1 (Alg1-Ln18). The third case is that it exists in the *PotentialList* and the combination has a different class than the one in the *PotentialList* (Alg1-Ln19). Thus the combination cannot form a rule and is therefore removed from the *PotentialList* and placed in

Algorithm 4: ***Filter***(*PotentialList*, *UnclassifiedExamples*)

```
1 for each Rule ∈ PotentialList do
2     $RuleClassifies = false$;
3     for each Example ∈ UnclassifiedExamples do
4         if Classifies(Rule, Example) then
5             $RuleClassifies = true$;
6             $UnclassifiedExamples = UnclassifiedExamples - Example$;
7     if !Classifies(Rule, Example) then
8         $PotentialList = PotentialList - Rule$;
```

the *Blacklist* so that next time such combinations can be ignored (Alg1-Ln20-21). After every example is visited and processed in a single pass, potential rules within the *PotentialList* form rules.

The extracted rules classify examples in descending order of their occurrence values (Alg4). If another rule also classifies all the examples that a rule can classify with a higher occurrence value, then the rule with lower occurrence value is discarded since it becomes obsolete (Alg4-Ln9-11). The algorithm also discards the examples that can be classified by the selected rules (Alg4-Ln4-7). At the end of this filtering process, the rules in *PotentialList* are added to the *SelectedRules* list (Alg1-Ln27).

The feature selection complexity of SILEA, along with some other algorithms, is provided in Table 3.1. SILEA's efficiency in terms of the number of combinations processed surpasses the algorithms it is compared. In order to visualize the complexity of feature selection of the algorithms, assume an example set where $n_a = 15$. For this particular example set, the maximum number of combinations that each algorithm considers are 32,766 for RULES3, 2,955 for RULES3-Plus and 120 for SILEA. The number of combinations considered in SILEA is considerably lower than both RULES3 and its successor RULES3-Plus algorithms.

TABLE 3.1: Algorithm Complexities

(n_a = # of attributes, m_{PRSET} = # of expressions stored in PRSET)

Algorithm	Number of combinations	Asymptotic growth
SILEA	$\frac{1}{2}n_a(n_a+1)$	$O(n^2)$
RULES3-Plus	$n_a + m_{PRSET}\sum_{i=1}^{n_a-1} n_a - 1$	$O(n^3)$
RULES3	$\sum_{i=1}^{n_a} \frac{n_a!}{(n_a-i)!i!}$	$O(nn!)$

3.2 Illustrative Problem

The Car Acceleration dataset [90] is used to illustrate the execution of the SILEA algorithm. The dataset consists of examples where the algorithm must execute all of its cases at some time during the extraction, which makes it easier to demonstrate how the algorithm functions in different cases. The dataset consists of three attributes and three classes. The attributes are; Fuel, Max-Speed and Car-Size. A combination of these attributes corresponds to a particular acceleration performance of the car, which could be good, excellent or poor.

The example set is given in Table 3.2 where attributes are sorted according to their entropies. In this example set, Fuel has the lowest entropy while Car-Size has the highest. Quantization is not applied since there are no numerical attributes in the dataset. A minimum number of conditions is set to 1.

For each example in the dataset, the algorithm goes through the following steps:

For each example, the *FormedRules* list is formed and filled by the ***Generate-FormedRules*** function. This list contains the expressions generated from the

TABLE 3.2: Example Set

Example	Fuel	Max-Speed	Car-Size	Acceleration
1	diesel	high	large	good
2	propane	high	large	good
3	petrol	high	compact	excellent
4	petrol	high	large	excellent
5	diesel	low	medium	good
6	petrol	low	compact	good
7	petrol	average	medium	excellent
8	diesel	average	medium	poor

example and is checked whether or not they form a rule. For the first example in the set, the following expressions are formed;

- $Fuel = diesel \rightarrow Acceleration = good$
- $Max\text{-}Speed = high \rightarrow Acceleration = good$
- $Car\text{-}Size = large \rightarrow Acceleration = good$

Since there are no items in both the *Blacklist* and the *PotentialList*, these combinations are added to the *PotentialList* and their occurrences are set to 1 for each one of them as shown in Table 3.3;

In the second example, the following expressions are formed;

- $Fuel = propane \rightarrow Acceleration = good$
- $Max\text{-}Speed = high \rightarrow Acceleration = good$
- $Car\text{-}Size = large \rightarrow Acceleration = good$

Since there are no items in the *Blacklist*, and there exist some combinations in the *PotentialList*, each one of the newly generated expressions is compared with those in the *PotentialList*. Since no potential rules are containing the attributes of the first expression, it is added to the *PotentialList* and its occurrence is set to

TABLE 3.3: *PotentialList* - Potential rules extracted from the 1st example

n_c	#	Rules
1	1	$Fuel = diesel \rightarrow Acceleration = good$
1	1	$Max\text{-}Speed = high \rightarrow Acceleration = good$
1	1	$Car\text{-}Size = large \rightarrow Acceleration = good$

1. The second and the third expressions do exist in the *PotentialList* and since they have the same class, there is no contradiction. Therefore, the rules are left in the *PotentialList* but their occurrence values are incremented. The updated list is shown in Table 3.4;

In the third example, the following expressions are formed;

- $Fuel = petrol \rightarrow Acceleration = excellent$
- $Max\text{-}Speed = high \rightarrow Acceleration = excellent$
- $Car\text{-}Size = compact \rightarrow Acceleration = excellent$

Since there are no items in *Blacklist*, the expressions are compared with items in the *PotentialList*. The first and the third expressions do not contradict any of the rules in the *PotentialList* and are therefore added to it. The second one exists in the *PotentialList* with a different class. So the contradicted rule, $Max\text{-}Speed = high \rightarrow Acceleration = good$, is removed from the *PotentialList* and

TABLE 3.4: *PotentialList* - Potential rules extracted from the 1st and 2nd examples

n_c	#	Rules
1	2	$Max\text{-}Speed = high \rightarrow Acceleration = good$
1	2	$Car\text{-}Size = large \rightarrow Acceleration = good$
1	1	$Fuel = diesel \rightarrow Acceleration = good$
1	1	$Fuel = propane \rightarrow Acceleration = good$

TABLE 3.5: *Blacklist* - Attribute name(s) which belong to more than one class

n_c	Conditions
1	$Max\text{-}Speed = high$

the attribute of this rule is added to the *Blacklist* to be ignored the next time it is observed. The lists are shown in Table 3.5 and Table 3.6;

The same process is applied to every example in the example set. After visiting every example in the set, the *PotentialList* will contain all the rules with condition number n_c, which is currently set to 1.

After the rule extraction, ***Filter*** function removes unnecessary rules from the *PotentialList*. Unnecessary rules are those that have become obsolete since the examples they can classify can be classified by other rule(s) with higher occurrence value(s). Starting from the most effective rules in the *PotentialList*, rules are checked to see whether they can classify existing unclassified examples. If the rules can classify at least one example in the *UnclassifiedExamples* set, then the examples that the rules can classify are removed from the set and the rest of the rules are checked with the remaining unclassified examples. If the rules cannot classify the unclassified examples or if there are no more examples in the *UnclassifiedExamples* set, then these rules are removed from the *PotentialList*. After the execution of the ***Filter*** function, the remaining rules in the *PotentialList*

TABLE 3.6: *PotentialList* - Potential rules extracted from the 1st, 2nd and 3rd examples

n_c	#	Rules
1	2	$Car\text{-}Size = large \rightarrow Acceleration = good$
1	1	$Fuel = diesel \rightarrow Acceleration = good$
1	1	$Fuel = propane \rightarrow Acceleration = good$
1	1	$Fuel = petrol \rightarrow Acceleration = excellent$
1	1	$Car\text{-}Size = compact \rightarrow Acceleration = excellent$

TABLE 3.7: *PotentialList* - Potential rules selected from all the examples with $n_c = 1$ and from the 1st example with $n_c = 2$

n_c	#	Rules
1	2	$Car\text{-}Size = low \rightarrow Acceleration = good$
1	1	$Fuel = propane \rightarrow Acceleration = good$
2	1	$Fuel = diesel\&Max\text{-}Speed = high \rightarrow Acceleration = good$
2	1	$Fuel = diesel\&Car\text{-}Size = large \rightarrow Acceleration = good$

are added to the *SelectedRules* list.

Since there remain unclassified examples, the number of conditions is incremented, the first attribute is fixed since ($n_c - 1 = 1$), and all the combinations with the remaining attributes are calculated. The following expressions are generated from the first example by the ***GenerateFormedRules*** function;

- $Fuel = diesel\&Max\text{-}Speed = high \rightarrow Acceleration = good$
- $Fuel = diesel\&Car\text{-}Size = large \rightarrow Acceleration = good$

Since there are no items in both the *Blacklist* and the *PotentialList*, these combinations are added to the *PotentialList* and their occurrences are set to 1 as shown in Table 3.7;

All the other examples are visited one by one going through the same steps as above. Rules in Table 3.8 are what are left in the *SelectedRules* list at the end of extraction and selection processes.

TABLE 3.8: *SelectedList* - All selected rules from the dataset

n_c	#	Rules
1	2	$Car\text{-}Size = low \rightarrow Acceleration = good$
1	1	$Fuel = propane \rightarrow Acceleration = good$
2	2	$Fuel = petrol\&Max\text{-}Speed = high \rightarrow Acceleration = excellent$
2	1	$Fuel = diesel\&Max\text{-}Speed = average \rightarrow Acceleration = poor$
2	1	$Fuel = diesel\&Max\text{-}Speed = high \rightarrow Acceleration = good$
2	1	$Fuel = petrol\&Max\text{-}Speed = average \rightarrow Acceleration = excellent$

Since the rules in the *SelectedRules* list can classify every example in the dataset, the algorithm selects all the rules in the *SelectedRules* list and terminates.

3.3 Experimental Results

In this section, we compare the classification accuracy of SILEA to some of the well-known algorithms in the inductive learning field using five different datasets. The datasets used for evaluation are; Balloons [91], Hayes-Roth [92], Hepatitis [93], Iris [94] and Lenses [95]. For each of the datasets, we generated ten randomly sorted versions of the datasets and recorded their average accuracy along with the average number of rules each algorithm extracted for the given dataset. The dataset was split as 60% and 40% for training and testing, respectively.

Tables 3.9, 3.10, 3.11, 3.12 and 3.13 present a number of experiments and their accuracy. The number of conditions set for SILEA and RULES algorithms is equal to 1 for all the datasets and the quantization levels set for the Iris dataset is 3 and for the rest of the datasets is 5. RULES3-Plus requires an additional

TABLE 3.9: Accuracy for the Balloons dataset

Algorithms	Avg. # of rules	μ	σ
SILEA	3	**100%**	0
RULES3	3	**100%**	0
RULES3-Plus	6	**100%**	0
C4.5	3	90.0%	16.1
CN2	3	**100%**	0
RIPPER	2	84.4%	21.08
RIDOR	2	71.1%	13.04
PART	3	90.0%	16.1
DecisionTable	4	86.7%	17.21
RandomTree	6	88.9%	18.89

TABLE 3.10: Accuracy for the Hayes-Roth dataset

Algorithms	Avg. # of rules	μ	σ
SILEA	24	**84.1%**	3.92
RULES3	21	72.3%	4.5
RULES3-Plus	46	64.4%	3.68
C4.5	12	80.7%	5.4
CN2	18	70.9%	9.05
RIPPER	6	72.8%	8.05
RIDOR	7	68.0%	12.56
PART	9	78.3%	7.46
DecisionTable	6	55.0%	3.03
RandomTree	50	74.4%	8.81

parameter to be set, which is called PRSET. For each dataset, this value was set equal to the number of attributes of the dataset's examples in order to assure the algorithm generates the highest accuracy possible. The remaining algorithms were run with their default parameter settings as they are the ones that were suggested to be used.

Even though the number of attribute combinations to be considered is reduced in SILEA to O(n^2), it still was able to perform either as well or better than other

TABLE 3.11: Accuracy for the Hepatitis dataset

Algorithms	Avg. # of rules	μ	σ
SILEA	23	**82.6%**	3.47
RULES3	33	73.3%	4.92
RULES3-Plus	41	73.7%	4.63
C4.5	5	78.2%	2.66
CN2	12	79.0%	2.4
RIPPER	3	79.0%	3.65
RIDOR	3	78.5%	4.44
PART	7	79.4%	3.63
DecisionTable	13	76.8%	2.43
RandomTree	48	76.6%	5.65

TABLE 3.12: Accuracy for the Iris dataset

Algorithms	Avg. # of rules	μ	σ
SILEA	5	**96.7%**	2.36
RULES3	5	86.1%	4.07
RULES3-Plus	19	**96.7%**	2.36
C4.5	4	93.5%	3.46
CN2	5	94.5%	3.41
RIPPER	4	92.3%	2.85
RIDOR	3	93.7%	3.41
PART	4	93.8%	4.01
DecisionTable	3	94.3%	2.96
RandomTree	12	94.8%	2

algorithms on averages of all 5 cases as seen in Tables 3.14. Considering average accuracy for all datasets, SILEA was 8.8% better than RULES3, 12.1% better than RULES3-Plus, 4.2% better than C4.5, 5.8% better than CN2, 9.8% better than RIPPER, 14.4% better than RIDOR, 4.4% better than PART, 13.7% better than DecisionTable, and 8.0% better than RandomTree algorithms. Standard deviations show that the accuracy of SILEA, in most cases, were either close to or more stable than the other algorithms.

Compared to similar algorithms like the RULES algorithms, SILEA was able to

TABLE 3.13: Accuracy for the Lenses dataset

Algorithms	Avg. # of rules	μ	σ
SILEA	6	**80.0%**	13.33
RULES3	6	68.0%	15.49
RULES3-Plus	9	48.0%	14.76
C4.5	3	**80.0%**	12.47
CN2	4	70.0%	9.43
RIPPER	2	66.0%	5.16
RIDOR	2	60.0%	14.91
PART	3	**80.0%**	12.47
DecisionTable	2	62.0%	13.17
RandomTree	11	69.0%	18.53

TABLE 3.14: Average accuracy

Algorithms	μ
SILEA	**88.7%**
RULES3	79.9%
RULES3-Plus	76.6%
C4.5	84.5%
CN2	82.9%
RIPPER	78.9%
RIDOR	74.3%
PART	84.3%
DecisionTable	75.0%
RandomTree	80.7%

achieve these accuracies by reducing the number of rules in 4 out of 5 datasets. The difference in terms of the number of rules generated between SILEA and the other algorithms on average is minimal, which shows that SILEA can extract as general rules as possible for each iteration with these datasets. SILEA's method of rule selection among the generated rules also work accurately since its accuracy has surpassed the other algorithms with the analyzed datasets.

Chapter 4

Operating System Classification Performance of TCP&IP Protocol Headers

Network measurement and management requires an understanding of devices connected to a network [96]–[98]. Operating system (OS) fingerprinting is the process of remotely detecting the OS of a target IP address. Note that a system might have multiple IP addresses assigned, which would require IP alias resolution [99]. There are two methods for performing OS classification: active and passive [63]. Active fingerprinting actively probes target devices and analyzes their responses [66]. Certain systems can respond in an unusual way upon specific requests. Such scenarios can allow for active fingerprinting tools to narrow down possibilities or even directly determine the OS of such systems. However, firewalls could block probes, and active fingerprinting might lead to limited or no knowledge of the target host. Passive fingerprinting passively sniffs packets from target devices and analyzes the packet header information. As passive fingerprinting merely depends on the information extracted from regular TCP/IP packet headers, it is possible to perform OS classification even when a firewall exists. On the other hand, passive fingerprinting might not be as accurate as active fingerprinting as it cannot observe distinct features. Passive fingerprinting techniques can take advantage of less than one-third of the features that active fingerprinting tools such as Nmap offers [11].

It can become impractical for system administrators to manage and monitor a vast amount of hosts which might be at different locations [13]. Therefore, OS fingerprinting tools and techniques can ease network management and security [18]. These tools may also be useful for collecting statistical data on the OSes that host devices within a network use. OS fingerprinting, however, can also be used for malicious purposes. It can allow intruders to detect devices and identify their vulnerabilities in a target network. When intruders detect the presence of unpatched or outdated OSes, they can easily deploy known malware to compromise them.

While current OS classification systems are expert-based, automated detection and configuration of networked systems are valuable [100], [101]. To this end, machine learning approaches can provide automated system configuration with minimal or no expert input [15].

In this section, we analyze the accuracy of TCP/IP headers in classifying OSes with machine learning. We perform a single-packet OS classification on packets originating from host devices. We compare the classification results for several protocols at layer 3 (i.e., IP and ICMP), layer 4 (i.e., TCP and UDP), and layer 5 (i.e., HTTP, DNS, SSL, SSH, and FTP) using multiple machine learning algorithms in order to determine their contribution in classifying OSes.

We tested the classification both with every non-null feature extracted from the protocol and with features that were selected by a genetic algorithm (GA). GA allows us to perform OS classification of packets with a less computational overhead with no significant degradation in classification accuracy. After selecting features that help classify the packet, we use different machine learning algorithms to detect the OS of the host devices. With the help of GA, we detected the features that contribute most to the classification of OSes. Using GA and machine learning, we identify TCP/IP header features that can guide OS classification.

Our results, in general, are consistent with an expert system based OS fingerprinting tools such as p0f, ettercap, and siphon [11]. While current tools that depend on specific packet types such as SYN, ACK, and SYN-ACK, our classifier can detect OSes of individual packets regardless of the packet type. In this measurement study, we performed OS classification with all types of packet observed from host devices.

4.1 Methodology

In this section, we measure the operating system (OS) classification accuracy of TCP/IP protocol headers using machine learning. We use the GA feature selection technique for determining the relevant features from TCP/IP protocol headers. After determining the protocol features, we used machine learning algorithms to populate a set of rules using the full and selected set of features.

We initially collected the set of all features to be used for performing OS classification. We then eliminated features that contained null values along with the ones that were machine learning incompatible such as cookie value in HTTP, the domain name in DNS, etc. and the ones that were machine dependent such as MAC address in DHCP, user agent in HTTP, etc.

For the implementation of the fitness function of the GA, we employed the wrapper method of feature selection [102]. The fitness function uses the trainer itself to determine the accuracy of feature combinations generated by the GA. We then determined the TCP/IP protocol header features that led to the best OS classification.

4.1.1 Data Initialization

We set up a local network consisting of 4 computers. Three of these computers run instances of Fedora 23, Xubuntu 14.04, Windows 7 and Windows 8 and the fourth computer runs an instance of OSX El Capitan. We collected packets from multiple devices to remove any possible bias from the collected packets towards one instance of the OSes. Every instance of OSes on the machines that we collected packets from was freshly installed. We did not use VirtualMachine in order to generate as realistic scenarios as possible and used the Wireshark tool to collect packets.

To generate HTTP protocol packets, we visited the same websites on every OS. These websites were;

`http://www.google.com`, `http://www.yahoo.com`,
`http://www.unr.edu` and `http://www.youtube.com`.

We also collected approximately 20 minutes of YouTube video streaming. To generate FTP protocol packets, we connected to `ftp://ftp.godaddy.com` FTP server and uploaded files to the server. To generate ICMP packets, we used the “traceroute” application to connect to all 4 of the domain names mentioned earlier. To generate SSH packets, we initiated connections to an SSH server and transmitted files to the server. The remaining protocols such as; IP, TCP, UDP, DNS, and SSL were observed among the collection of packets for the protocols mentioned above.

The number of packets collected for each protocol is provided in Table 4.1. The number of packets among different protocols differs as different protocols have different popularity in real network flows. 5% of the dataset was dedicated to the training of the GA's fitness function, and another 5% was dedicated to the testing of the GA. The remaining 90% of the packets were split into five parts to perform a 5-fold cross-validation test. Four of the five parts of the packets were used to train the system, and the remaining one was used to test it. This process was performed for every combination of these parts of data, and their average accuracy was recorded.

For packet collection, packets containing specific protocols were generated and collected within this local network. These protocols include; HTTP, DNS, SSL, FTP, SSH, ICMP, UDP, TCP, and IP. For each protocol, feature selection and machine learning classification were performed for two sets. One of them used every possible feature of the provided protocol, and the other one used the GA selected features of this protocol. For each test, features of the protocols were collected, and those features with only null values were eliminated since they do not contribute to the classification.

SSH and FTP protocol packets seemed to have many packets with similar headers, and after the removal of the duplicate packets, there was an insufficient number of packets for GA. Therefore, we did not run the GA feature selection technique for SSH and FTP protocols. We calculated the classification accuracy of these protocols using all available features.

TABLE 4.1: Number of packets

Protocols	GA train	GA test	Train	Test	Total
IP	4,711	4,711	339,280	84,820	433,522
ICMP	39	39	2,900	725	3,703
TCP	5,583	5,583	402,100	100,525	513,791
UDP	711	711	51,240	12,810	65,472
HTTP	308	308	22,320	5,580	28,516
DNS	561	561	40,580	10,145	51,847
SSL	201	201	14,660	3,665	18,727
SSH	-	-	60	15	75
FTP	-	-	100	25	125

4.1.2 Feature Selection

In addition to testing OS classification accuracy with every feature determined from the protocols, we tried to reduce the number of features to be considered. We use a GA to determine a subset of features that maintains high classification accuracy. GA is the process of searching and testing the accuracy of a solution among a space of solutions [103]. The idea is based on the biological mechanisms of natural selection and reproduction. GA uses an objective (or fitness) function to evaluate every solution it finds. This process is performed until a certain criterion is met. In our case, the criterion was the generation of 15 consecutive identical solutions. The fitness function to evaluate the solutions is:

$$\begin{aligned} Fitness = {} & 0.80 \times Accuracy + \\ & 0.15 \times \left(1 - \frac{|SelectedFeatures| - 1}{|AllFeatures| - 1}\right) + \\ & 0.05 \times \left(1 - \frac{|SelectedRules| - 1}{|AllRules| - 1}\right) \end{aligned}$$

where *Accuracy* is a measure of the classification accuracy with the provided machine learning algorithm.

Different weight values for accuracy, features, and rules were tested. Since we aim to optimize classification accuracy, we set the weight for accuracy to 80%. We also wanted to use as few features as possible while maintaining high classification accuracy. Therefore, we set the weight for the number of features to 15%. To end up with the fewest rules or exemplars as possible, we set the weight for the number of rules to be extracted by the classifier to 5%.

4.1.3 Rule Extraction

After selecting the features to be used, we analyzed different machine learning algorithms to perform OS classification. We used the WEKA tool [104] for classification. We utilized a set of algorithms in the WEKA tool, namely; J48, JRip, Ridor, PART, DecisionTable, RandomForest, NaiveBayes, and MultilayerPerceptron.

4.2 Experimental Results

In this section, we provide the classification accuracy of different layer 3 (i.e., IP and ICMP), layer 4 (i.e., TCP and UDP), and layer 5 (i.e., HTTP, DNS, SSL, SSH, and FTP) protocols. We demonstrate classification accuracy with all the extracted features from the protocol header and with features selected by GA. The fitness function for the GA tries to optimize the classification accuracy with the smallest possible number of features and number of rules. Table 4.2 shows the number of features that initially existed in our dataset along with the number of GA-selected features for each protocol. As seen in the table, GA was able to decrease the number of features immensely.

TABLE 4.2: Number of selected features

Protocol	All	J48	JRip	Ridor	PART	DT	RF	NB	MP
IP	23	5	6	4	6	4	9	3	5
ICMP	20	4	10	5	3	1	5	5	5
TCP	62	7	8	7	12	4	10	12	-
UDP	7	1	3	1	1	3	1	2	2
HTTP	17	3	5	1	7	1	2	3	7
DNS	34	1	8	1	5	5	12	3	2
SSL	31	1	10	4	4	3	2	7	11
SSH	18	-	-	-	-	-	-	-	-
FTP	4	-	-	-	-	-	-	-	-

The use of GA for selecting features had a significant impact on the results that we generated. As presented in Table 4.2, the number of features that algorithms used for the classification of operating systems (OS) are much lower than the initial number of features. It can be expected for the classification accuracy to drop when the number of features is immensely reduced. However, GA can select the ones that are most helpful in identifying the OSes. As shown in the following protocol evaluations, there are even cases where the classification accuracy improves when a GA is used.

4.2.1 IP Protocol Headers

IP protocol is among the ones that provide an acceptable amount of uniqueness to the classification of OSes as seen in Figure 4.1. Error bars present the *min* and *max* accuracy of the algorithm while the bars present the average of 5 evaluations. On average of all algorithms, we observe 68.0% accuracy when all features are considered, while accuracy becomes 67.5%, on average, when a subset of features is utilized. At its best, it has 76.2% classification accuracy with the J48 algorithm.

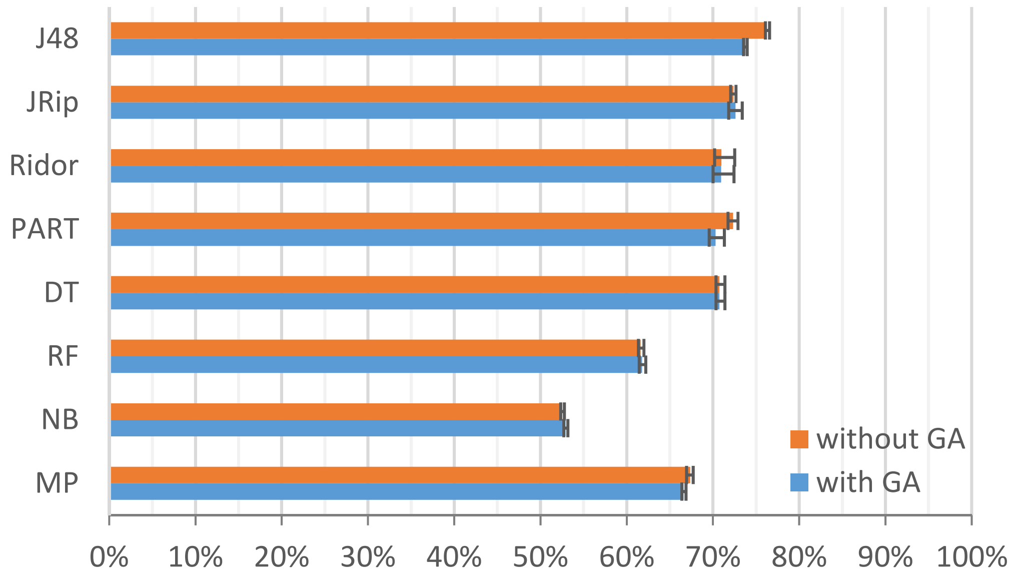


FIGURE 4.1: IP Accuracy

Table 4.3 presents IP protocol features that were selected by different machine learning algorithms. As seen in the table, among the features extracted from the IP protocol packets, the Checksum (checksum) feature seems to be the most preferred feature by all of the algorithms for determining the uniqueness of different OSes. In general, the checksum contains distinguishing information about packets since it contains a summary of the protocol header.

TABLE 4.3: Selected IP Features

Features	J48	JRip	Ridor	PART	DT	RF	NB	MP	Σ
checksum	✓	✓	✓	✓	✓	✓	✓	✓	8
ttl		✓	✓	✓	✓	✓	✓	✓	7
id	✓	✓	✓	✓	✓	✓		✓	7
len	✓	✓	✓			✓		✓	5
proto	✓	✓		✓					3
opt.ra	✓							✓	2
dsfield.dscp				✓		✓			2
frag_offset				✓		✓			2
dsfield.ecn						✓			1
flags.df						✓			1
opt.type						✓			1
hdr_len							✓		1
opt.type.number					✓				1
checksum_bad		✓							1
Σ (features)	5	6	4	6	4	9	3	5	

Along with checksum, the ID and TTL features are among the most selected features by the algorithms. IP ID is the identification number of a packet, and it is often incremented for every packet sent from the OS. As long as its values do not overlap for packets from multiple OSes, it is expected for the feature selection algorithms to detect it as a unique feature. One of the reasons for collecting packets from multiple devices was to eliminate such biases, but it is challenging to eliminate them entirely.

TTL is used to prevent infinite loops on the Internet where every router decrements this value until it becomes 0 or reaches its intended destination. As the second most common differentiator, it seems OSes prefer different initial values for this feature and hence TTL becomes useful to classify OSes based on IP protocol header information.

The fourth most important feature is the Total Length (len) feature, which contains the number of 32-bit words in the header. Depending on the content's uniqueness in every packet, this particular feature can aid in determining the OS from which the packet is originated.

Even though the number of features GA extracted for each algorithm differs, Ridor seems to perform well using only the four most popular features indicated earlier. It was even able to outperform the RandomForest algorithm using less than half of the features RandomForest selected.

According to [11], TTL, and Total Length are among the features open-source tools use, and we can observe their importance in our evaluations as well. However, another feature that was mentioned in this section was the Don't Fragment bit, but a single algorithm only selected it in our experiments.

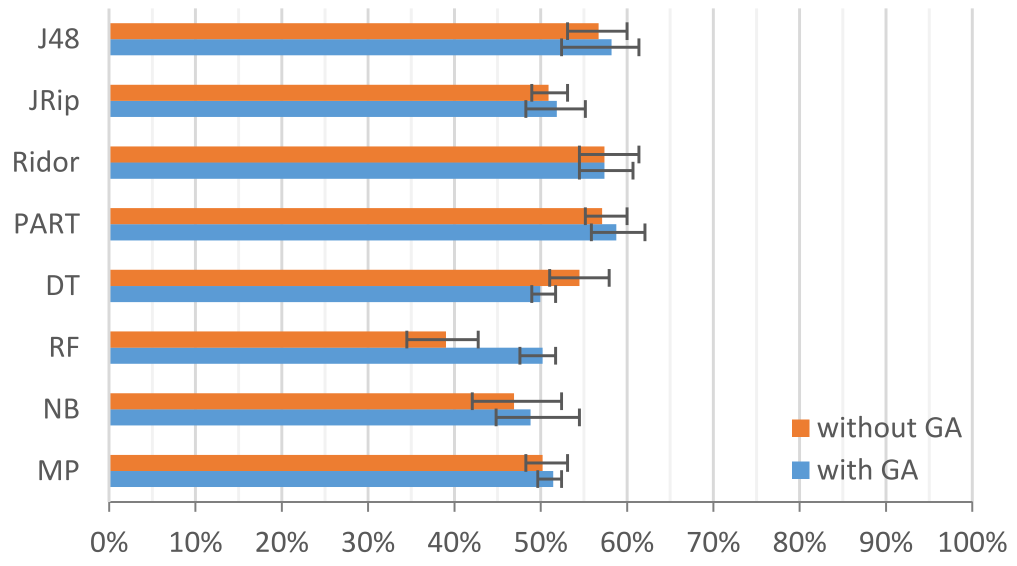


FIGURE 4.2: ICMP Accuracy

4.2.2 ICMP Protocol Headers

ICMP protocol does not contribute much to the classification of OSes, as seen in Figure 4.2. On average of all algorithms, we observe 51.6% accuracy when all features are considered, while accuracy becomes 53.3%, on average, when a subset of features is utilized. At its best, it has 58.8% classification accuracy with the PART algorithm on selected features.

Similar to the IP protocol, the most preferred feature by the algorithms is the checksum, as seen in Table 4.4. However, the results for the DecisionTable algorithm show us that the accuracy of 54.4% can be achieved with just the identification feature. The identifier feature for an ICMP packet provides unique classification, above the average of all algorithms that have additional features.

TABLE 4.4: Selected ICMP Features

Features	J48	JRip	Ridor	PART	DT	RF	NB	MP	Σ
checksum	✓	✓	✓	✓		✓	✓	✓	7
type		✓		✓		✓	✓	✓	5
mpls.s	✓	✓				✓		✓	4
checksum_bad		✓	✓					✓	3
ext.class	✓	✓							2
ext.ctype		✓	✓						2
ext.length		✓	✓						2
ext.res							✓	✓	2
ext.version				✓			✓		2
ident					✓	✓			2
ext		✓							1
ext.checksum		✓							1
mpls.label		✓							1
seq							✓		1
code						✓			1
mpls.ttl			✓						1
mpls.exp	✓								1
Σ (features)	4	10	5	3	1	5	5	5	

4.2.3 TCP Protocol Headers

TCP protocol has the most distinguishing header to accurately classify OSes, as shown in Figure 4.3. Note that, MultilayerPerceptron algorithm could not complete its classification due to its considerably high complexity and a large number of features and TCP packets. Hence, it is omitted in TCP analysis. On average of all algorithms, we observe 98.4% accuracy when all features are considered, while accuracy becomes 99.1%, on average, when a subset of features is utilized. At its best, it has 99.9% classification accuracy with the PART, Ridor, Jrip, and J48 algorithms with both all features and selected features. Overall, we realize that TCP is a good differentiator of OSes as it is a complex protocol with various features that are implemented differently by OSes.

Although the number of features used by certain algorithms can reach up to 12, as seen in Table 4.5, some algorithms were able to classify at a very close accuracy rate with fewer features. For example, the DecisionTable algorithm

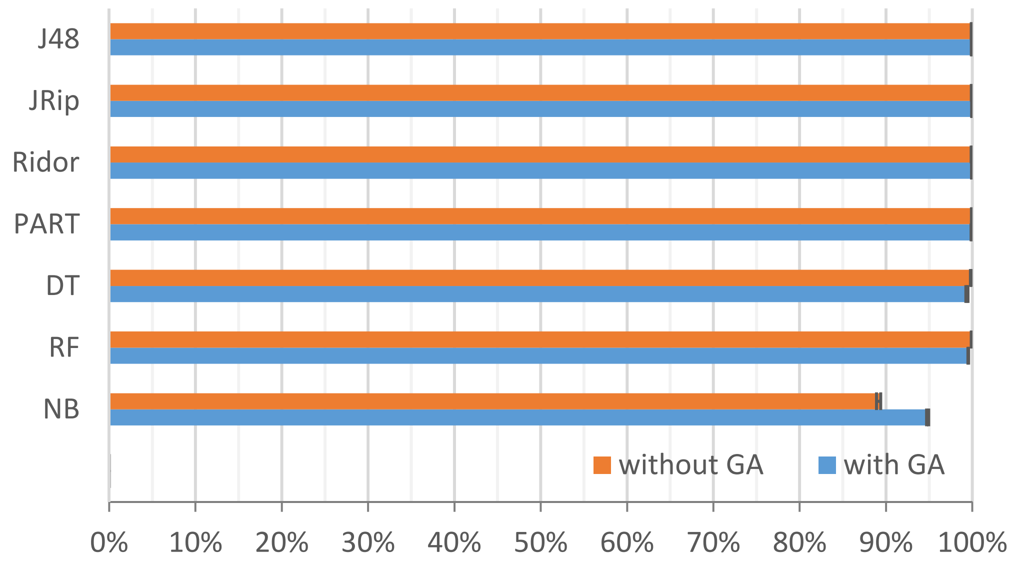


FIGURE 4.3: TCP Accuracy

was able to perform classification at a rate of 99.4% using only four features, which is just 0.5% less than the PART algorithm, which used 12 features. The four features extracted by the GA using the DecisionTable algorithm are; the source or destination port (port), analysis.out_of_order, analysis.retransmission, and copy on fragmentation (options.type.copy).

According to [11], Window size (WS), TCP Max Segment Size Option* (MSS), TCP Window Scaling Option Flag* (WSO), TCP Selective Acknowledgments Options Flag* (SOK), TCP NOP Option Flag* (NOP) and TCP Timestamp Option Flag* (TS) are among the features open-source tools use. In our experiments, however, the Window size was selected by only the PART algorithm, and none of the algorithms selected TCP Max Segment Size Option. As for the TCP Window Scaling Option Flag, similar to what the tools use, four of the algorithms extracted the (window_size_scalefactor) feature. As opposed to TCP Selective Acknowledgments Options Flag alone, our GA implementation selected (options.sack.count), (options.sack_le), (options.sack_perm) and (options.sack_re) features. These features were also selected by single algorithms

TABLE 4.5: Selected TCP Features

Features	J48	JRip	Ridor	PART	DT	RF	NB	Σ
stream	✓	✓	✓	✓			✓	5
options.timestamp.tsval	✓	✓	✓	✓				4
window_size_scalefactor	✓			✓		✓	✓	4
port		✓			✓	✓	✓	3
srcport	✓		✓	✓				3
analysis.duplicate_ack_frame				✓			✓	2
analysis.duplicate_ack_num		✓				✓		2
analysis.out_of_order					✓		✓	2
analysis.rto_frame				✓			✓	2
flags		✓		✓				2
flags.reset				✓		✓		2
option_kind		✓				✓		2
nxtseq		✓					✓	2
flags.push							✓	1
option_len							✓	1
options.wscale.multiplier							✓	1
options.wscale.shift							✓	1
pdu.size							✓	1
ack						✓		1
segment.count						✓		1
seq						✓		1
dstport						✓		1
flags.cwr						✓		1
options.type.copy					✓			1
analysis.retransmission					✓			1
window_size_value				✓				1
options.sack_perm				✓				1
analysis.zero_window				✓				1
analysis.bytes_in_flight				✓				1
options.timestamp.tsecr			✓					1
options.sack.count			✓					1
checksum_good			✓					1
analysis.duplicate_ack			✓					1
options.sack_le		✓						1
reassembled.length	✓							1
options.sack_re	✓							1
hdr_len	✓							1
Σ (features)	7	8	7	12	4	10	12	

individually. Four of utilized machine learning algorithms selected the (options.timestamp.tsval) feature, and this feature has an important impact on our results.

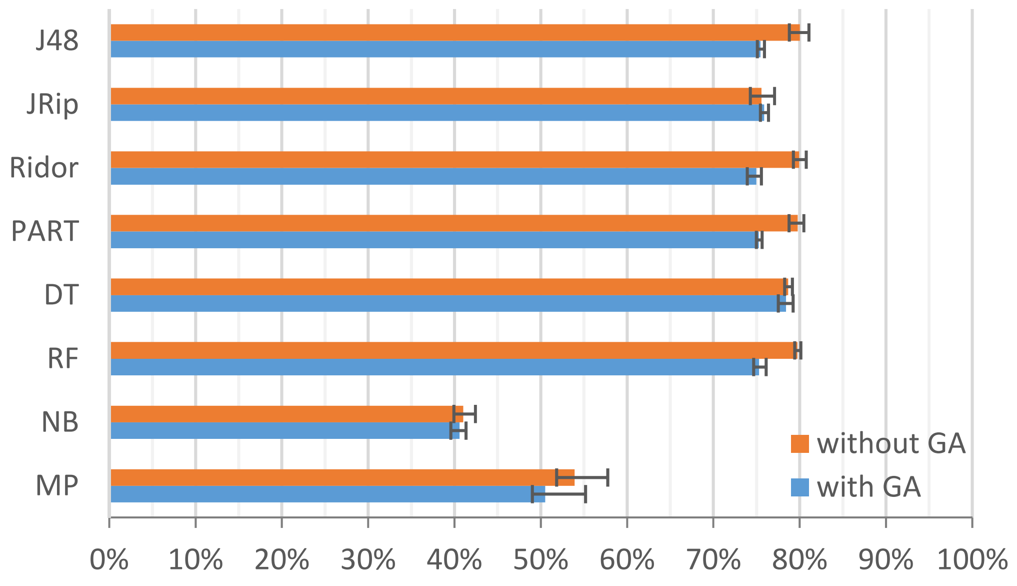


FIGURE 4.4: UDP Accuracy

4.2.4 UDP Protocol Headers

UDP protocol yielded different accuracy with different machine learning algorithms, as shown in Figure 4.4. On average, we observe 71.1% accuracy when all features are considered, while accuracy becomes 68.3%, on average, when a subset of features are utilized. At best, it has 80.0% classification accuracy with the J48 algorithm while several algorithms have very similar results.

Overall, there are only four UDP features utilized by the algorithms, as shown in Table 4.6, these features allow classification at a rate of around 80%. Every algorithm selects the checksum feature. As indicated earlier, this is likely because the checksum contains a summary of the entire packet. Even though certain

TABLE 4.6: Selected UDP Features

Features	J48	JRip	Ridor	PART	DT	RF	NB	MP	Σ
checksum	✓	✓	✓	✓	✓	✓	✓	✓	8
srcport		✓			✓			✓	3
port		✓					✓		2
dstport					✓				1
Σ (features)	1	3	1	1	3	1	2	2	

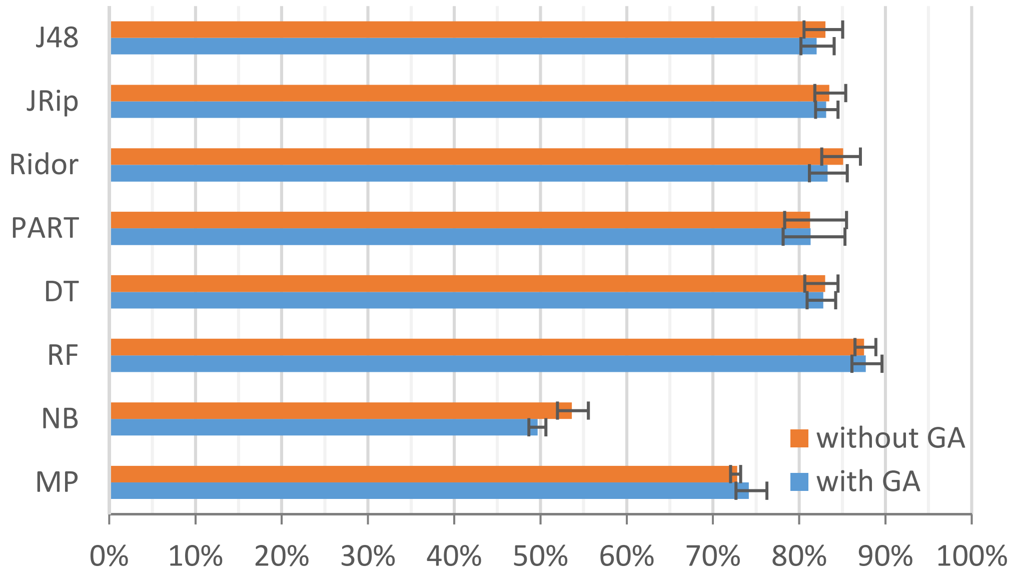


FIGURE 4.5: HTTP Accuracy

algorithms also selected other features, the results for J48 shows that even the checksum alone can achieve classification accuracy similar to other algorithms that use additional features.

4.2.5 HTTP Protocol Headers

HTTP protocol is another protocol that yielded an acceptable accuracy with most of the machine learning algorithms, as shown in Figure 4.5. On average, we observe 78.7% accuracy when all features are considered, while accuracy becomes 78.0%, on average, when a subset of features is utilized. At its best, it has 87.7% classification accuracy with the RandomForest algorithm with a subset of features.

For the HTTP protocol, as seen in Table 4.7, (prev_request_in) feature is selected for the majority of algorithms. This feature is the previous request in a frame and is represented as a frame number. There are a few other features that were

TABLE 4.7: Selected HTTP Features

Features	J48	JRip	Ridor	PART	DT	RF	NB	MP	Σ
prev_request_in	✓	✓	✓	✓	✓	✓		✓	7
accept_encoding	✓	✓		✓			✓	✓	5
cache_control	✓			✓			✓	✓	4
request.method				✓			✓	✓	3
response		✓				✓			2
content_length		✓							1
connection		✓							1
content_length_header				✓					1
accept				✓					1
content_type				✓					1
notification								✓	1
prev_response_in								✓	1
request								✓	1
Σ (features)	3	5	1	7	1	2	3	7	

selected by most of the algorithms, namely; (accept_encoding), (cache_control) and (request.method).

4.2.6 DNS Protocol Headers

Among the protocols analyzed in this study, **DNS protocol** was unsuccessful in correctly classifying the OS of the individual packets as seen in Figure 4.6. On average, we observe 29.2% accuracy when all features are considered, while accuracy becomes 30.0%, on average, when a subset of features is utilized. At its best, it has 30.9% classification accuracy with the J48 algorithm.

The GA-selected features for the DNS protocol, as seen in Table 4.8, seem to be distributed among different machine learning algorithms. This inconsistent behavior is reflected in the poor classification results in Figure 4.6. GA was unable to find very distinguishing features, and therefore for each algorithm, selected features do not necessarily have a pattern. J48 and Ridor algorithms reached average accuracy by using only a single feature which is the (flags.truncated)

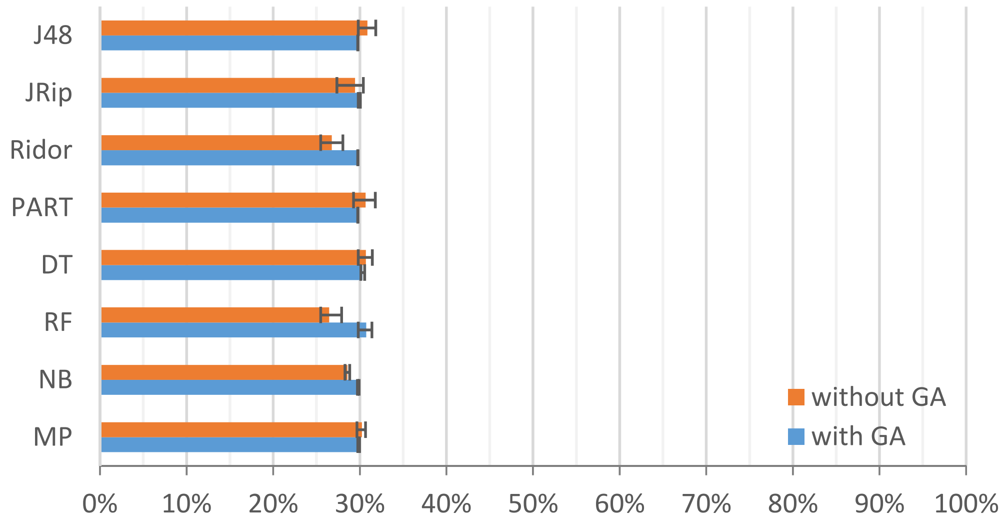


FIGURE 4.6: DNS Accuracy

and (flags.rcode) fields, respectively. Hence, while the DNS header is not sufficiently promising to be used by itself, certain features of the header can be used

TABLE 4.8: Selected DNS Features

Features	J48	JRip	Ridor	PART	DT	RF	NB	MP	Σ
soa.retry_interval		✓			✓	✓		✓	4
count.answers		✓			✓	✓			3
resp.cache_flush		✓			✓	✓			3
soa.serial_number		✓		✓		✓			3
flags.checkdisable		✓		✓					2
resp.len		✓		✓					2
flags.truncated	✓			✓					2
flags.conflict					✓	✓			2
flags.recdesired						✓	✓		2
qry.type						✓		✓	2
count.auth_rr						✓			1
flags						✓			1
flags.authoritative						✓			1
resp.class						✓			1
soa.expire_limit						✓			1
flags.opcode							✓		1
flags.recavail							✓		1
flags.authenticated					✓				1
resp.ttl				✓					1
flags.rcode			✓						1
qry.class		✓							1
soa.refresh_interval		✓							1
Σ (features)	1	8	1	5	5	12	3	2	

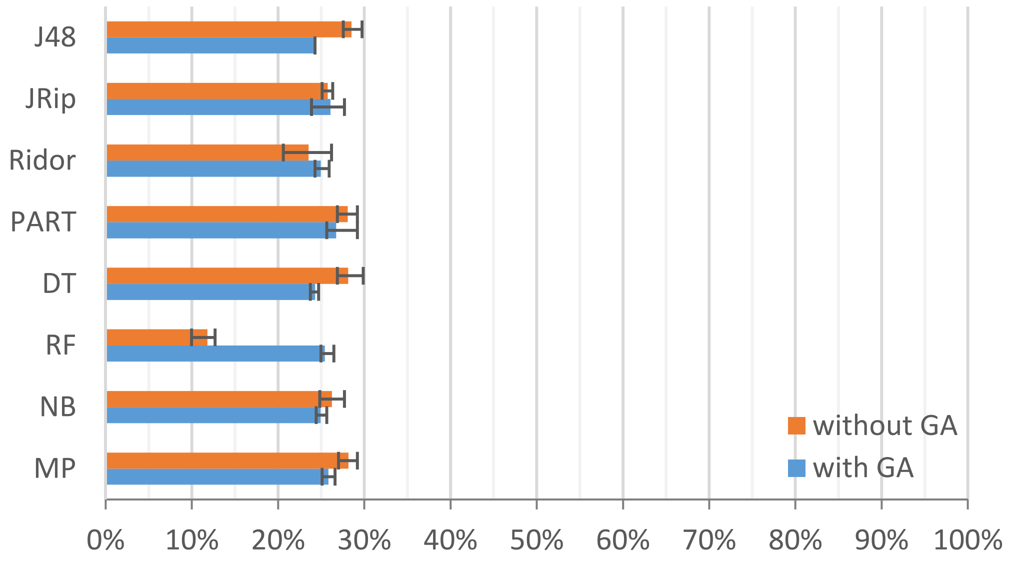


FIGURE 4.7: SSL Accuracy

to determine the OS of packets.

4.2.7 SSL Protocol Headers

On average, **SSL protocol** classification results were even lower than the DNS protocol, as shown in Figure 4.7. The average accuracy over all algorithms is 25.0% when all features are considered and becomes 25.3% when a subset of features are utilized. At best, it has 28.5% classification accuracy with the J48 algorithm.

Feature selection among different algorithms are also scattered around, and there is no consistency among algorithms for the features, as shown in Table 4.9. Only Cipher Suites Length (handshake.cipher_suites_length) is used by half of the algorithms for classification while Server Name length (handshake.extensions_server_name_len) gives close to the average accuracy by itself under the J48 algorithm.

TABLE 4.9: Selected SSL Features

Features	J48	JRip	Ridor	PART	DT	RF	NB	MP	Σ
handshake.cipher_suites_length			✓	✓		✓		✓	4
handshake.extensions_server_name_len	✓	✓						✓	3
handshake.session_id_length				✓	✓			✓	3
handshake.ciphersuite		✓					✓	✓	3
handshake.extension.len			✓	✓			✓		3
change_cipher_spec		✓	✓						2
handshake.length						✓	✓		2
record		✓					✓		2
handshake							✓	✓	2
handshake.extension.type							✓	✓	2
handshake.extensions_elliptic_curves		✓						✓	2
record.version		✓						✓	2
record.content_type		✓	✓						2
record.length		✓		✓					2
alert_message.desc		✓							1
alert_message.level								✓	1
handshake.extensions_elliptic_curves_length								✓	1
handshake.type								✓	1
handshake.ciphersuites					✓				1
handshake.extensions_ec_point_format					✓				1
handshake.extensions_ec_point_formats_length							✓		1
handshake.extensions_length		✓							1
Σ (features)	1	10	4	4	3	2	7	11	

4.2.8 SSH Protocol Headers

SSH protocol is the second least successful in the classification of the OSes from the packet header, as shown in Figure 4.8. On average, over all algorithms, we were able to correctly identify the OS of 22.5% of SSH packets. The reason for such low accuracy maybe because most of the packets' header information was identical, which yielded very few unique packets to train the machine learning algorithms. We did not select features with the GA because an insufficient amount of packets remained.

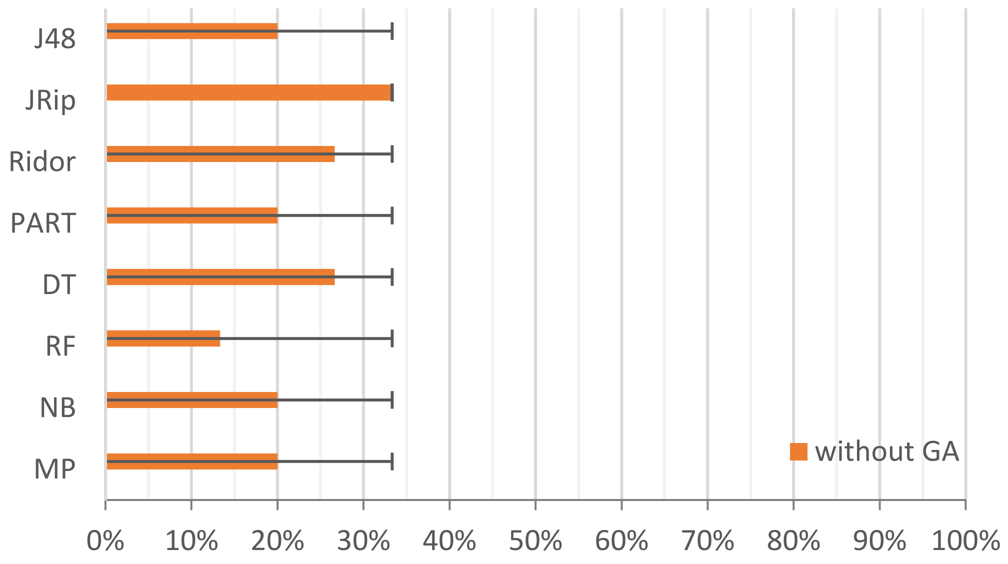


FIGURE 4.8: SSH Accuracy

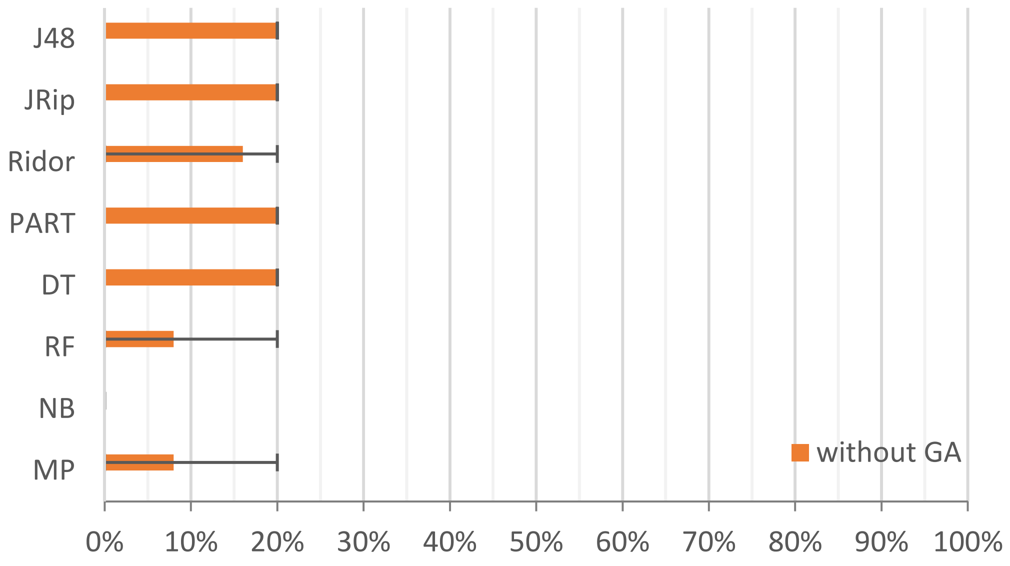


FIGURE 4.9: FTP Accuracy

4.2.9 FTP Protocol Headers

FTP protocol is the worst protocol header for the classification of the OSes as shown in Figure 4.9. On average, over all algorithms, we were able to correctly identify the OS of 14.0% of FTP packets. Similarly to SSH, we did not select features with GA due to an insufficient amount of packets remaining.

4.2.10 Discussion

In this section, we presented the contribution level of popular protocols to classify the operating system (OS) of hosts from which the packets originated. We also presented how well certain machine learning algorithms performed for the operating system classification from TCP/IP protocol headers. By using genetic algorithm (GA) to select most distinguishing features, we demonstrated the amount of contribution that the selected features have in affecting the classification performance while reducing computation overhead in classifying OSes. Since classification was performed individually on the packets, results obtained are not bound to restrictions such as classification of certain packet types only (e.g. SYN packets in TCP). Overall, with IP, ICMP, TCP, UDP, HTTP, DNS, SSL, SSH, and FTP, protocol header information, on average, operating systems of 68.0%, 51.6%, 98.4%, 71.1%, 78.7%, 29.2%, 25.0%, 22.5%, and 14.0%, respectively, packets were correctly classified. Among analyzed protocols, TCP provided the best overall performance with all algorithms that completed analysis of TCP headers and we obtained close to perfect classification with all but one algorithm.

Chapter 5

Operating System Fingerprinting via Automated Network Traffic Analysis

In this section, we present improvements to our operating system (OS) classification approach. Our approach has several unique features.

First, we generate our own set of signatures to perform OS classification on newly seen packets and do not depend on other fingerprinting tools or their databases.

Second, unlike other approaches to passive OS fingerprinting that manually select "useful" features like TTL, window size, and de-fragmentation flag, we have no bias. The genetic algorithm (GA) generates the set of most useful features based on the data collected and the specific bias of the machine learning approach used.

Third, existing tools such as p0f, ettercap and siphon seek a perfect exemplar data match when classifying a packet [11], [23]. However, a non-perfect match can occur in some cases. Rather than discarding such packets, the machine learning algorithms that we utilize learn to classify them.

Fourth, the advantage of using machine learning techniques rather than depending on expert signatures is that we can dynamically adapt our generated models to different OSes and other dynamic environmental changes. By providing training data from the newly introduced OSes, we can automatically re-generate our classifier system's set of signatures to include these new OSes. When newly created proxy firewalls tamper with packet header information [11], our approach can learn to adapt.

Fifth, many passive fingerprinting tools such as p0f, ettercap, and siphon depend on specific packet types such as SYN, ACK, or SYN-ACK [11]. Unlike systems with such strong constraints, we can perform classification on any packet. We also do not depend on specific network protocol packets for performing OS classification. Any TCP/IP header protocols can be used for performing OS classification using our approach. In [14], we presented the protocols which contribute the most for performing OS classification.

Finally, tools such as SinFP, Nmap, and p0f usually have a preset number of features that they use to generate their signatures for OSes. A high preset number has the potential to increase the number of redundant features that do not necessarily contribute to the classification. The second-generation Nmap's database contains 4,766 signatures where 17% of these belong to different varieties of Linux and introduce much redundancy [22]. A low preset number may lose critical features and degrade accuracy. With our approach, the GA evolves the number of features based on a fitness function that moves towards high classification accuracy and small feature subset size. The provided data and the machine learning algorithm determine feature selection and signature size reducing administrator burden.

5.1 Methodology

The presented approach is an entirely automated machine learning-dependant approach for classifying OSes using TCP/IP packets. We use genetic algorithm (GA) feature subset selection to determine relevant packet header features for learning to classify operating systems (OS). To the best of our knowledge, our approach is the first to use GA for feature subset selection in OS fingerprinting. In this section, we investigate classification accuracy using TCP, IP, and UDP packets and extract features from these packet headers. Broadly speaking, each member of the GA specifies the set of features to use for classification. The fitness of this individual is obtained by running a classifier with the individual-specified feature set on training data and using the accuracy obtained on a test set as the fitness. We describe the process in more detail below.

5.1.1 Data Collection

We set up and used a local network consisting of three PCs, three Macs, and three Raspberry Pis. To eliminate hardware bias, we collected packets from three instances of each OS on each hardware platform. Each instance, on average, contains around 79K packets from a single machine for every protocol and OS. We used Raspberry OS, Xubuntu 14.04, Windows 7, Windows 8, Mac OS X El Capitan, and Mac OS X Lion. As shown in Figure 5.1, we dedicated two instances for each OS for training and validation, and the third for testing.

We generated the data from these machines by visiting the top 10 websites mentioned on Alexa (`http://www.alexa.com/topsites`).

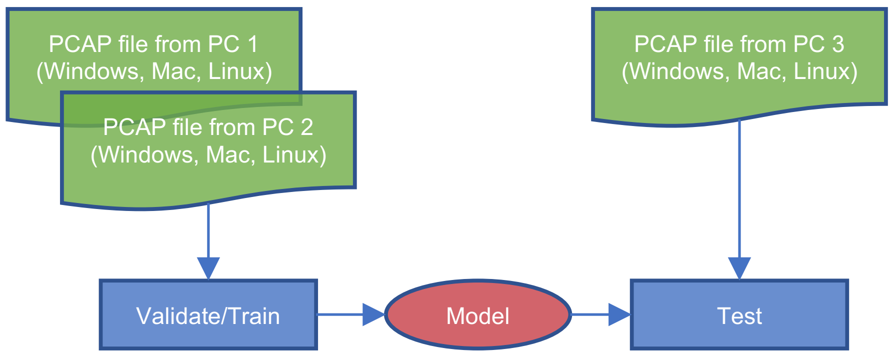


FIGURE 5.1: Instances

These websites were;

`http://www.google.com`, `http://www.youtube.com`, `http://www.facebook.com`, `http://www.baidu.com`, `http://www.yahoo.com`, `http://www.amazon.com`, `http://www.wikipedia.org`, `http://www.twitter.com`, `http://www.google.co.in` and `http://www.qq.com`.

We also collected 30-second Youtube video streaming packets from every OS on every device. To generate FTP and SSH protocol packets, we connected to *GoDaddy* and *CSE UNR*'s servers (our department) and uploaded files to these servers. To generate ICMP packets, we used the *traceroute* application to connect to these top 10 websites mentioned above. We also performed ping tests on the top 10 websites. However, since Amazon servers did not allow ping tests, we used the 11th website on Alexa's website list, *www.live.com*. We also performed 30-second Skype calls and sent e-mail, and received e-mail from these OSes.

Our setup enabled us to collect IP, ICMP, UDP, DNS, HTTP, IGMP, TCP, FTP, SSH, and SSL protocol header packets. Our prior work has shown that IP, TCP, and UDP are among the best protocols for OS fingerprinting [14]. In this study, we classify OSes using these protocols' header fields. The data collected is stored in two *pcap* (a specific format for storing network data) files for each OS for training and one *pcap* file for testing. We extract all possible features for every packet in the *pcap* files. These features are saved to files and converted to arff format to make them compatible with the WEKA tool [104]. We then run a GA for selecting relevant features.

5.1.2 Feature Selection

We use two instances of each OS for selecting features using GA. For maximum classification accuracy, the fitness function used by our GA implementation is $Fitness = \sum_{i=1}^{n} Accuracy_i$ where n is the number of instances of OSes, $Accuracy$ is the classification accuracy with the provided machine learning algorithm.

For every chromosome that the GA wants to evaluate, an arff file for two out of the three OS instances is created containing the GA specified features. Let us call the first instance I_1 and the second I_2. Weka builds a classifier by training on I_1 and testing on I_2. We then train on I_2 and test on I_1. Fitness is the sum of testing the accuracy of the two combinations. The results presented in this section are based on a single GA evolutionary process. However, we observed the consistency in our results when we ran it multiple times.

The GA converges when we have five consecutive generations with no improvement in classification accuracy. Although we investigated other termination criteria, our strategy works well in practice.

We use a population size of 50 for the GA and simple binary encoding for the chromosome. Each bit represents a feature; 1 means that the feature is selected and 0 that it is not. The GA we used implements Elitism selection. Initially, the

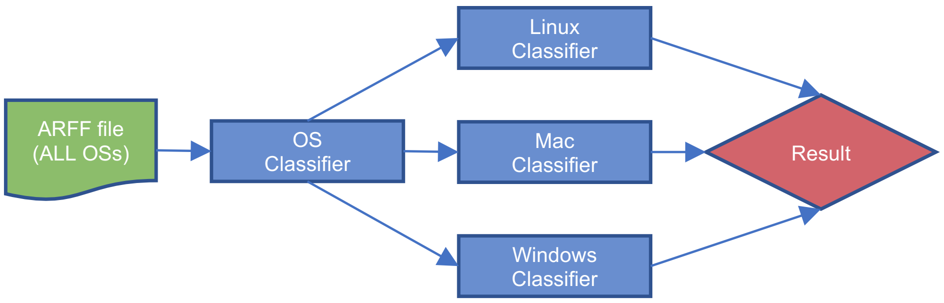


FIGURE 5.2: Classifiers

best individual is saved for the next population. For each of the remaining individuals in the population, two sets of 5 randomly selected individuals are chosen, and the best ones among the two sets are selected for performing crossover. Uniform crossover is used where bits are randomly exchanged between two individuals. The resulting individual is saved for the next population. Then each of the newly generated individuals is looped over and mutated. For mutation, each bit other than the best one is inverted with a probability. The mutation rate is 0.015, and the crossover rate is 0.5.

5.1.3 OS Classification

We use the machine learning algorithms in the WEKA tool [104] for OS classification, and in this section, we provide the classification accuracy for J48, RandomForest, OneR, and ZeroR algorithms. We generated four separate classifiers in a two-level hierarchy, as shown in Figure 5.2. The first classifier learns to classify the OS genre: Linux, Windows, or Mac OS. Once we know a packet's OS genre, the packet is passed on to one of three next level classifiers. The three two-level classifiers, one for each OS genre, learn to detect the OS version assuming that the packet belongs to a specific OS genre. If the classifier in the first layer classifies a packet as Mac OS, then it is sent to the Mac OS classifier in layer two to find out if it is Mac OS X El Capitan or Mac OS X Lion. This architecture improves classification accuracy since different protocols, and different machine learning algorithms perform better than others for possible packet origin scenarios.

5.2 Experimental Results

As mentioned in Sections 5.1.1 and 5.1.2, we dedicated three *pcap* files to each operating system (OS) for training, validating and testing our approach's accuracy. We used two of the *pcap* files to validate and train the machine learning models. Validation is the process of determining relevant features using genetic algorithm (GA), where we used one of two *pcap* files to train and the other to test the classification accuracy with the GA selected subset of features. After determining relevant features, we merged both of the *pcap* files used for validation into a single *pcap* file. We then used the merged file to train the machine learning model with the selected subset of features and then tested with the remaining third *pcap* file, which is only used to test the overall accuracy. Since we have three *pcap* files, there are three possible different combinations of selecting among these *pcap* files (e.g., *pcap* 1 & 2 for training, *pcap* 3 for testing; *pcap* 1 & 3 for training, *pcap* 2 for testing; and *pcap* 2 & 3 for training, *pcap* 1 for testing). We applied our approach for all three possible combinations and recorded the average accuracy. The results presented in this section are averages of all three combinations.

In this section, we present the results for the optimum accuracy for each of the classifiers shown in Figure 5.2. The optimum case is the highest classification accuracy of the classifiers with the data at hand. Since there are two levels of classifiers, the optimum case is when the second layer classifiers receive packets

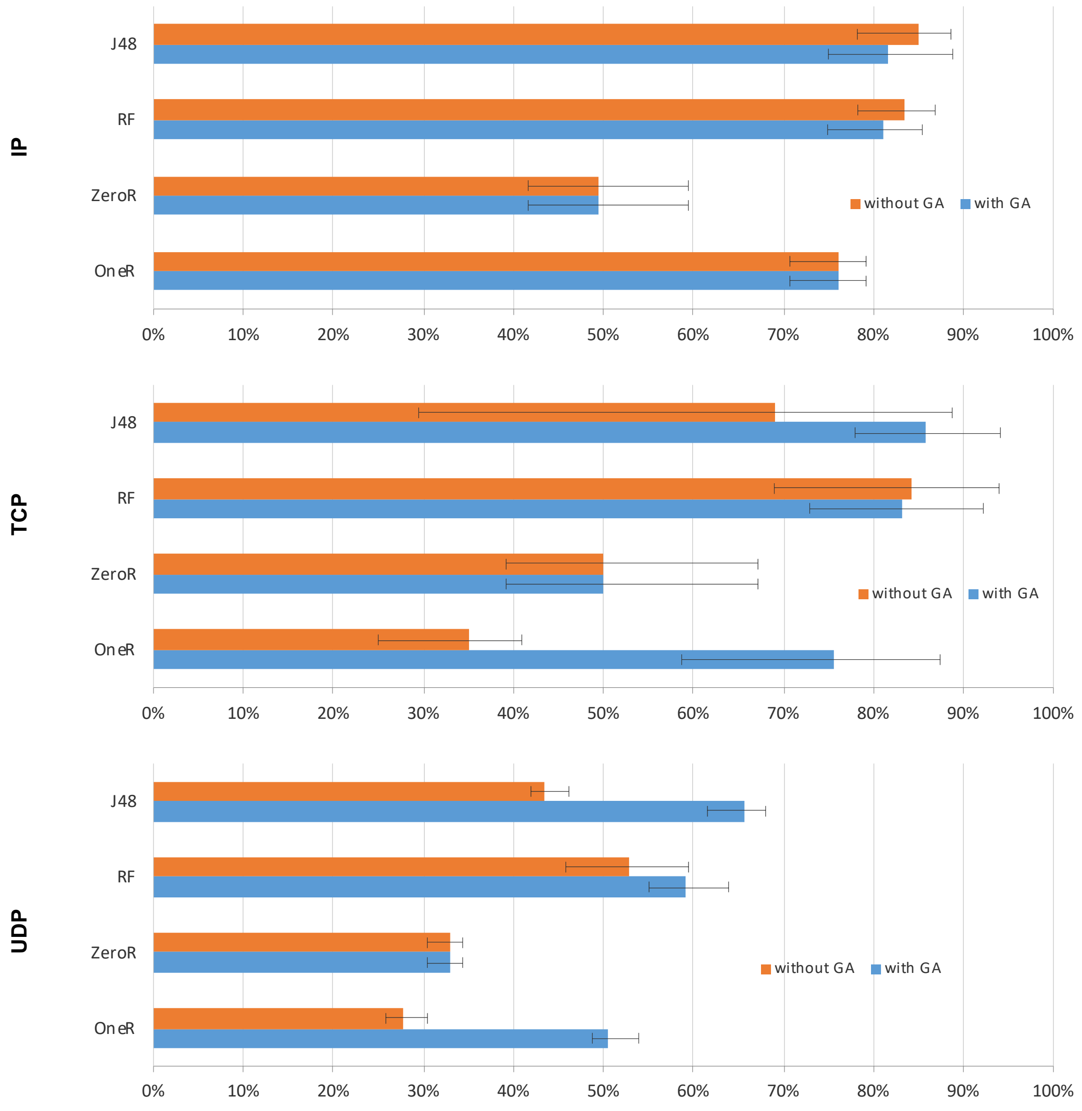


FIGURE 5.3: Classification accuracy of IP, TCP and UDP protocols for OS genre

from the first layer that belong to the OSes that the classifiers in the second layer classify with 100% confidence.

5.2.1 OS Genre Results

For the classifier in the first layer, we merged packets that belong to the same genre of OSes. After merging, the data contains packets that belong to three classes: Linux, Mac OS, and Windows. Figure 5.3 shows the classification accuracy of the genre classifier for IP, TCP, and UDP protocols, respectively. IP packets generally give good results. Both J48 and RandomForest generate classification accuracy greater than 80%. However, we observed the highest accuracy when TCP packets were used. J48, along with GA feature selection, was able to generate the highest accuracy at the rate of 86%. Even though it was not as high as J48, RandomForest was able to generate good accuracy at 84%. UDP, however, was not as useful as the other two protocols. The highest accuracy was at 66% with J48 using GA feature subset selection. The GA was able to select features that significantly helped the classification accuracy in many cases. For IP, classification accuracy with GA selected features was either equal to or very close to the classification accuracy using all features. For TCP, when the J48 algorithm was used, the accuracy without GA feature selection was at 69% and with GA was at 86%. When the OneR algorithm was used, the accuracy increased to 76% from 35% using the GA feature selection. For UDP, GA selected features helped improve the accuracy of J48, RandomForest, and OneR compared to accuracy using all features.

5.2.2 Linux OS Results

Figure 5.4 shows the classification accuracy for Linux. For IP, features selected by the GA appear to generate better results than the full set. For three of the algorithms used, the GA selected feature set provides a similar classification accuracy of 76%. We observed the highest accuracy with the OneR algorithm at 76.3%. For TCP, except for the Random Forest algorithm, the rest were able to perform better when GA is used. However, the OneR algorithm results far surpass the others at a rate of 95.3%, which is also the highest classification accuracy

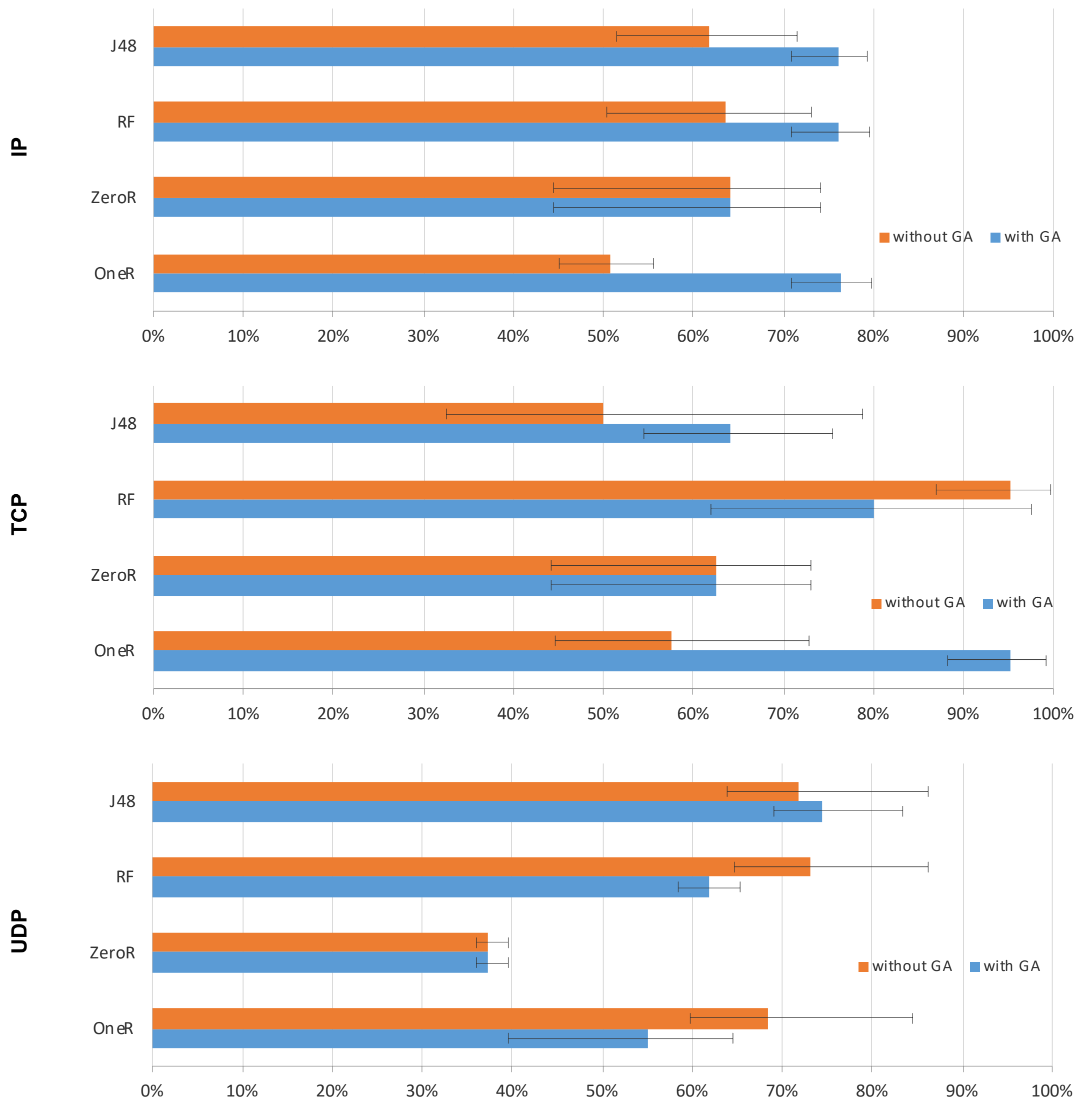


FIGURE 5.4: Classification accuracy of IP, TCP and UDP protocols of Linux versions

among all three protocols. For UDP, the GA in most cases was unable to select subsets of features that perform better than the full set except for with J48. We observed the highest accuracy again by J48 at 75%. Since the highest accuracy was achieved by the TCP protocol with OneR, as listed in Table 5.5, TCP along with OneR were set to be used to perform classification for Linux OSes. For IP packets, the GA was able to increase classification accuracy from 51% to 76% for J48, from 64% to 76%, for RandomForest, and from 51% to 76% for OneR. For

TCP, the GA was able to perform the best with the OneR where the accuracy was increased from 58% to 95%, which yielded the best accuracy for classifying Linux OSes. Since the UDP protocol does not significantly contribute to OS classification, GA was unable to find the optimal subsets for RandomForest and OneR algorithms. However, GA for the J48 algorithm was still able to increase the accuracy from 72% to 75%.

5.2.3 Mac OS Results

Figure 5.5 shows the classification accuracy for Mac OS. IP packets do not appear to contain features usable by our machine learners to achieve good accuracy. Deviation levels, as shown in the figures, are high for almost all the algorithms indicating that IP packets may not be able to classify Mac OS versions reliably. The deviation levels for TCP packet data are also high when all the features are used. However, with the help of GA feature selection, deviation levels decrease considerably, especially for J48. J48 is also the best performing algorithm for TCP protocol at a rate of 96%. OneR also gives as nice results as J48 at a rate of 95%. Even though UDP packet data gives better results than IP, accuracy is still low compared to TCP. The highest rate achieved was 55%, which is much worse than what TCP provides. Therefore, for Mac OS classification, we used TCP along with J48. It is also important to note that according to p0f v3 signatures [23], p0f can identify Mac OS versions when they are either 10.x or 10.9. Mac OS X Lion is version 10.7, and Mac OS X El Capitan is version 10.11. According to these signatures, p0f cannot distinguish between Mac OS X Lion and Mac OS X El Capitan where for the first signature both of these algorithms fall into the same category, and in the second, only Mac OS X El Capitan falls into the category. However, our approach can distinguish between these two versions at

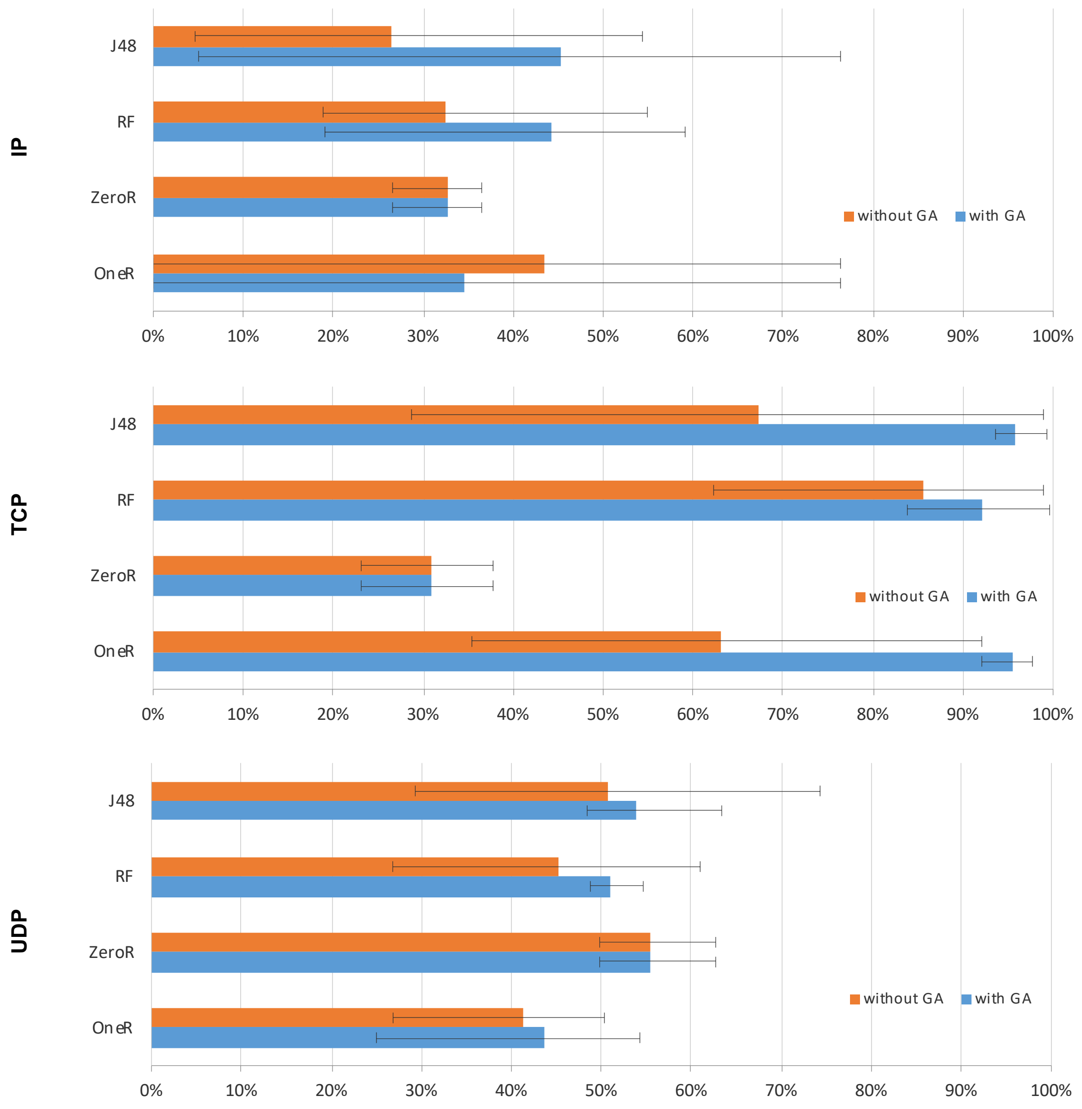


FIGURE 5.5: Classification accuracy of IP, TCP and UDP protocols for Mac OS versions

a rate of 96%. GA was able to increase the classification in all algorithms for TCP protocol packets. Specifically, GA feature selection was able to increase accuracy from 67% to 96% for J48, 86% to 92% for RandomForest and 63% to 95% for OneR algorithm. It should also be noted that when GA is used, the deviation levels decrease significantly, which yields more reliable results. GA was also able to increase the classification accuracy for UDP protocol packets.

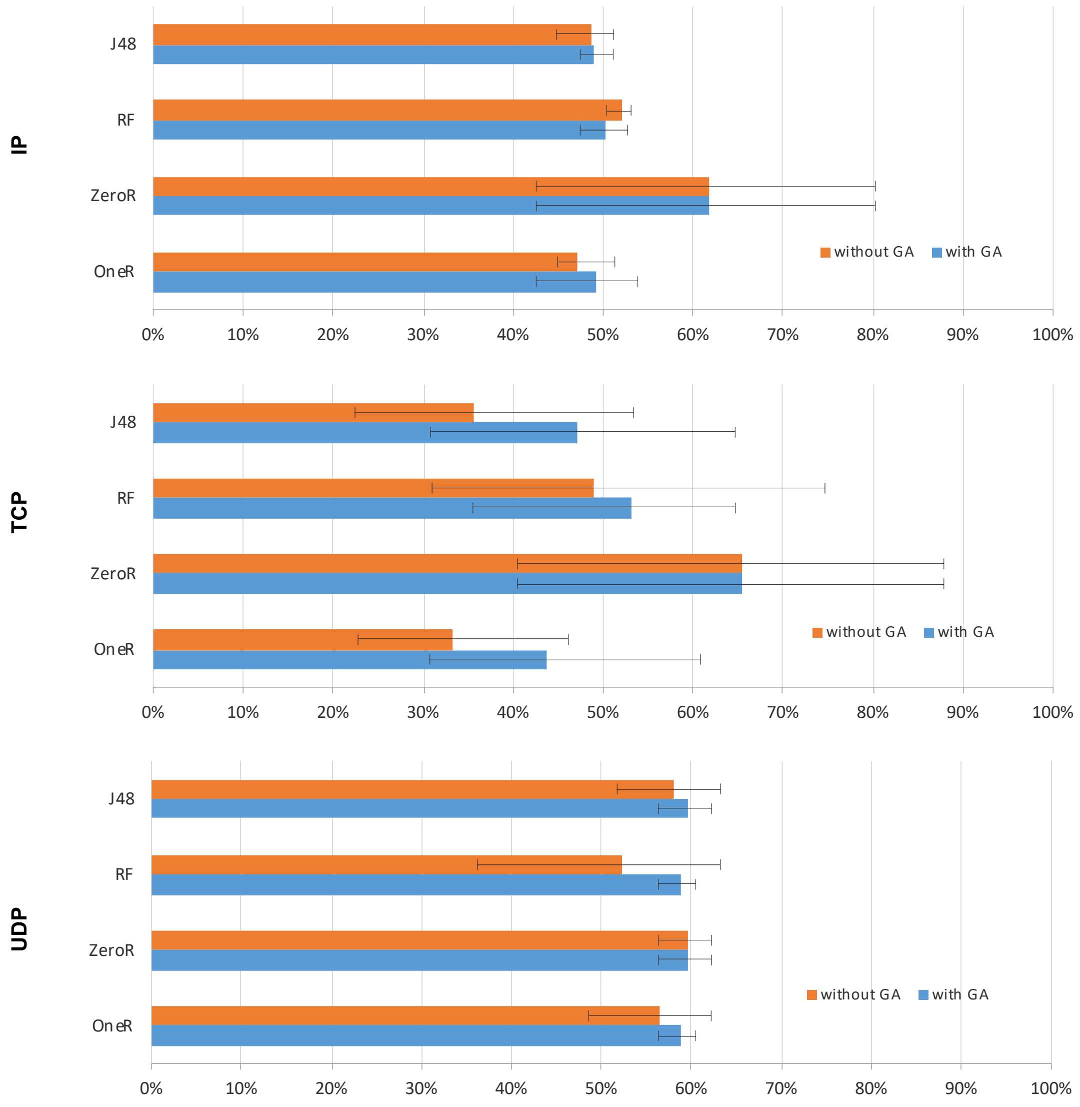


FIGURE 5.6: Classification accuracy of IP, TCP and UDP protocols for Windows versions

5.2.4 Windows OS Results

Figure 5.6 shows the classification accuracy for Windows OS. As the results indicate, the classification accuracy of distinguishing between Windows 7 and Windows 8 are almost 50%, which suggests that the behaviors of these two versions of OSes are very alike. p0f's signature database also has a single label for classifying these two versions of Windows OSes, which indicates that the network libraries of these Windows versions have no distinguishing feature. Due to this similarity, we merged Windows 7 and Windows 8 packets and considered them to be Windows. However, if different versions of Windows OSes with distinguishable behaviors were included, our approach could be used to specify them further.

In J48, the lowest entropy features contribute the most to the classification accuracy since they provide the most information gain. The lowest entropy feature selected by the J48 algorithm for classifying the OS genres was related to the TCP window size, which is consistent with the signatures of p0f as well. In p0f's signatures, the first term of the signatures is the window size. As it is known, the OneR algorithm tries to find a single rule to perform classification. The only rule selected by the OneR algorithm for classifying Linux OSes is also the window size. Similarly, the lowest entropy feature selected by the J48 algorithm for classifying Mac OSes is related to the TCP window size, which shows that our approach can detect features that are known to aid in classifying OSes. However, more specific features are selected further down the process in order to better learn the differences in OS behaviors.

5.2.5 Feature Subset Selection

Tables 5.1, 5.2, 5.3 and 5.4 present the number of features that GA selected for each protocol for OS genre, Linux, Mac OS and Windows respectively. Although we have not specified the weight for reducing the number of features in our fitness function, GA was still able to select smaller subsets for each scenario, which shows that there exist many features that either contribute little or nothing to OS classification.

TABLE 5.1: Number of selected features for OS identification

Classifier	Protocol	Algorithm	GA	ALL
OS	IP	J48	10	17
		RF	8	
		OneR	9	
	TCP	J48	28	61
		RF	29	
		OneR	27	
	UDP	J48	5	7
		RF	4	
		OneR	3	

TABLE 5.2: Number of selected features for Linux version identification

Classifier	Protocol	Algorithm	GA	ALL
Linux	IP	J48	6	16
		RF	8	
		OneR	7	
	TCP	J48	27	59
		RF	24	
		OneR	26	
	UDP	J48	3	7
		RF	3	
		OneR	3	

TABLE 5.3: Number of selected features for Mac OS version identification

Classifier	Protocol	Algorithm	GA	ALL
Mac OS	IP	J48	11	17
		RF	8	
		OneR	9	
	TCP	J48	25	59
		RF	33	
		OneR	26	
	UDP	J48	4	7
		RF	3	
		OneR	3	

TABLE 5.4: Number of selected features for Windows version identification

Classifier	Protocol	Algorithm	GA	ALL
Windows	IP	J48	7	17
		RF	8	
		OneR	5	
	TCP	J48	23	59
		RF	27	
		OneR	29	
	UDP	J48	1	7
		RF	1	
		OneR	2	

After comparing the classification accuracy of different protocols with different algorithms for each classifier, we selected the ones that yield the highest classification accuracy. We present the protocols and algorithms each classifier uses in Table 5.5.

We believe the reason behind certain algorithms performing better than the others is due to both the inbuilt biases within each machine learning algorithm and the selection of features. For example, for the Linux OS classification, the GA selected a single feature for OneR, the TCP window size, while the GA selected multiple features for J48 along with the TCP window size. Although features other than TCP window size might yield better results during training, it may not necessarily yield better results during testing.

TABLE 5.5: Classifier settings

Classifier	Protocol	Algorithm	# of features	Accuracy
OS	TCP	J48	28/61	85.9%
Linux	TCP	OneR	26/59	95.3%
Mac OS	TCP	J48	25/59	95.8%

5.2.6 Discussion

We developed a genetic algorithm (GA) feature selection mechanism for machine learning for Operating System (OS) fingerprinting. With this approach, we can perform single-packet OS classification with high classification accuracy. The GA is able to find smaller and thus more efficient sets of features to perform OS classification. Unlike expert signature generation for OS classification, our machine learning algorithms can automatically and dynamically generate classification signatures. The fact that we use GA feature selection for machine learning algorithms to generate classifiers, allows us to dynamically adapt our approach to different OSes without expert input and unlike many available approaches to OS fingerprinting, we do not depend on specific types of packets to perform classification. Our results show that GA feature subset selection was able to increase OS classification performances significantly for OS genre classification for Linux and Macs. We were able to increase the overall classification performance from 69% to 86% for the OS genre, 58% to 95% for the Linux OSes and 58% to 95% for the Mac OSes. These results were achieved with much fewer features than the initial set of features.

Chapter 6

Operating System Identification using Network Packet Headers

In this study, we present the Operating System IDentifier (OSID) tool, which employs machine learning and feature selection to build an automated OS classifier using TCP/IP headers on packets passively collected from host devices [105]. In particular, OSID analyzes TCP/IP protocol headers of IP and ICMP at the network layer, TCP and UDP at the transport layer, and HTTP, DNS, and SSL at the application layer. Note that OSID may work with any protocol suite as it is a self-learning approach that is agnostic to any particular protocol.

OSID employs a genetic algorithm (GA) to analyze the contribution of each packet header information and selects relevant header features for OS identification. GA considers each of the protocols independently or in an aggregated manner (e.g., IP+ICMP, IP+UDP+DNS, IP+TCP+SSL, and IP+TCP+HTTP). GA allows OSID to perform OS identification with fewer features at a higher classification accuracy. Feature selection can increase the identification accuracy by prioritizing the most contributing features and also by eliminating features which might introduce noise [56]. Also, having a fewer number of features to analyze reduces the computational overhead of OS identification. OSID employs several machine learning algorithms (e.g., DecisionTable, J48, Artificial Neural Networks, and PART) to perform OS identification using selected packet header features and determines the best algorithm for each classification phase [14]. OSID can detect OS signatures without any expert knowledge and automatically adapt to any new OS or protocol suite.

We analyzed the accuracy of single-packet OS identification to analyze the contributions of specific protocols. We also used several machine learning algorithms to analyze their classification accuracy. Based on our previous work [14], we observed that several algorithms have higher accuracy when classifying OSes. These algorithms are J48, OneR, DecisionTable, and PART.

OSID can detect OS-genres and Linux versions with over 99% accuracy using ICMP protocol packets. Similarly, Linux distros are classified by ICMP protocol with 98% accuracy. Mac OS versions are detected with over 98% accuracy using SSL packets. Note that neither of the existing similar tools can determine Windows versions, and OSID's results are not much better than random classification. Moreover, OSID is also able to reduce the number of features by eliminating those which do not contribute to the classification. Feature selection provides the ability to achieve faster classification and to build more general machine learning models. We also compared our results to p0f, one of the most prominent passive OS identification tools. Although p0f can classify the OS-genres of packets with as high accuracy as our results, we observed that it is not as consistent when classifying distros and versions of OSes.

Unlike many of the available tools, OSID does not focus on specific features such as; TTL, Window size, etc. as it detects useful features by analyzing the data using GA. Lack of expert input allows OSID to adapt itself to potential changes in OS implementations. It can also automatically learn new OSes by merely training with the data from OSes. Although OSID performs better with packets belonging to specific protocols, OSID is not restricted to be used with particular types of protocols such as the SYN packets, which is relied on by the popular passive analysis tool p0f. OSID is also flexible in terms of the number of features to be used for performing classification. With the help of GA, it detects whether it can perform classification with a high accuracy using a fewer number of features.

6.1 OSID: Operating System Identifier

This section presents Operating System Identifier (OSID), an entirely automated machine learning system for classifying OSes from network packets. OSID employs a genetic algorithm (GA) for determining relevant features of TCP/IP protocol headers. To the best of our knowledge, OSID is the first to use a GA for automated protocol feature selection in performing OS identification. Additionally, OSID uses machine learning algorithms to generate a set of signatures based on the selected features. For the fitness function implementation of GA, OSID employs the wrapper method of feature selection approach where the fitness function is determined by the accuracy that the feature combinations generated with the machine learning algorithm itself. After determining which features to use, machine learning algorithms generate a model for the classification of novel TCP/IP packets. OSID tests various machine learning algorithms to determine which algorithm generates the best results for OS identification at each level.

6.1.1 Data Initialization

OSID is initially provided with the protocol header format so that it can parse TCP/IP protocol header values from *pcap* network capture files. It eliminates features that only contain null values. Certain classifiers in WEKA tool [104] only work with numerical or ordinal features and not textual features. Therefore, we excluded character string features such as HTTP Accept request-header field and HTTP cookie to be able to use the WEKA tool for performing OS identification. Upon deciding which features to use, OSID extracts these features from every packet and converts to *arff* format to make them compatible with the WEKA tool. OSID then runs a GA for selecting relevant features as detailed below.

6.1.2 Feature Selection

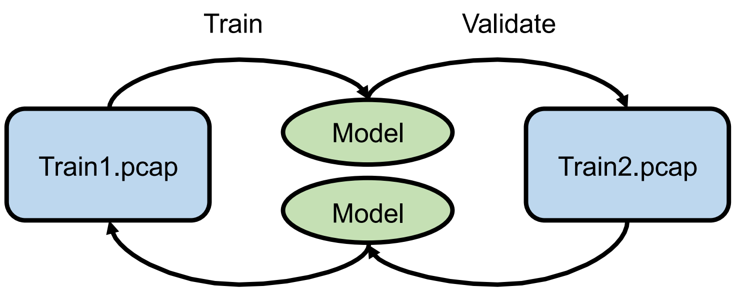
To reduce the computational complexity while keeping the classification accuracy as high as possible, OSID performs feature selection using a GA. Genetic algorithms (GA) search space of solutions while testing the performance of candidate solutions [103]. They are based on the biological mechanisms of natural selection and reproduction. GAs utilize a fitness function to evaluate the gain that a particular solution yields.

GAs start with a specific number of populations where each population contains a set of chromosomes (i.e., 0's and 1's). In the experiments, we set the population size to 50, the default value in GA applications. This number can be changed depending on the need for accuracy or speed. The GA initially generates a population of 50 chromosomes where each chromosome contains 0's and 1's, as many as the number of features in the dataset. If a particular bit in the chromosome contains 0, it indicates that the corresponding feature is to be ignored, and if it contains 1, then it is to be considered in the signature generation. At the initial phase, the chromosomes are assigned random values. In each iteration, GA determines the chromosome(s) that yield the highest accuracy and updates the population. This process is repeated at every iteration until a certain criterion is met. In OSID, the criteria are determined to be the generation of *n* consecutive iterations that yield the same highest accuracy. The best result is then selected as the optimal solution.

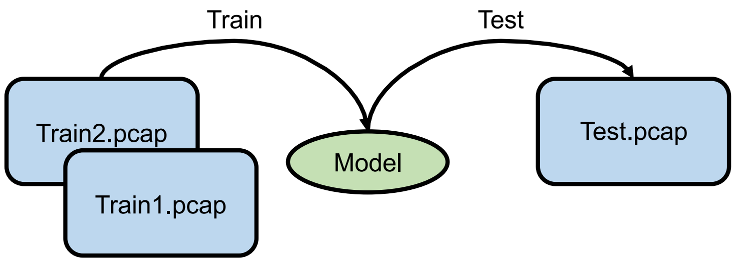
To calculate each chromosome's accuracy, OSID runs the machine learning algorithm with the selected features in the chromosome. The fitness function indicates how well the chromosome performs. The fitness function considers not only the machine learning classification accuracy with the selected features but also the number of features selected. OSID selects a minimal subset of features, while the identification accuracy is as high as possible. The fitness function is formulated as:

$$\begin{aligned} Fitness = \; & w_1 \times Accuracy + \\ & w_2 \times \left(1 - \frac{|SelectedFeatures| - 1}{|AllFeatures| - 1}\right) \end{aligned}$$

where *Accuracy* is the classification accuracy with the selected features and the provided machine learning algorithm, w_1 is the weight for the accuracy and w_2 is the weight for the feature reduction. Note that *Accuracy* is formulated as the proportion of true positives, and true negative in all evaluated cases. We have set the w_1 weight for accuracy to 0.95 and w_2 weight for feature reduction to 0.05. This balance gives a higher preference for accuracy than the minimization of the number of features.



(A) Validation



(B) Training

FIGURE 6.1: Integrated Validation and Training

OSID utilizes two data sets for training and validation of GA. Figure 6.1 shows the use of the two data files for training and validation of the GA-selected features. For every chromosome whose contribution GA wants to evaluate, arff files contain only the features specified in the currently processed chromosome, called Train1 and Train2. A classifier is built by training with the Train1 data file using the specified machine learning algorithm and is then validated on the Train2 data file. The same process is applied where the classifier is trained with the Train2 file and then validated on the Train1 file. The average of the accuracy of both identifications is taken as the *Accuracy* in the fitness function. The number of selected features is then calculated for the second term of the fitness function. A GA uses the fitness function for every chromosome to converge to a solution ensuring as high identification accuracy as possible while using as few features as possible.

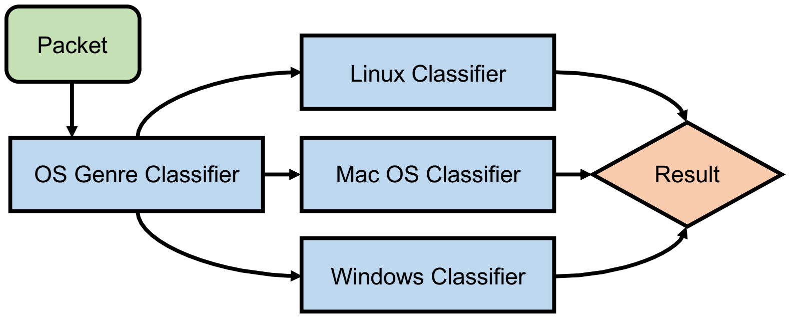


FIGURE 6.2: Classifiers

Since the fitness function might not always yield a 100% accuracy, a termination criterion needs to be set to decide when to end GA execution. If the last n solutions of the iterations are the same, then the system terminates the feature selection process with the current best solution as the final solution. If the n parameter is set to a small number, the system could terminate with local optima. However, if the parameter is set to a large number, the GA can loop redundantly without any improvement. In this study, we have set $n = 15$, which yielded good performance. After determining the relevant features, OSID performs OS identification using machine learning algorithms, as detailed below.

6.1.3 OS Identification

OSID provides flexibility to choose among several machine learning algorithms to perform OS identification. In this study, we used the machine learning algorithms implemented in the WEKA tool [104] for classification. In the experimental results, we demonstrate that DecisionTable, J48, PART, and Artificial Neural Network algorithms generate better results with certain classification levels.

We generated four classifiers in two levels to perform OS identification, as shown in Figure 6.2. The first layer contains a single classifier that tries to tell which OS family the input packets belong to (i.e., Linux, Windows or Mac OS). The second layer contains three classifiers, one for each OS family. After the first layer, the classifier determines the OS family of the packet, then the corresponding classifier in the second layer identifies the specific version of the OS. For instance, if the first classifier identifies a packet as coming from a Mac system, then it is sent to the Mac OS classifier in the second layer to find out if it is El Capitan or Lion. Having four separate classifiers in two layers allows OSID to select different algorithms and set of features for each classifier. This approach considerably improves the overall accuracy as compared to a single classifier, as analyzed in our preliminary study [27].

We performed the feature selection and identification processes for each of these classifiers to achieve the best set of features along with the best machine learning algorithm for each of these classifiers. We observed that different classifiers tend to perform better with each of the classifiers. Therefore, OSID generates a model for each of the classifiers using different machine learning algorithms, which gives OSID the flexibility to perform OS identification where the system adapts itself not only with selecting the most discriminative features in the OS versions but also with selecting different machine learning algorithms that obtain the best identification accuracy for each OS-genre.

6.2 Experimental Results

In this section, we present the accuracy of OSID over a sample data set with three OS families.

6.2.1 Data Collection

We obtained network packet captures from three different devices for each of the Mac OS X versions of El Capitan and Lion; Linux versions of Xubuntu 14.04 and Raspberry Pi 3b; and Windows versions of 7, 8 and 10. To avoid any network interface card (NIC) dependent behavior, each PC was equipped with a different NIC (except for Raspberry Pis hosting the Broadcom 5720).

To obtain uniform network traffic from each of the systems, we performed the same set of activity after the new initialization of each OS. We visited the top 10 websites listed on the Alexa `http://www.alexa.com/topsites`. These websites were;

`http://www.google.com`, `http://www.youtube.com`, `http://www.facebook.com`, `http://www.baidu.com`, `http://www.yahoo.com`, `http://www.amazon.com`, `http://www.wikipedia.org`, `http://www.twitter.com`, `http://www.qq.com` and `http://www.live.com`.

We also pinged each of these servers and performed a *traceroute* to them. We also initiated 30 seconds of Skype calls and streamed 30 seconds of YouTube video from each OS. Finally, we sent and received e-mails to web-based university e-mail servers from these OSes.

We utilized layer 3 and higher layer's protocol headers (i.e., network, transport, and application layers) in the OS identification. We ignored link-layer headers as they change at each hop and would not reflect the source system's network characteristics unless it was obtained in the first hop.

We dedicated three *pcap* files from each OS for training, validating, and testing the OSID system's accuracy. Each pair of datasets were used to validate and train the system. Validation is the process of determining relevant features using a genetic algorithm (GA) where we used one dataset to train and the other to test the identification accuracy of the GA-selected features. After determining important features, we merged both of the datasets used for validation into a single input file. The merged file is then used to train the system with the selected subset of features, and the system is tested with the remaining third dataset, which is only used to test the overall system accuracy. Since we have three datasets, there are three possible different combinations of selecting these datasets to run the OSID system. We performed identification with each of the three possible combinations and recorded their accuracy. The results presented in the text are typically the averages of these three batches.

Additionally, since a GA initially randomizes the chromosomes, it can find different results at every run. Therefore, we made sure to run every test ten times and recorded the results. The results presented in Sub-sections 6.2.2 and 6.2.3 are the averages of these 10 runs. However, in Sub-section 6.2.4, the best of the results within these ten runs are used to obtain the best accuracy. In addition to performing OS identification using GA-selected features, we also ran the OSID system using every feature that existed in every protocol to be able to compare results without any selection.

6.2.2 Individual Protocol Classification Results

First, we analyzed the contribution of every protocol to the identification of OSes. We analyzed IP and ICMP protocols at the network layer; TCP and UDP at the transport layer; and HTTP, DNS, and SSL at the application layer. We

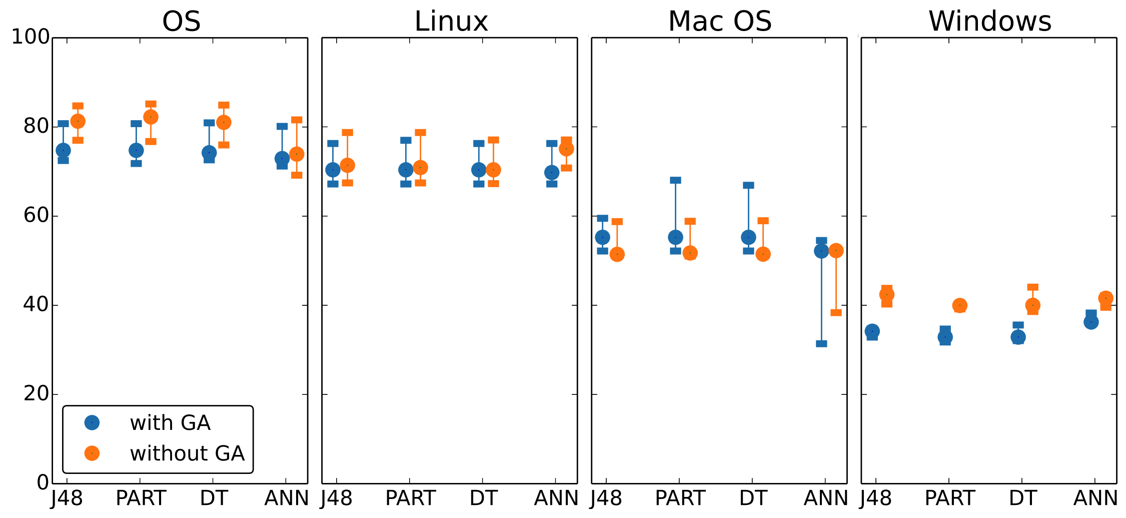


FIGURE 6.3: IP Performance

perform each analysis with and without GA using each of the machine learning algorithms. As GA initially randomizes the bits in the chromosomes, we performed ten experiments with a GA and provided the min-max and the average results. For each classifier, different protocols can yield better identification rates. Hence, OSID selects the best performing machine learning algorithm for every classifier, which assures the highest identification accuracy at every step of the OS identification. Note that as there are three OS families, the purely random classification would yield a 33% accuracy. Similarly, there are three Windows versions but two Mac OS and Linux versions.

For **IP protocol**, there were 17 features in total. On average, GA was able to reduce the number of features selected to an average of 1 to 2 features. Figure 6.3 presents OS identification accuracy using IP protocol alone. Note that each marker indicates the results for each of the datasets as the test data. We observed that for OS-genre detection, an average accuracy of 76% is obtained using GA selected features, and 81% accuracy is obtained using all IP protocol fields with the PART algorithm. We also observed that the next highest accuracy achieved using IP protocol headers is with the identification of Linux distros

TABLE 6.1: Selected IP Features

Features	OS	Linux	Mac OS	Windows
ip.checksum.status	✓			✓
ip.dsfield	✓	✓	✓	✓
ip.dsfield.dscp	✓	✓	✓	✓
ip.dsfield.ecn	✓		✓	✓
ip.flags	✓	✓	✓	✓
ip.flags.df	✓	✓	✓	✓
ip.flags.mf	✓			✓
ip.flags.rb	✓			✓
ip.frag_offset	✓			✓
ip.hdr_len	✓	✓	✓	✓
ip.len	✓	✓	✓	✓
ip.proto	✓	✓	✓	✓
ip.ttl	✓	✓	✓	✓
ip.version	✓			
Σ (features)	14	8	9	13

at an average of 72% with GA using the PART algorithm and 74% with all the fields using Artificial Neural Network (ANN). However, the IP protocol is not able to distinguish Mac OS versions or Windows versions with high accuracy. While certain protocols can distinguish Mac OS versions, none of the analyzed protocols could distinguish Windows versions 7, 8, and 10 at high rates due to the similarity in their behavior.

Additionally, Table 6.1 tabulates IP fields selected for classification. Header fields such as the Time-to-live (ip.ttl), Header length (ip.hdr_len), and Explicit Congestion Notification (ip.dsfield.ecn) are among the GA-selected header fields. The time-to-live field is used to determine the maximum number of hops a packet can be traversed on the Internet before being dropped. The initial value set by OSes for this parameter varies [17], [25], [106], [107], and many OS identification tools rely on it. Some OSes use different values for different protocols

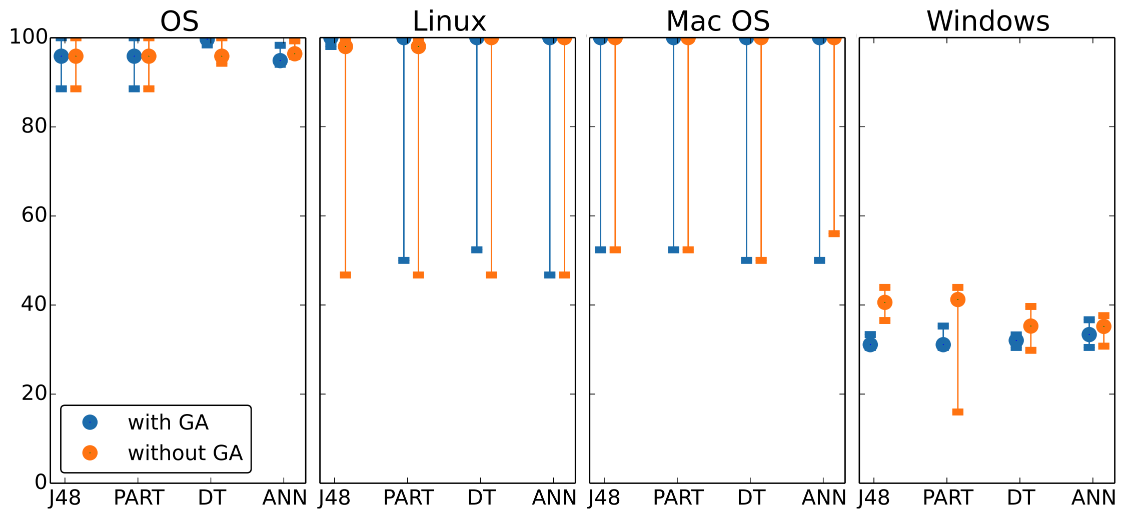


FIGURE 6.4: ICMP Performance

as well. For example, BSD distros use 64 in TCP and 255 in ICMP. We also observed that the header size has a significant contribution in distinguishing OSes, and OSID was able to capture these differences.

ICMP protocol had 13 features in the header fields. It is the best of the analyzed protocols in classifying OS-genres and Linux distros as seen in Figure 6.4. ICMP header fields help classify OS-genres at an average accuracy of 99% using GA-selected features and the DecisionTable algorithm. When all of the features are used, ANN was able to classify the packets at an accuracy of 97%. ICMP protocol header fields help detect particular Linux distros with high accuracy in all but one test dataset with the majority of experiments. OSID was able to detect Linux versions at an accuracy of 99% using GA selected features and the J48 algorithm. The average accuracy when all of the features are used was 82% using the DecisionTable or ANN algorithms, which shows that GA was able to find a smaller subset of features while increasing the identification accuracy considerably. OSID achieved an average accuracy of 84% when all of the features or a subset of the features were used for detecting the Mac OS versions. Windows OS versions were identified with an accuracy of 52%. While this is better than

TABLE 6.2: Selected ICMP Features

Features	OS	Linux	Mac OS	Windows
icmp.checksum	✓		✓	✓
icmp.checksum.status	✓			
icmp.code				✓
icmp.data_time_relative		✓		
icmp.ext.checksum	✓			✓
icmp.ext.res				✓
icmp.ident	✓	✓	✓	✓
icmp.length	✓			
Σ (features)	5	2	2	5

random classification (i.e., 33%), it is not of much value.

Since ICMP protocol is used to convey control information to source hosts, it provides OSes to have more control over its content, which enables us to capture more uniqueness across OSes. Table 6.2 enumerates selected ICMP features. One of the most distinguishing features is the Identifier (icmp.ident), which specifies whether big-endian (BE) or little-endian (LE) format is used. Linux OSes use a unique identifier for every ping process which makes these features a candidate for detecting OS uniqueness. The GA was able to include these features in the selected subset of features. Another important feature is the (icmp.data_time_relative), which is the RTT of the ECHO request. Although Mac and Linux OSes contain a value for this feature, the Windows OS does not, which helps OSID detect whether a packet is from a Windows OS or not. Also, for the Mac and Linux OSes, the range of the values contained in this feature helps distinguish them as well. When classifying Xubuntu and Raspberry OS, we also observed that Raspberry OS's RTT is much faster than Xubuntu, which helps distinguish these distros as well. However, since this feature is time-dependent, relying solely on this feature can introduce bias in the classification, especially when trying to distinguish Mac OS and Linux versions.

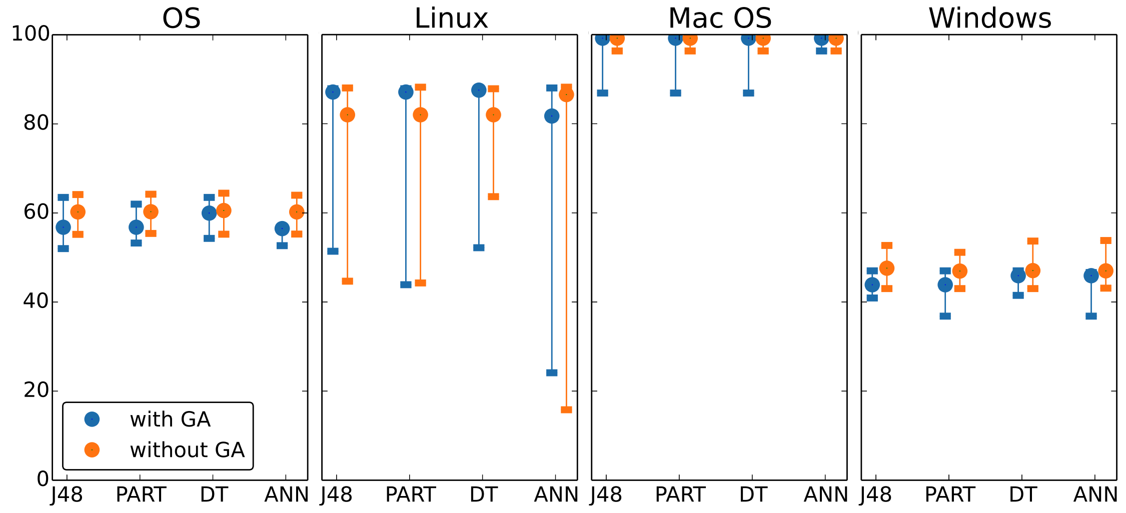


FIGURE 6.5: SSL Performance

DNS protocol is among those which do not contain much unique information in its header. Therefore DNS does not classify different OSes for accurate identification. Note that while Chang et al. [70] utilized DNS logs for OS identification, individual DNS packets do not have unique characteristics to reveal the OS. The average identification accuracy for detecting the OS family, Mac OS versions, and Windows versions is between 40%-55%. Only in Linux distro, the identification accuracy is the highest at 63%.

While **HTTP** is a more complex protocol with different options, it did not provide high classification results as we eliminated textual features. After filtering features, we had 11 features left for the classification. While individual datasets yielded around 90% accuracy of OS identification at best, other instances had a much lower accuracy of 30% and 50%. Similarly, the identification rate using the HTTP protocol headers was around 65% when classifying Mac OS versions with a smaller variance.

The most contribution that the **SSL protocol** makes is when trying to distinguish the Mac OS versions, as shown in Figure 6.5. OSID achieved an average accuracy of 98% with ANN. The GA was able to achieve as high identification rate as when all of the features are used by selecting 3 of the 40 features. The second-highest identification accuracy was achieved with the classification of the Linux distros with 76% accuracy using 6 of the features and at 78% using all of the features. We observed that one of the test datasets performs much worse than the other two, indicating it had different parameters with the selected set of features than the other two samples.

The SSL handshake is used to establish the secret keys between the client and server to communicate securely. During the handshake process, both parties exchange information such as the protocol version and crypto algorithms to be used as well as the digital certificates for authentication purposes. Hence, the handshake fields in the SSL header contribute the most for classifying OSes as different OSes have a differing set of crypto algorithms and defaults. As seen in Table 6.3, GA was able to detect such signatures and select handshake options for OS identification.

TCP protocol is one of the most complex protocols in the TCP/IP protocol suite. It has ambiguous fields and different default values selected by different OS versions. Hence, OS implementations have unique values in the TCP protocol headers. One of the protocols where the use of GA feature selection proves its contribution the most is the TCP protocol. TCP protocol when using GA-selected features is the most contributing for detecting the Mac OS and Linux versions. Using 23 of the 70 features in the TCP protocol, OSID was able to increase the Mac OS version identification rate from 48% to 98% as shown in Figure 6.6, which indicates that the TCP protocol contains features that cause noise in the data for OS identification. GA feature selection was able to ignore such features and significantly improve identification accuracy. Most of the OS fingerprinting tools rely on the TCP SYN packets for detecting OSes since TCP

TABLE 6.3: Selected SSL Features

Features	OS	Linux	Mac OS	Windows
ssl.alert_message	✓	✓	✓	✓
S.H.ext_server_name_len	✓	✓	✓	✓
ssl.change_cipher_spec		✓	✓	✓
S.H.ext_server_name_list_len		✓		✓
ssl.handshake (S.H)	✓	✓	✓	✓
S.H.ext_server_name_type	✓	✓		✓
S.H.ciphersuite	✓	✓		✓
S.H.ext_status_request_exts_len	✓	✓	✓	✓
S.H.ciphersuites	✓	✓		✓
S.H.ext_status_request_responder_ids_len		✓	✓	✓
S.H.cipher_suites_length	✓	✓	✓	✓
S.H.ext_status_request_type	✓		✓	✓
S.H.comp_method	✓	✓	✓	✓
S.H.extension.type		✓	✓	✓
S.H.comp_methods	✓	✓	✓	✓
S.H.length	✓	✓	✓	✓
S.H.comp_methods_length	✓	✓	✓	✓
S.H.sig_hash_alg	✓	✓	✓	✓
S.H.extension.len	✓	✓	✓	✓
S.H.sig_hash_alg_len	✓	✓	✓	✓
S.H.ext_alpn_len		✓	✓	✓
S.H.sig_hash_algs	✓	✓	✓	
S.H.ext_alpn_list	✓	✓	✓	✓
S.H.sig_hash_hash	✓	✓	✓	✓
S.H.ext_alpn_str_len	✓	✓	✓	✓
S.H.sig_hash_sig	✓	✓		✓
S.H.ext_ec_point_format	✓	✓	✓	✓
S.H.type		✓	✓	
S.H.ext_ec_point_formats_length	✓		✓	✓
S.H.version	✓	✓	✓	
S.H.ext_elliptic_curve	✓	✓	✓	✓
ssl.record	✓	✓	✓	✓
S.H.ext_elliptic_curves	✓	✓	✓	✓
ssl.record.content_type	✓		✓	✓
S.H.ext_elliptic_curves_length		✓	✓	✓
ssl.record.length	✓	✓	✓	
S.H.ext_length	✓	✓	✓	✓
ssl.record.version	✓	✓	✓	✓
S.H.ext_reneg_info_len		✓		✓
Σ (features)	31	36	33	35

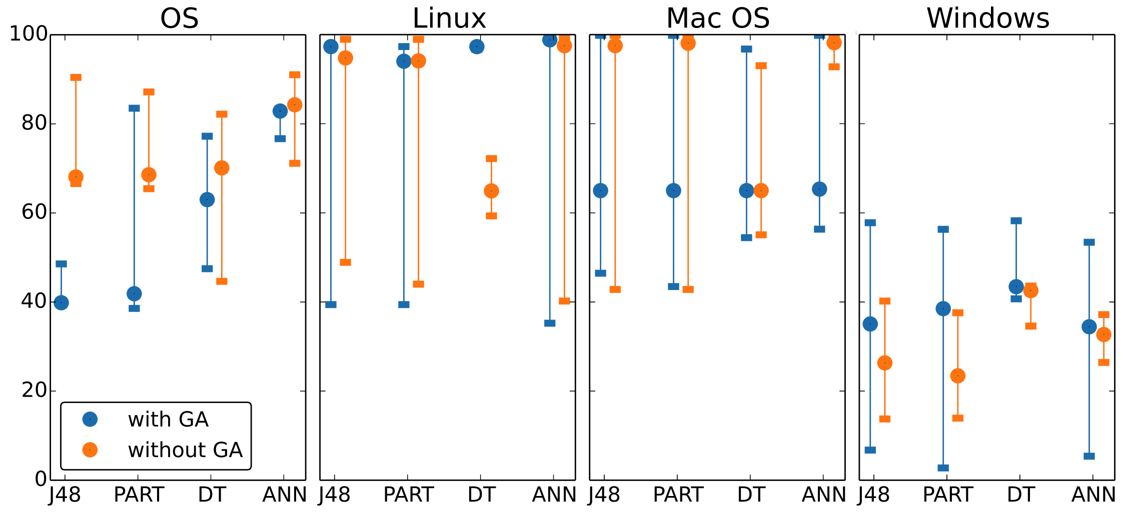


FIGURE 6.6: TCP Performance

SYN packet headers of different OSes contain distinguishing values. OSID was able to extract uniqueness within other TCP packets as well to classify OS-genre as well as Linux and Mac OS versions.

OSID was able to perform OS-genre identification with an average accuracy of 82% using ANN with all of the 28 features. For Linux distro detection, the accuracy of 97% was achieved using seven GA-selected features and the DecisionTable algorithm. Similarly, Mac OS versions were detected with an accuracy of 97% using the ANN on all of the features. The reason for the decrease in accuracy in classifying OS-genres is that the features in TCP protocols that many tools depend on detecting their uniqueness are in the header fields of TCP SYN packets. When any TCP packet is used, it becomes more challenging to capture this uniqueness.

As shown in Table 6.4, GA was able to select features that are used by the state-of-the-art tools for OS identification. p0f, a popular passive OS identification tool, considers the TCP options order, the maximum segment size, the window size, window size scale-factor, and TCP timestamps within TCP SYN packets. Without any expert knowledge, GA was able to select the header fields of TCP

TABLE 6.4: Selected TCP Features

Features	OS	Linux	Mac OS	Windows
tcp.analysis	✓	✓	✓	✓
tcp.flags.res	✓	✓	✓	✓
tcp.analysis.flags	✓	✓	✓	✓
tcp.flags.reset	✓	✓	✓	✓
tcp.analysis.initial_rtt	✓	✓	✓	✓
tcp.flags.syn		✓	✓	✓
tcp.analysis.window_update	✓	✓	✓	✓
tcp.flags.urg	✓	✓	✓	✓
tcp.checksum	✓	✓	✓	✓
tcp.hdr_len	✓	✓	✓	✓
tcp.checksum.status	✓	✓	✓	✓
tcp.len	✓	✓	✓	
tcp.dstport	✓	✓	✓	✓
tcp.option_kind	✓	✓	✓	✓
tcp.flags	✓	✓	✓	✓
tcp.option_len	✓		✓	✓
tcp.flags.ack	✓	✓	✓	✓
tcp.srcport	✓	✓	✓	✓
tcp.flags.cwr		✓	✓	
tcp.stream	✓	✓	✓	✓
tcp.flags.ecn	✓	✓	✓	✓
tcp.urgent_pointer	✓	✓	✓	✓
tcp.flags.fin	✓	✓	✓	✓
tcp.window_size	✓	✓	✓	✓
tcp.flags.ns	✓	✓	✓	
tcp.window_size_scalefactor	✓	✓	✓	✓
tcp.flags.push		✓	✓	✓
tcp.window_size_value	✓	✓	✓	✓
Σ (features)	25	27	28	25

SYN packets in a completely automated way. OSID selected unique features for other TCP packets as well. There are six flags in TCP header which are used to determine whether a packet contains data, initiation or termination for a connection. Since these flags can be combined in multiple ways, they help capture uniqueness in OSes. Another vital feature that OSID was able to select using GA was the Window size value. This value determines the additional bytes of data that the sender can accept. Many OS fingerprinting tools also heavily rely on

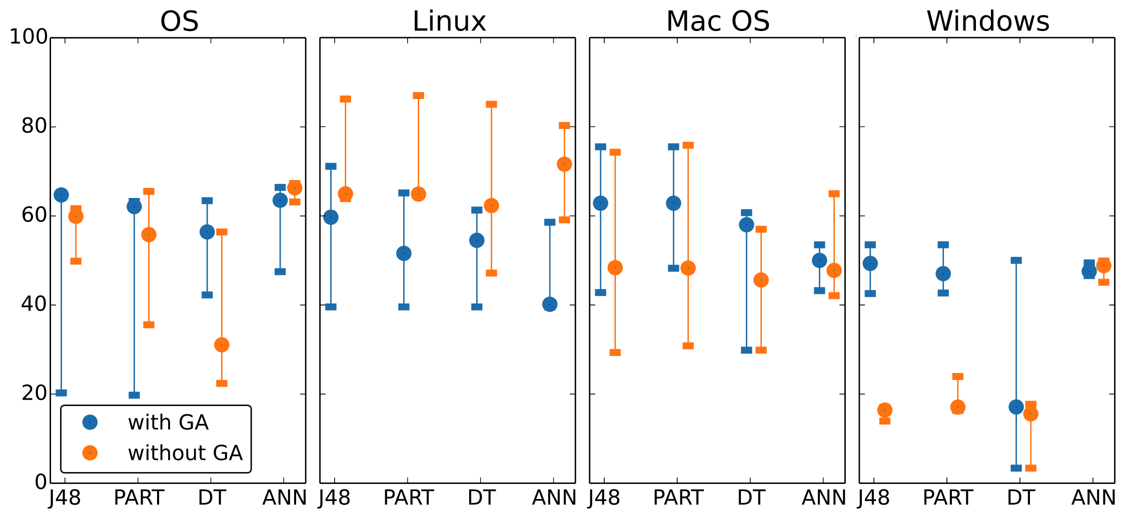


FIGURE 6.7: UDP Performance

this value since the values used for this field vary across OSes. Another relevant field in TCP headers that help distinguish OSes is the TCP options field. Every TCP option takes 8 bits, and multiple options can be used when generating packets. These options are appended to the original 20-byte long TCP header. As expected, the number of TCP options used along with the values set for each of them becomes crucial in capturing the uniqueness of OSes. Similar to the use of multiple columns in database tables to form a unique primary key, this shows that GA was able to detect uniqueness further using features from both of the protocols.

UDP is a connectionless protocol and therefore does not provide as complex header information as the TCP protocol. The highest identification rate with UDP was achieved when classifying Linux versions with an average accuracy of 72%.

Overall, we observed that IP, ICMP, SSL, and TCP protocols had the most discriminative header fields between different OSes and their versions. While HTTP is a protocol with many header options, filtering textual features diminished its utility for OS identification.

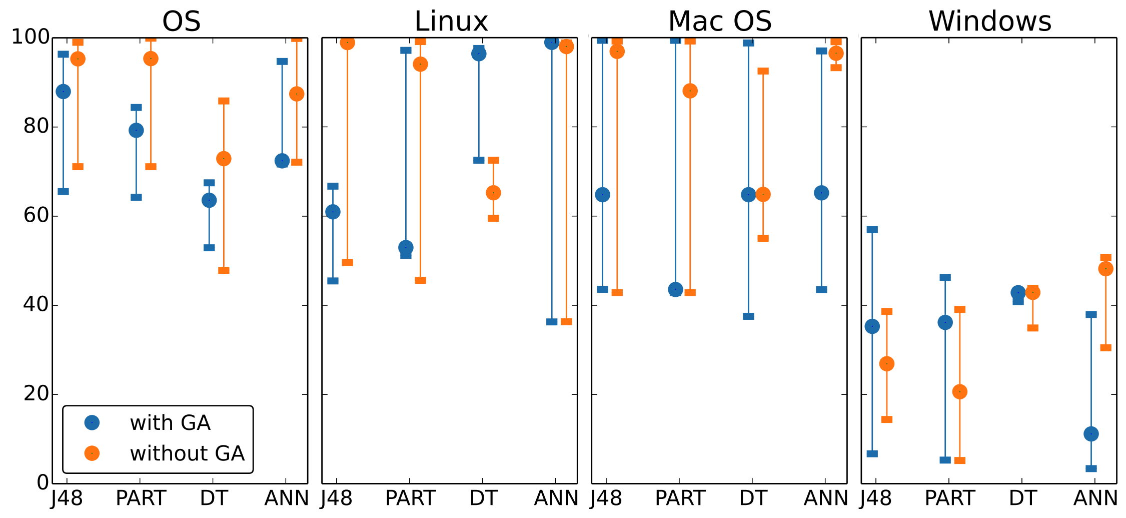


FIGURE 6.8: IP & TCP Performance

6.2.3 Individual Packet Classification Results

As presented in the previous section, different protocols yield different identification rates for each classifier. It is, however, possible for the combination of various protocols to yield higher accuracy in OS identification. When multiple protocols' features are merged, it is possible to catch a unique fingerprint for the OSes. In this section, we analyze the classification accuracy when layer 3 to layer 5 protocols of a packet are combined. In particular, we focus on analyzing the combination of protocols within a single packet. These protocols are: IP & TCP, IP & TCP & HTTP, IP & TCP & SSL, IP & UDP, and IP & UDP & DNS.

When combining IP & TCP protocols within **TCP packets**, OSID yields an average of 96% when all the features are used for detecting Mac OS versions and 97% for detecting Linux versions as shown in Figure 6.8. However, IP & TCP protocols do not yield as high identification rates for detecting OS families as it provides an average accuracy of 89% when all the features are used. Don't fragment is an IP header field that became useful with the TCP packet. Don't fragment bit is used to set whether the IP datagram is to be fragmented or not.

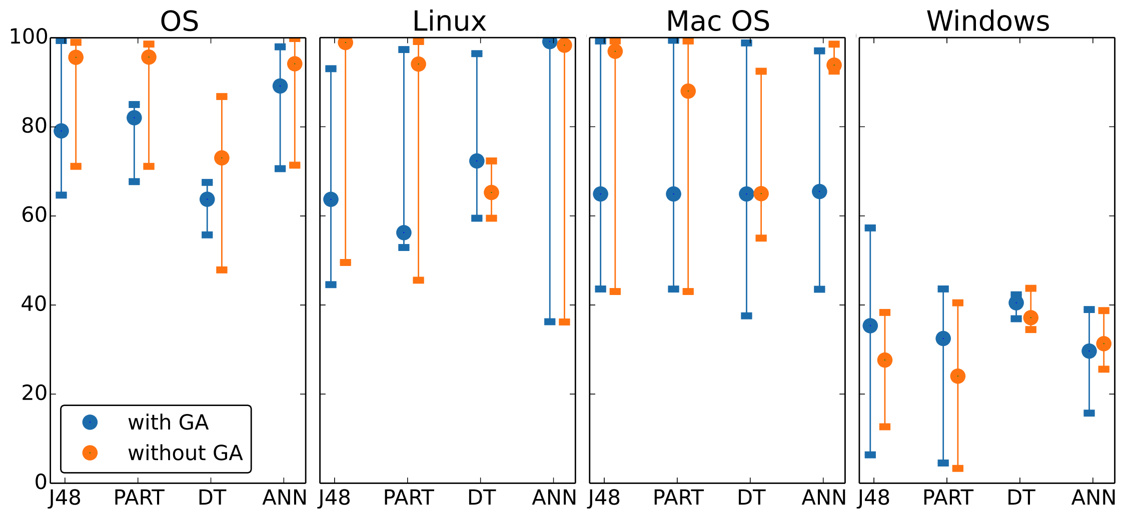


FIGURE 6.9: IP & TCP & HTTP Performance

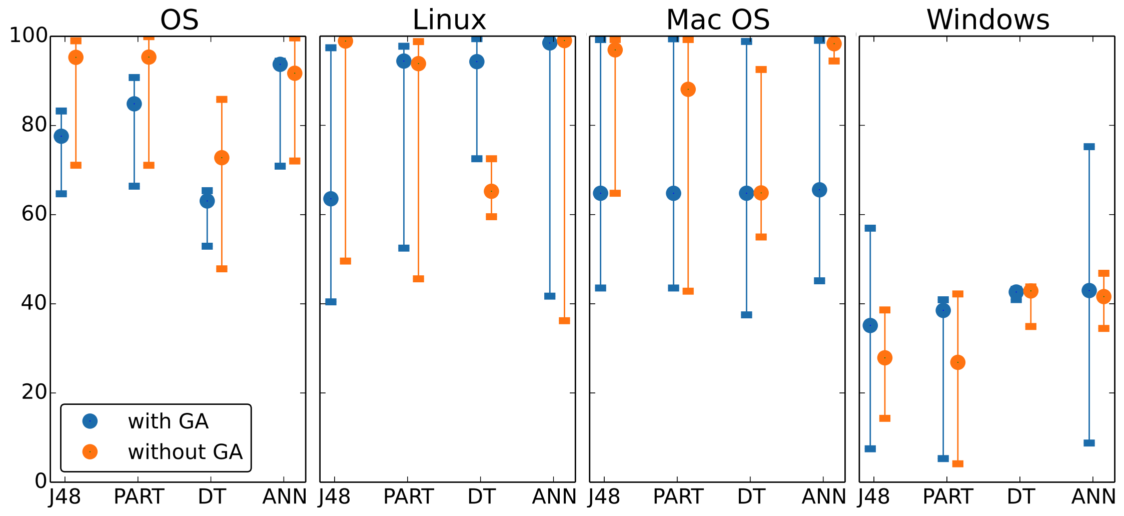


FIGURE 6.10: IP & TCP & SSL Performance

Many operating systems set this value by default mostly with the TCP SYN packets.

Figure 6.9 presents the accuracy when IP & TCP & HTTP protocol features are merged for an **HTTP packet**. Using all of the features of HTTP packets, OSID was able to achieve an average accuracy of 89% when classifying OS-genres. OSID determined Mac OS versions and Linux distros with an average accuracy of 97% and 86%, respectively.

Similarly, when IP & TCP & SSL protocols of an **SSL packet** are merged, OSID

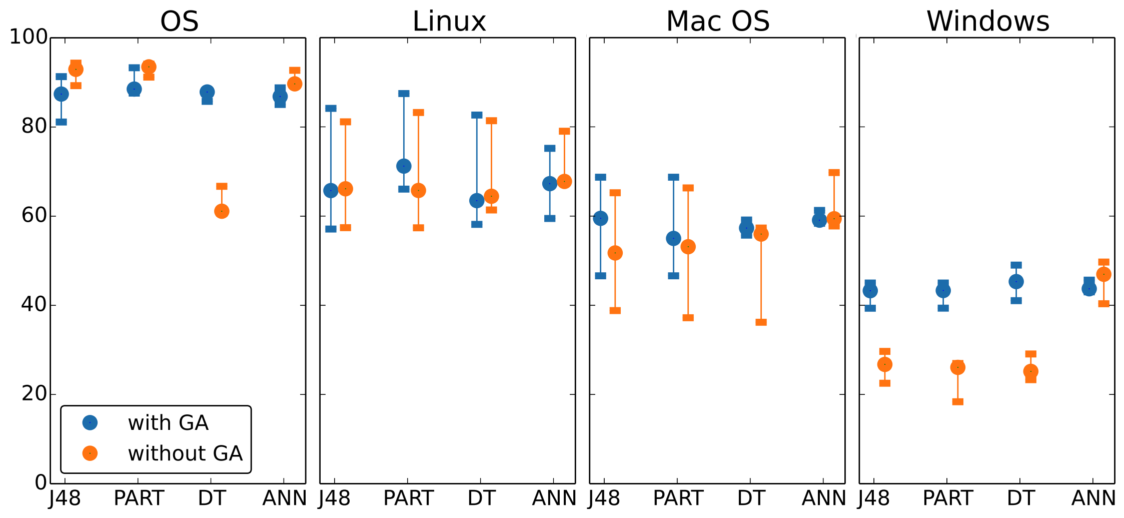


FIGURE 6.11: IP & UDP Performance

yields high results for Mac OS and Linux version detection. Figure 6.10 shows that OSID obtained 97% and 98% identification accuracy for Mac OS and Linux versions, respectively. In Mac OS version detection, a subset of features selected by GA yielded a lower identification rate than the original set of features. Hence, it is essential to fine-tune the parameters for GA and the weights in the fitness function for optimal classification accuracy.

As seen in Figures 6.11 and 6.12, the identification rates for IP & UDP of **UDP packets** and IP & UDP & DNS of **DNS packets** are very similar. Although they do not contribute much in detecting Linux and Mac OS versions, these protocols help detect OS-genres. OSID was able to classify OS-genres at 90% when GA selected features were used and 94% when all the features were used with the J48 algorithm.

We analyzed the results OSID obtained from every protocol and determined the most suitable protocols and algorithms for each classifier to assure the highest identification in each of them. The best performing protocols, along with the algorithms, are provided in Table 6.5. As seen, for the OS family classifier, we observed that the highest identification rate when GA is used was achieved

TABLE 6.5: Protocols Selected for Flow Analysis

Classifier	Protocol	Algorithm	Accuracy
OS	ICMP	DecisionTable	99.4%
	IP & UDP & DNS	J48	94.0%
	IP & UDP	PART	93.0%
Linux	ICMP	J48	99.3%
	IP & TCP & SSL	DecisionTable	97.7%
	TCP	DecisionTable	97.2%
	IP & TCP	DecisionTable	97.0%
Mac OS	SSL	ANN	98.3%
	IP & TCP & SSL	ANN	97.2%
	TCP	ANN	97.0%
	IP & TCP	ANN	96.3%
	IP & TCP & HTTP	ANN	95.0%
Windows	UDP	J48	48.5%
	DNS	DecisionTable	48.3%
	SSL	ANN	48.0%
	TCP	DecisionTable	47.4%

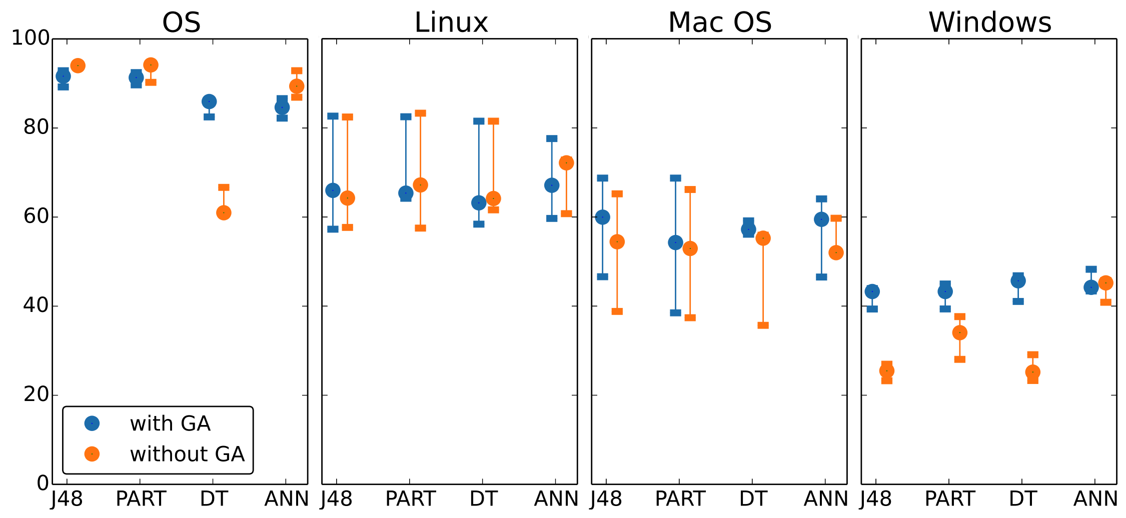


FIGURE 6.12: IP & UDP & DNS Performance

with the ICMP protocol and the DecisionTable algorithm at a rate of 99%, which means that by receiving any single ICMP packet from a system, OSID can detect whether it is running Mac OS, Linux, or Windows at a rate of 99%. The highest identification of Mac OS versions was achieved at a rate of 98% when SSL protocol and the ANN algorithms were used, which means that if a target system is using a Mac OS, we were able to tell whether it was using Mac OS X Lion or Mac OS X El Capitan at a rate of 98%. p0f is unable to yield such detailed information in terms of detecting the specific version of a Mac OS that is being used. We also observed that when trying to determine the Linux distros, OSID could achieve a rate of 99% when the ICMP protocol and J48 algorithm are used.

Similarly, this allows OSID to distinguish Raspberry and Xubuntu OSes with less than 1% error. The identification rate of detecting Windows versions, however, is low since Windows uses a very similar behavior across its OS versions. The highest rate of identification we achieved was at 49% when UDP protocol and J48 algorithm is used.

6.2.4 Network Flow Classification Results

In this section, we present the accuracy of detecting the OS-genre and the OS version from a sequence of packets originating from a single device. After analysis of individual packet classification results, OSID selects the best discriminating packet types and algorithm for each classification level. In particular, for the analyzed data, OSID selected the set of protocols listed in Table 6.5 to determine the OS and version of a given network flow. When considering a sequence of packets, OSID ignores packets with protocols that have low classification accuracy to obtain high classification accuracy.

For every packet belonging to the selected protocols, OSID determines the number of instances classified for each class and chooses the one with the most number of classification to be the class of the device. Table 6.6 shows the confusion matrix, i.e., the actual OS and the predicted OSes by OSID, for each of the data sets. In sampled data sets, OSID detected the OS-genre for Mac OS devices with 99%, Linux devices with 82% and Windows devices with 100% reliability. The reason for lower reliability with Linux devices is due to its behavior's similarity

TABLE 6.6: OSID Confusion Matrix

OS	Distro	Classified as	Data 1	Data 2	Data 3
Linux	Raspberry	Raspberry	100%	100%	96%
		Xubuntu	-	-	4%
	Xubuntu	Xubuntu	100%	100%	100%
Mac OS	Lion	Lion	100%	100%	100%
		El Capitan	-	-	-
	El Capitan	Lion	0.8%	1.1%	4.4%
		El Capitan	99.2%	98.9%	95.6%
Windows	Windows 10	Windows 10	2.8%	2.8%	4%
		Windows 7	15.5%	27.4%	3.9%
		Windows 8	81.7%	69.8%	92.1%
	Windows 7	Windows 10	4.8%	1.6%	3.9%
		Windows 7	0.7%	11.1%	-
		Windows 8	94.5%	87.3%	96.1%
	Windows 8	Windows 10	0.4%	18.3%	1.4%
		Windows 7	0.4%	-	-
		Windows 8	99.2%	81.7%	98.6%

to Mac OS. However, since OSID accepts the class with the majority of classification instances, it was able to detect the genres of OSes with 100% accuracy.

Additionally, OSID was able to detect the Mac OS versions such as the Mac OS X Lion with 96% and the Mac OS X El Capitan with 97% reliability. OSID was also able to detect the Linux versions such as the Raspberry OS with 98% and the Xubuntu with 98% reliability. Although OSID was able to detect the OS-genre for Windows OSes with 100% reliability, it was not able to distinguish the specific versions of Windows OSes with more than 49% reliability, which is due to the usage of the same network libraries across the Windows OSes and neither of existing similar tools can distinguish them from their packets.

6.2.5 TCP SYN results

In this section, we analyze the classification accuracy of OSID only using TCP SYN packets and p0f, state-of-art passive OS identification tool. p0f is one of

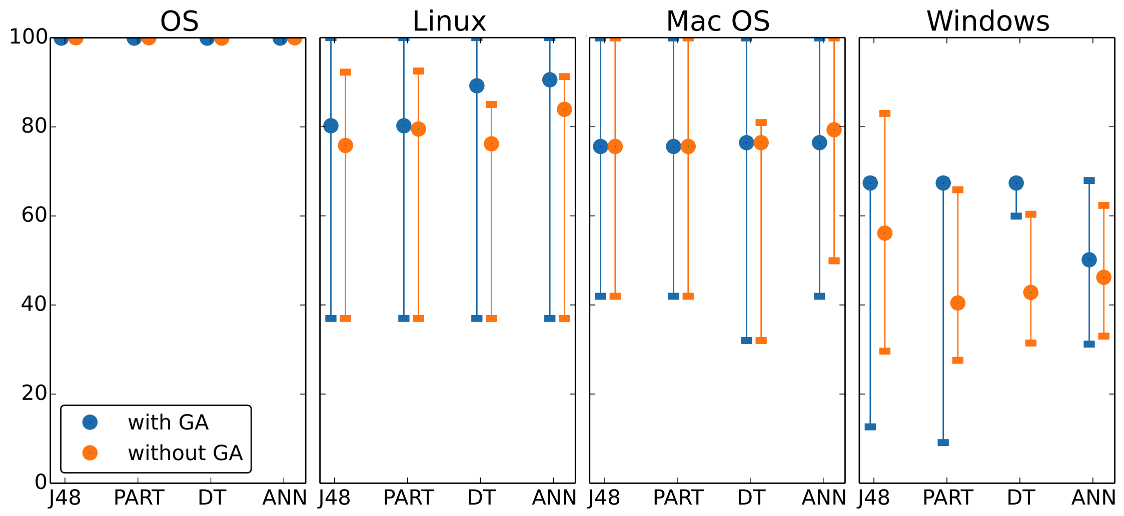


FIGURE 6.13: TCP SYN Packet Performance

the most prominent passive OS identification tools, and hence, we analyzed the results of the p0f tool using our sampled data.

Many OS fingerprinting tools [23] rely on the IP & TCP header fields of the TCP SYN packets to detect the OS of a system. TCP SYN packets are used to initiate a TCP session. In this section, we present the identification accuracy of OS detection using only the TCP SYN packets. As seen in Figure 6.13, OSID was able to detect OS-genre with a very high accuracy. Although all of the classifiers were able to perform identification at almost 100%, the highest accuracy was when GA selected features were used or when all the features were used with the ANN. However, as shown in Figure 6.13, we observed that TCP SYN packets are not very helpful in detecting the versions of OSes.

Table 6.7 shows the actual OS and the predicted OSes by p0f for each of the data sets. We also present the percentage of packets that p0f classified for each OS label. We observed that when determining the Linux distros or Mac OS versions, p0f detects the kernel version. p0f even associates Mac OS X Lion with the iPhone or iPad. As far as we can tell from these results, p0f can only detect whether a packet is from a Mac OS or not. We observe similar results with OSID

TABLE 6.7: p0f Confusion Matrix

OS	Distro	Classified as	Data 1	Data 2	Data 3
Linux	Raspberry	Linux 3.1-3.10	100%	100%	100%
	Xubuntu	Linux 3.1-3.10	98.4%	-	-
		Linux 3.x	1.6%	-	-
		Linux 2.2.x	-	1.2%	-
		Linux 3.1 and newer	-	98.8%	98.6%
		Linux 2.2.x-3.x	-	-	1.4%
Mac OS	Lion	Mac OS X 10.x	98.7%	100%	99.5%
		iPhone iOS or iPad	1.3%	-	-
		Mac OS X	-	-	1.3%
	El Capitan	Mac OS X	93%	91%	97.6%
		???	7%	9%	1.1%
		Mac OS X 10.x	-	-	1.3%
Windows	Windows 10	???	0.6%	0.4%	0.3%
		Windows NT 5.x	87.8%	92.2%	93.5%
		Windows NT	11.6%	7.4%	6.2%
	Windows 7	Windows 7 or 8	25.3%	100%	98.4%
		Windows XP	73%	-	-
		Windows NT	1.7%	-	1.6%
	Windows 8	Windows 7 or 8	89%	80.2%	88%
		Windows NT 5.x	-	18.6%	-
		Windows NT	11%	1.2%	12%

when only TCP SYN packets were used. For Windows detection, p0f confuses the OSes with a wide range of possible OSes such as Windows NT, XP, 7 or 8. However, we see that p0f uses the same label for Windows 7 and 8.

OSID was able to generate high identification accuracy in detecting OS-genres when using TCP SYN packets only. The main advantage of OSID is that it can automatically detect the uniqueness in the data without any expert input. However, OSID can perform further detection than p0f and identify which Mac OS version the packet belongs to with very high confidence. p0f uses TCP SYN packets only, but OSID does not have such restriction and can perform classification on packets belonging to multiple protocols with high identification rates.

6.3 Discussion

We tested different combinations of header features for analysis of the contribution of different protocols and the header fields in these protocols. However, this is a computationally costly task. For example, in IP & TCP & SSL protocols, there exist 88 features, and the unique number of combinations is 2^{88}. Since it is impractical to try this many numbers of runs, we needed a faster but accurate way of determining relevant combinations of features. A genetic algorithm (GA) became useful in searching for useful features without full exploration. We compared the results of both the GA-selected features and all the features available in every protocol to determine a subset of features that are the most contributing ones to the classification. Another advantage of the selection of a subset of features is the ability to perform classification with much less computational overhead. We also observed that the identification accuracy improves in many cases when the machine learning algorithms are fed with only the relevant features as opposed to dumping all of the information available due to possible noise in the data.

Many of the available tools focus on predetermined features such as; TTL, Window size, De-fragmentation flag, etc. OSID, however, is not fixated on the particular header fields of protocols or implementations by particular OSes for classification. With the help of a GA, OSID uses features that contribute most to the identification of OSes of packets, whether they are well-known for performing OS identification or not. As long as they help perform identification at high identification rates, they will get selected, which helps OSID extract as much relevant information as possible to increase the identification accuracy.

Another advantage of merely depending on machine learning techniques such as the OSID system is that it can dynamically adapt itself to different OSes. OSID can automatically re-generate a set of signatures after including training data for a newly introduced OS. Also, it is possible for proxy firewalls to temper with packet header information [11]. Therefore, having a dynamic system can be beneficial in order for the system to adapt itself to changes and still be able to perform classification with as high identification accuracy as possible.

Many of the passive identification tools such as p0f, ettercap, and siphon depend on specific packet types such as SYN, ACK, or SYN-ACK [11]. Unlike such systems, OSID tries to perform OS identification using any packet it is provided.

Tools such as SinFP, Nmap, p0f, etc. are usually limited in terms of the number of features they use to generate their signatures for OSes. If the number of features is preset to a high number, there might exist redundant, non-contributing features to the classification. If the number of features is preset to a small number, certain distinguishing information might be lost. OSID does not require the number of features to be specified. The system initially is provided with every possible feature for the selected protocol, and with the help of a GA, OSID automatically selects as small subset of features as possible while trying to keep the identification accuracy as high as possible. Therefore, both the number of features and the number of signatures are determined according to the data provided.

Although it is time-consuming to determine the relevant features with GA, it can be distributed among a cluster of computers and executed off-line. As training is a single time process, the generated signatures can be employed in a network with any subset of the trained OSes. Even though the system can adapt itself to the data at hand for performing OS identification, the data used for the training should be as representative of network layer behaviors of different OSes as possible in order for the OSID system to be usable across different networks.

Although we have not tested it, OSID allows performing OS identification both actively and passively. For example, after making an ICMP request to the targeted system, OSID can perform detection of the OS family and the Linux version by classifying the packet that it receives as a response. Since we observed that OSID achieves higher accuracy when detecting the Mac OS versions using the SSL protocol, after determining the OS family with the responded ICMP packet, further detection can be performed on the device's OS on an SSL packet by sniffing. Such a hybrid approach can also be implemented within OSID to make sure to benefit from the advantages of both approaches.

Finally, even though we have not implemented, as long as header fields are extracted similar to the IPv4 packets, the OSID system can perform identification of IPv6 packets as well.

Chapter 7

Automated IoT Device Identification from Network Traffic

Network administrators can automate the process of identifying attached devices using device fingerprinting. Device fingerprinting is the process of remotely detecting the kind of device from its network traffic [27]. There exist two general approaches to device fingerprinting, active and passive. Active fingerprinting probes the system and analyzes the response it receives. Passive fingerprinting sniffs packets generated by the system and analyzes the information extracted from the network traffic.

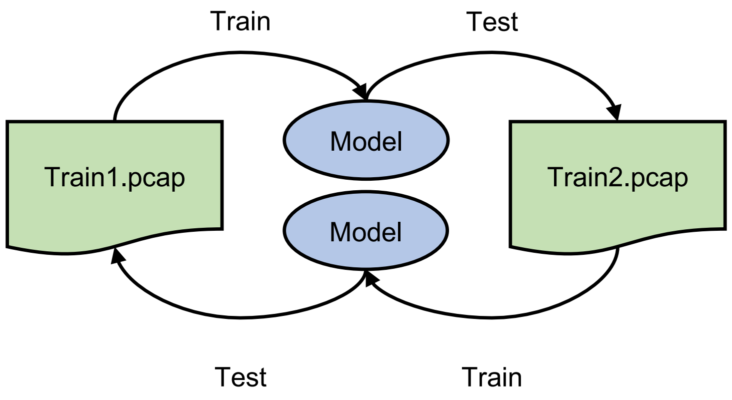
In this section, we introduce an automated IoT device fingerprinting system called SysID. SysID uses GA to determine the features that provide the most information-gain for extracting fingerprints of IoT devices. It uses machine learning algorithms to generate models that later are used to classify newly seen data. SysID is applicable to both active and passive fingerprinting. However, the data considered in this section was passively collected, and hence, the discussion is for passive fingerprinting.

In our analysis, depending on the packet content, we detected 212 features in the data used, which means that 2^{212} unique combinations could be considered for classification. With fewer features, a lower classifier complexity would be possible. Therefore we analyze both the number of features that is possible to reduce and also the possible increase in our classification accuracy using GA.

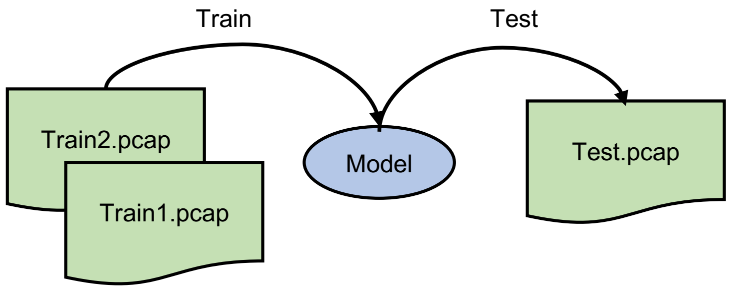
In an earlier study, we analyzed the contribution of various machine learning algorithms and identified J48, DecisionTable, OneR, and PART, among the best algorithms for Operating System classification using network traffic [27]. For every classifier that we build, we analyze the accuracy of all these algorithms and select the one with the best accuracy to optimize the overall classification of SysID. Rule-based algorithms such as PART and J48 also allow us to analyze the features and their contributions to fingerprint IoT devices.

7.1 Methodology

In this section, we present SysID, an entirely automated system to generate device fingerprints and classify IoT devices using a single TCP/IP packet. We use a GA for feature subset selection to determine unique packet header features for device fingerprinting among a large number of possible packet headers. To the best of our knowledge, our approach is the first to automate IoT device fingerprinting without expert input. SysID considers the network layer (i.e., IP and ICMP), the transport layer (i.e., TCP and UDP) and the application layer (e.g., DNS, HTTP, and SSL) protocols to determine the most significant protocol features among the protocols' header fields. SysID then runs a machine learning algorithm to determine the classification accuracy based on the set of features determined by the GA.



(A) Validation



(B) Training

FIGURE 7.1: Validation/Training

7.1.1 Feature Selection

We created three groups of *pcap* network captures for training, validating, and testing the accuracy of GA-selected features, as shown in Figure 7.1. In each group, we merged the *pcap* files into a single file and converted it to the *arff* file format that is compatible with the WEKA tool. After this process, we ended up with three *arff* files where we used two of them to train and validate the machine learning models, and the remaining one for testing. We named the *arff* files for training as *Train1.arff* and *Train2.arff* and the *arff* file for testing as *Test.arff*. Rather than only training with *Train1.arff* and validating it on *Train2.arff*, we wanted to make sure that the GA-selected features were sufficiently general for our classifiers when either of these two sets of packets was used for building a model, which helps us adapt our classifier to possible unexpected cases in the testing data. We did not perform 10-fold cross-validation since we observed the occurrence of over-fitting with the classifiers when we merged the data [27].

We used a population size of 30 chromosomes in our GA implementation. A chromosome is a series of 0′s and 1′s, at the same length as the number of features in the *arff* file. If the *arff* file, for example, contains 100 features except for the class, then the chromosome's length is 100. The values 0 and 1 for each bit of the chromosome determines whether to exclude or include a specific feature in machine learning. We set the GA to initially populate 30 chromosomes containing random 0′s and 1′s. For each of these chromosomes, the GA runs the fitness function to determine their strength with machine learning. Depending on the fitness values that the GA obtains for each chromosome, it tries to converge to the as optimal solution as possible by repeating the same process.

For our fitness function implementation, we considered two metrics to determine the most suitable set of header features. To perform classification with as high accuracy as possible with the smallest subset of features possible, we considered two metrics that represent these values in our fitness function. The fitness function returns a fitness value between 0 and 1, where 1 means the most contributing. As shown in the fitness function below, we used a weight of 0.9 for the accuracy and 0.1 for the reduction of the number of selected header fields.

$$\begin{aligned} Fitness = \; & 0.9 \times Accuracy + \\ & 0.1 \times \left(1 - \frac{|SelectedFeatures| - 1}{|AllFeatures| - 1}\right) \end{aligned}$$

If the number of features selected is equal to the number of features in the dataset, the component returns 0, which indicates that all header fields in the dataset are needed without any selection. However, if the number of features selected is 1, it indicates the contribution of this single header feature is sufficient for classification.

The accuracy in the fitness function is calculated by applying machine learning to the selected features in a chromosome. This term tells us how well the selected features in the chromosome classifies the testing dataset when the machine learning model is trained using these features. To calculate this component, we train a machine learning model on the *Train1.arff* dataset using only the features selected by the chromosome. Then we test the model on *Train2.arff* and record the classification accuracy.

Additionally, we swap train datasets and train the model using *Train2.arff* using only the features selected by the chromosome to be tested on the *Train1.arff*. After getting the accuracy for both cases, we calculate the average, which is used as the accuracy component in our fitness function.

The fitness function returns a value between 0 and 1 to tell GA how different a particular chromosome is from the others. GA tries to optimize this fitness value to converge to the best solution it can find, which then helps us find a set of features that are used for classification.

Since it is not guaranteed for GA to converge to the optimum set of features yielding the highest fitness value, it needs a termination point to prevent itself from going into an infinite loop of space exploration. In our case, we employed the decision of stopping the exploration after n consecutive occurrences of the same set of features being selected by the GA. If n is too low, the GA can quit before converging to a better possible set of features. If n is too high, depending on the implementation of the GA, it is possible for GA to either enter an infinite loop or to take a very long time to converge to a solution. We observed that $n = 5$

was sufficient for GA termination in our case. Additionally, the GA can converge to different sets of features in each run because it is randomly initialized. Hence, rather than relying on the outcome of a single run, we run the GA 10 times and select the features that produce the highest classification accuracy.

7.1.2 IoT Device Classification

For the classification of packets, we use machine learning algorithms in the WEKA tool [104]. WEKA provides a variety of machine learning libraries, and we used DecisionTable and J48 Decision Trees, OneR, and PART. We observed in our previous work [27] that in most cases, the rule-based algorithms outperform other approaches in classifying packets using the header fields. Since rules are generated based on the actual attribute values of the packets, algorithms may catch unique header features to classify devices. They also allow us to observe the exact values of attributes used for classification, which helps us analyze the behavior of a device by investigating the features used for identification and identifying the values of features that are distinctive between devices.

7.2 Experimental Results

In this section, we present the accuracy of System IDentifier (SysID) using 20 measurements from 23 IoT devices that include home sensors, coffee makers, power switches, and lights.

7.2.1 Data

In this section, we use a dataset collected for a similar study [77]. The dataset contains packets captured from numerous IoT devices. We used the data for 23 IoT devices that contained 20 measurements. The devices are: *Aria*, *D-LinkCam*, *D-LinkDayCam*, *D-LinkDoorSensor*, *D-LinkHomeHub*, *D-LinkSensor*, *D-LinkSiren*, *D-LinkSwitch*, *D-LinkWaterSensor*, *EdimaxPlug1101W*, *EdimaxPlug2101W*, *Ednet-Gateway*, *HomeMaticPlug*, *HueBridge*, *HueSwitch*, *iKettle2*, *Lightify*, *MAXGateway*, *SmarterCoffee*, *TP-LinkPlugHS100*, *TP-LinkPlugHS110*, *WeMoLink*, and *Withings*. For consistency reasons, we removed four devices that were included in [77] as they had less than 20 measurements.

We separated the *pcap* network captures into training (14 measurements for each device, i.e., 70% of datasets) and testing (remaining six measurements, i.e., 30%) data. In each group, we merged the *pcap* files and extracted all possible features for every one of their packets. Then we removed the IP address and the IP checksum (which correlates with IP address) to remove any bias that may occur in classifying the devices. We then ran GA to select relevant features with each machine learning algorithm (i.e., J48, DecisionTable, OneR, and PART) and recorded the algorithm with the highest accuracy.

7.2.2 One-level Classification

We first performed a classification of all devices and tried to detect individual device types using SysID. Figure 7.2 presents the classification accuracy of a single packet for each device. Because the PART algorithm performed classification

TABLE 7.1: Confusion Matrix

Actual \ Predicted	Withings	Lightify	D-LinkSiren	Aria	D-LinkWaterSensor	D-LinkSensor	TP-LinkPlugHS100	MAXGateway	EdimaxPlug2101W	TP-LinkPlugHS110	D-LinkDayCam	D-LinkCam	D-LinkHomeHub	HueSwitch	SmarterCoffee	HueBridge	D-LinkDoorSensor	WeMoLink	HomeMaticPlug	iKettle2	EdnetGateway	EdimaxPlug1101W	D-LinkSwitch
Withings	334	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Lightify	0	1931	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
D-LinkSiren	0	0	1397	0	1199	348	5	0	0	3	0	1	121	1	0	0	8	4	0	0	0	0	121
Aria	0	0	0	237	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D-LinkWaterSensor	0	0	727	0	1575	508	10	0	0	5	0	0	167	1	0	0	11	4	0	0	0	0	271
D-LinkSensor	0	0	1007	0	345	1589	7	0	0	7	0	1	106	1	0	1	7	3	0	0	0	0	336
TP-LinkPlugHS100	1	0	3	0	2	1	209	0	1	91	0	2	13	0	0	3	1	1	0	0	0	0	1
MAXGateway	0	7	0	0	0	0	0	308	0	0	0	0	0	16	0	0	0	0	0	0	0	0	0
EdimaxPlug2101W	0	0	1	0	0	17	3	2	125	2	2	0	0	0	0	1	0	0	0	0	0	128	0
TP-LinkPlugHS110	0	0	3	0	3	9	98	0	1	204	0	0	6	0	0	6	0	0	0	0	0	0	5
D-LinkDayCam	0	0	0	0	0	2	12	0	0	0	73	0	1	0	0	0	0	0	0	0	0	0	0
D-LinkCam	0	0	5	6	5	30	3	0	0	9	0	933	13	0	0	7	0	1	0	0	0	0	5
D-LinkHomeHub	0	0	175	0	106	204	7	0	0	16	0	10	3284	9	0	2	56	54	0	0	0	0	352
HueSwitch	0	0	1	0	1	0	0	0	0	0	0	0	1	10762	0	230	0	0	0	0	0	0	0
SmarterCoffee	0	1	0	0	0	0	0	0	0	0	0	0	0	0	13	0	0	0	0	0	0	0	0
HueBridge	0	0	0	0	0	6	0	1	0	13	0	0	2	520	0	7026	0	1	0	0	7	0	1
D-LinkDoorSensor	0	0	19	0	10	23	8	0	1	0	0	1	148	1	0	1	865	95	0	0	0	1	30
WeMoLink	0	10	0	0	6	2	1	0	2	0	0	0	2	0	0	8	5	2775	0	0	0	0	4
HomeMaticPlug	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	165	0	0	0	0
iKettle2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	13	0	0	0
EdnetGateway	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	5	0	0	0	0	294	0	0
EdimaxPlug1101W	0	0	0	0	0	15	5	0	53	1	0	0	0	2	0	2	1	0	0	0	0	191	0
D-LinkSwitch	0	0	310	0	146	393	7	0	0	7	0	4	649	1	0	0	32	13	0	0	0	0	1828

at higher accuracy than other algorithms, it was used in the classification of devices. Although numerous devices were classified correctly with high accuracy, some devices yielded a lower accuracy.

Table 7.1 presents the confusion matrix of SysID where rows show the actual device and columns show the predicted classification. The optimal case is when all packets are at the intersection of the same class, both for actual and predicted classes. For example, for the "Withings" packets, 334 packets were classified as "Withings", but only one packet was classified as "EdimaxPlug2101W", which is an error. However, when we inspect "HueSwitch" and "HueBridge" packets, we observe that most of the packets were classified as either of these two devices. For "HueSwitch", 10762 packets were correctly classified, but 230 of the packets were classified as "HueBridge". Similarly, for "HueBridge", 7026 were correctly classified as "HueBridge", but 520 of the packets were incorrectly classified as "HueSwitch", which indicates that the machine learning had difficulty

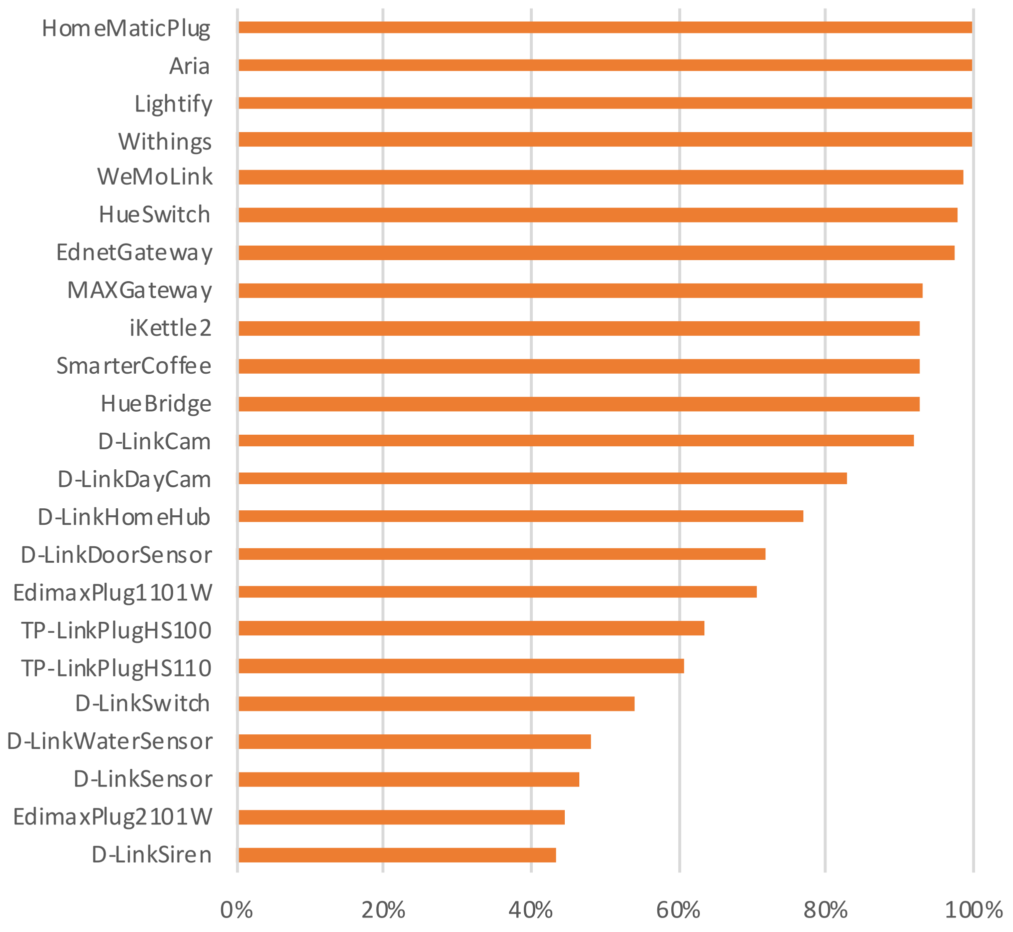


FIGURE 7.2: Overall classification accuracy

distinguishing between the behaviors of these two devices.

Overall, we observed that the machine learning had difficulty in distinguishing the behaviors of several groups, namely {EdimaxPlug1101W, EdimaxPlug2101W}, {HueBridge, HueSwitch}, {TP-LinkPlugHS100, TP-LinkPlugHS110} and {D-LinkCam, D-LinkDayCam, D-LinkDoorSensor, D-LinkHomeHub, D-LinkSensor, D-LinkSiren, D-LinkSwitch, D-LinkWaterSensor}. These groups of devices are from the same vendors and hence most likely utilize similar network libraries. While {iKettle2, SmarterCoffee} belonged to the same vendor, their network packet headers were distinguishable.

7.2.3 Two-level Classification

As IoT devices from the same vendor were confused by the machine learning classifiers, we decided to perform two-level classification where in the first layer we determine the device vendor, then in the second layer differentiate between devices from the same vendor. Hence, we generated six separate classifiers in a two-level hierarchy, as shown in Figure 7.3. The first classifier learns to classify either the device itself, if there is only one device from the vendor (i.e., Aria, EdnetGateway, HomeMaticPlug, Lightify, MAXGateway, WeMoLink, and Withings), or the device genre if there are multiple devices from the same vendor (i.e., D-Link, EdimaxPlug, Hue, TP-LinkPlug and Smarter). If the packet's class is determined to be one of the devices themselves, then that is the label that the classifier returns. If the packet is determined to belong to one of the genres, then the packet is forwarded to the corresponding classifier in the second layer for further processing to detect the actual device. This hierarchical structure allows us to detect the best classification algorithm in each classifier, which helps improve overall classification accuracy. Different protocols and different machine learning algorithms can perform better for different packet sources.

Figure 7.4 presents the classification accuracy for each class in our dataset when the PART algorithm is used. The figure also shows the individual device classification accuracy of IoT Sentinel [77], marked with blue. The lowest classification accuracy is at 95% for MAXGateway. We observe that SysID can obtain similar accuracy as IoT Sentinel, which uses 12 packet sequence in classification with expert input. As for the individual vendors with multiple devices, SysID can tell if a device is a Hue device at 99.8%, D-Link at 99.7%, EdimaxPlug at 98.9%, TP-LinkPlug at 96.2% and Smarter at 96.4%.

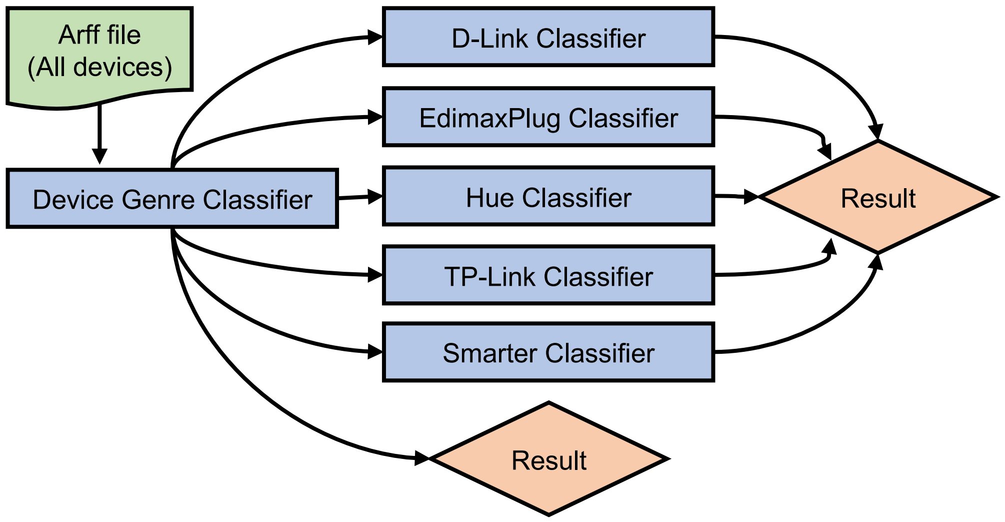


FIGURE 7.3: Classifiers

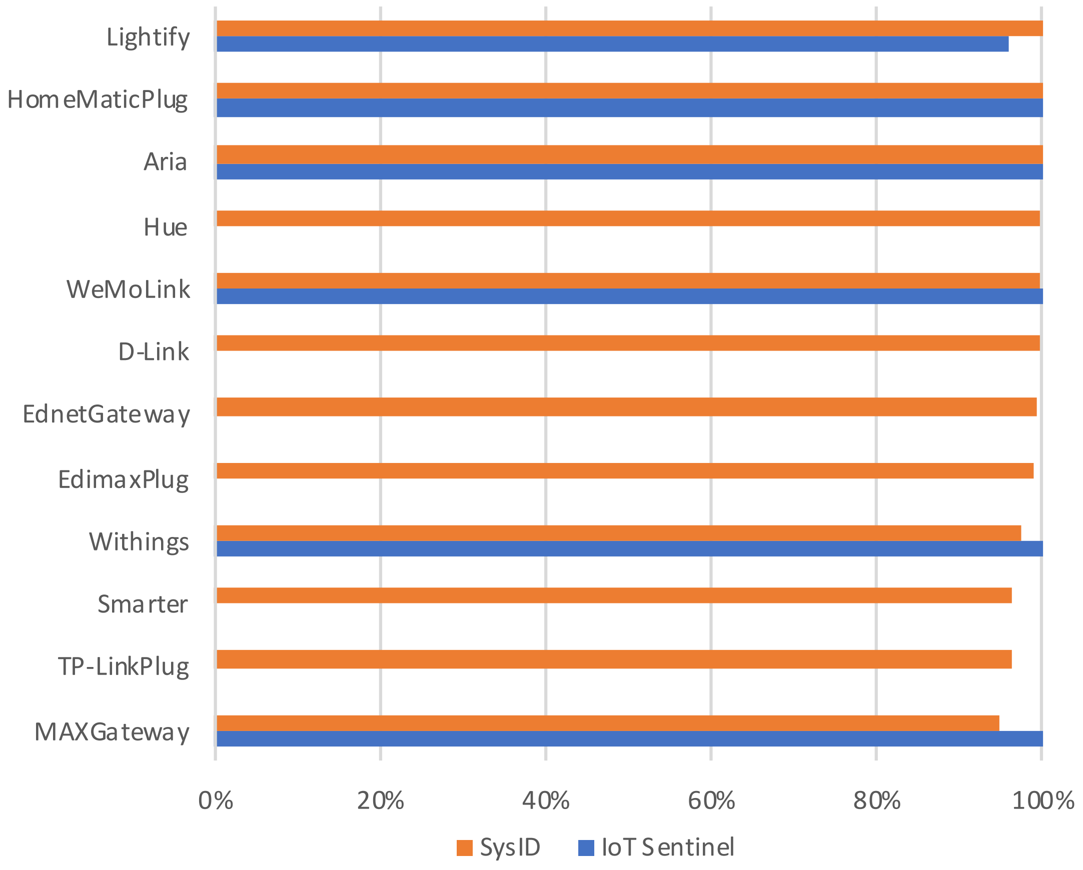


FIGURE 7.4: Device genre classification accuracy

We then identify devices belonging to the same vendor. The PART algorithm was able to differentiate between Hue devices with high accuracy, as shown in

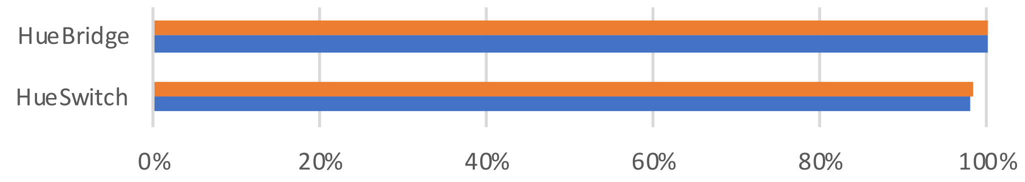


FIGURE 7.5: Hue devices classification accuracy

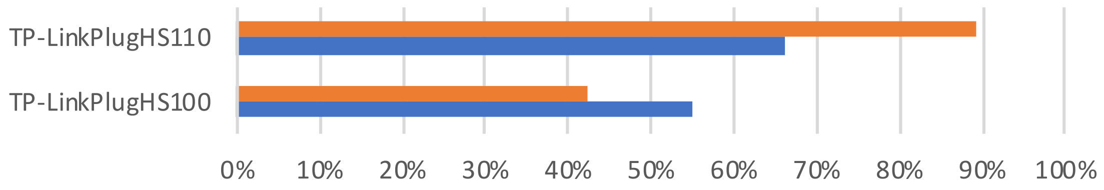


FIGURE 7.6: TP-LinkPlug devices classification accuracy

Figure 7.5. SysID can classify any individual packet originated from a HueSwitch with a rate of 98.8% and a HueBridge with a rate of 95.9%.

Similarly, the PART algorithm provided the best identification of TP-LinkPlug devices in Figure 7.6. We observed that SysID could determine TP-LinkPlugHS110 with an accuracy rate of 89.3% and TP-LinkPlugHS100 with 42.2% accuracy. We observed that the classifier has difficulty in distinguishing these two devices. Note that IoT Sentinel was unable to identify these devices with high accuracy either. The reason for one of these two devices to have a higher classification accuracy is due to the order of rules that the classifier uses.

Additionally, the J48 algorithm provided the best identification of EdimaxPlug devices in Figure 7.7. SysID can distinguish EdimaxPlug1101W at an accuracy rate of 70% and EdimaxPlug2101W at a rate of 72%. Although the classifier cannot make a perfect distinction between these two devices, it can still tell them apart at a rate of 70%, which is higher than the one-level classification results of Section 7.2.2.

J48 algorithm also provided the best identification of D-Link devices in Figure 7.8. SysID achieved identification accuracy of 94.2% for D-LinkCam, 86.4%

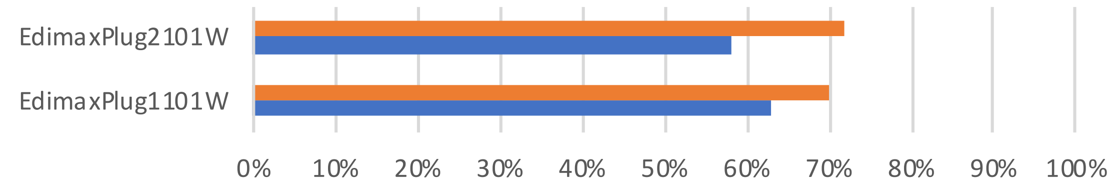


FIGURE 7.7: EdimaxPlug devices classification accuracy

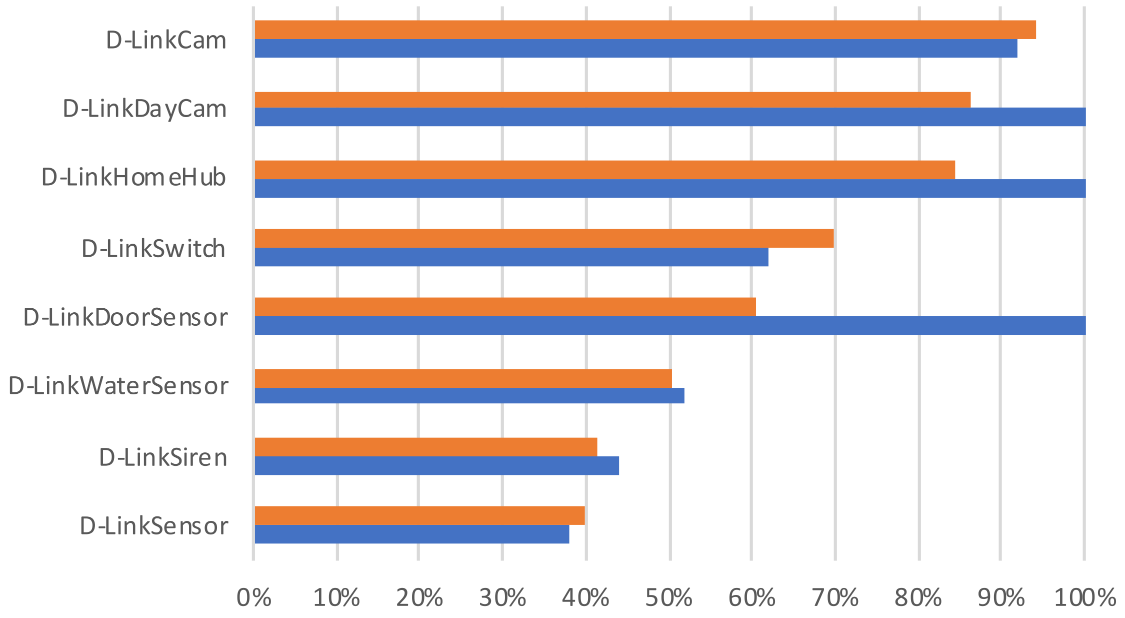


FIGURE 7.8: D-Link devices classification accuracy

for D-LinkDayCam, 84.5% for D-LinkHomeHub. However, the remaining devices perform at lower classification rates indicating the possibility of similar single packet behavior. Note that IoT Sentinel had a higher classification rate for D-LinkDayCam, D-LinkHomeHub, and D-LinkDoorSensor. The possible reason for this is that IoT Sentinel considers a sequence of 12 packets, and this may be helpful to identify the distinctive behavior of these devices.

Finally, the J48 algorithm provided the best identification of Smarter devices in Figure 7.9. SysID can determine if a packet originated from an iKettle2 with an accuracy of 93% or from a SmarterCoffee with a rate of 100%. These results tell us that these two devices have very distinguishable behaviors that were not captured by IoT Sentinel.

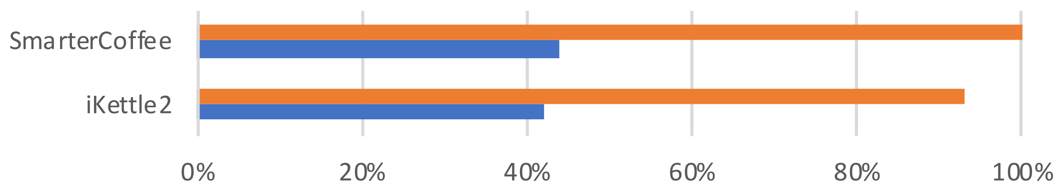


FIGURE 7.9: Smarter devices classification accuracy

TABLE 7.2: Number of selected features

Classifier	Algorithm	GA	ALL	%
Device genre	PART	93	212	44%
Hue	J48	78	147	53%
TP-Link	PART	43	115	37%
EdimaxPlug	J48	51	123	41%
D-Link	J48	90	204	44%
Smarter	J48	8	55	15%

7.2.4 Feature Subset Selection

SysID aims to increase fingerprinting accuracy by removing possible noisy data and decreasing complexity by reducing the number of features considered using GA. Table 7.2 provides the number of features selected by GA for each classifier. Except for Hue devices, using less than half of the features leads to better classification accuracy than when all of the features are used. SysID was able to fingerprint Smart devices at an average accuracy of 96% while using only 15% of the packet header features.

GA helps to automatically select a subset of features that contribute to the fingerprinting the most. However, for every classifier, the set of selected features differs. Table 7.3 presents the top 10 features selected by the classification algorithms for device genre and each vendor classifier. We indicate whether a particular packet header feature was selected for each classifier. We observe that *udp.checksum* is the most preferred feature as it was selected among the top 10

TABLE 7.3: Selected Features

Features	All	Genre	Hue	TP-Link	EdimaxPlug	D-Link	Smarter
udp.checksum	✓	✓		✓	✓	✓	✓
dns.qry.class	✓	✓				✓	
ip.len	✓			✓		✓	
tcp.window_size	✓	✓	✓				
tcp.flags		✓	✓			✓	
icmp.checksum	✓	✓					
ip.dsfield	✓	✓					
dns.resp.len						✓	
ip.flags				✓			
ip.id		✓			✓		
ip.proto					✓		
ip.ttl	✓		✓			✓	
tcp.seq			✓			✓	
tcp.ack						✓	
tcp.options.timestamp.tsval			✓	✓	✓		
tcp.port			✓	✓	✓		
tcp.stream			✓	✓			
tcp.window_size_scalefactor					✓	✓	
udp.dstport		✓		✓			
udp.stream					✓		
ip.flags.df	✓						
tcp.option_len	✓						
tcp.dstport		✓					
tcp.hdr_len		✓					
tcp.window_size_value			✓				
tcp.analysis.acks_frame			✓				
tcp.flags.push			✓				
udp.srcport				✓			
ip.dsfield.dscp				✓			
dns.flags					✓		
dns.flags.response					✓		
dns.qry.name.len					✓		
tcp.options.timestamp.tsecr						✓	
Σ (features)	9	10	10	9	10	10	1

by all the classifiers except for Hue devices. We believe *udp.checksum* can capture a unique periodic packet from these devices because the UDP checksum is calculated over the data as well as the IP and UDP header fields.

7.3 Discussion

SysID's accuracy is similar to the IoT Sentinel. The average classification accuracy for the cumulative of all the devices were 79% for IoT Sentinel and 82% for SysID. IoT Sentinel was able to achieve these results using a sequence of 12 packets whereas SysID uses only a single packet for fingerprinting. IoT Sentinel benefits from investigating a sequence of packets, whereas we benefit from investigating the header field content. Another possible bias that IoT Sentinel may have is the consideration of link-layer packets. In SysID, we ignored link-layer protocols since it is highly likely for these protocol contents to have information specific to devices in a network. We aimed to generate a comprehensive model-based system that can be applied to different networks.

Overall, SysID has several general advantages and unique features. First, certain device fingerprinting tools depend on specific information extracted from network protocols. The downside of these approaches is that in case of a change of the behavior of the protocol or its fields, these approaches would not be able to perform fingerprinting correctly. Although we detect and use such distinguishing features from the protocols, SysID can re-process new data after protocol change to re-extract new distinguishing features without expert supervision.

Since we extract features directly from the standard protocol header fields, the fingerprints that we extract are stable. Such fingerprints can be used when a different environment or even the mobility of devices is considered. Depending on which protocol header fields are used, SysID can generate fingerprints that are either very general or very specific depending on the need.

Different from other approaches, we do not hand-select *useful* features such as packet size, port number, or IP header options. GA helps SysID to automatically detect the set of the most useful features in a given dataset. Although the dataset used in this study contained fewer than 50 packets for some devices, SysID was still able to detect the unique behaviors of IoT devices and obtain very high classification accuracy.

In particular, SysID performs a single-packet fingerprinting of devices using any of the packets generated from the device. Single-packet fingerprinting is useful for both reducing the complexity of classification and also for reducing expectations from the targeted device in terms of both the number of packets and a sequence of packets to be generated.

Chapter 8

Automated Quantization of Numerical Features using K-means and Genetic Algorithms

In this section, we propose an automated approach to determining the quantization levels for numerical features in inductive learning. For each numeric feature, we use k-means to determine the clusters and their ranges. As known, it is essential to initially specify the number of clusters when running the k-means algorithm. One possible approach is to start with $k = 1$, k being the number of clusters, and to keep incrementing until a certain number is reached to find the optimal number of clusters. However, different numbers of clusters for each numeric feature can yield different classification accuracy. For example, if the maximum number of clusters for each feature to be considered is 10, there could be up to 10^n number of combinations to process where n is the number of features within a dataset. Depending on what the value of n is, it could be very computationally intensive to find an optimal solution.

We use a genetic algorithm (GA) to determine the number of clusters for each numeric feature in a given dataset to select as optimal solution as possible. GA helps reduce the enormous amount of a possible number of cluster combinations for numeric features. It also helps determine the most information-gaining ones in order to provide as high classification accuracy as possible.

Every time GA selects a potential solution, the number of clusters for each numeric feature is determined from the selected chromosome. Then, a portion of the dataset is trained using the SILEA algorithm with the number of clusters provided as a parameter. The user can adjust the portion of the dataset dedicated to training and testing. The generated rule set is then used to perform classification on the remaining portion of the data to obtain the classification accuracy. The accuracy retrieved from this execution is returned as the fitness value for the considered solution and is used by GA to compare different solutions to find the best solution.

In this section, we automate the process of finding the number of clusters for each numeric feature to determine as optimal quantization levels as possible to increase the classification accuracy of the SILEA algorithm. We use a GA to determine the most information-gaining combination of the number of clusters. Then we use the k-means algorithm to determine the ranges for these clusters for each feature. Our approach allows us to completely automate hyper-parameter optimization using GA for inductive learning algorithms, precisely the SILEA algorithm.

8.1 K-means Clustering

The most important problem with the quantization of numeric features is to determine where to split the data points. SILEA employs a straightforward approach where the data points are split up in equal sizes of chunks in the number provided by the user manually. The downside of this approach is that the data is very unlikely to be split in such equal chunks. We needed to optimize the start and endpoints for each cluster based on the data. Therefore, we employed the k-means algorithm to be able to split the data points into clusters dynamically. K-means clustering is an unsupervised learning algorithm. K-means is usually used on datasets which are not labeled. K-means tries to cluster the data based on a similarity measure.

For each numeric feature, we initially sort the values incrementally and run the k-means algorithm to determine the clusters for the considered feature. Once the clusters are determined, we determine the ranges for each of these clusters and make sure that the data points for the feature considered are replaced with the ID of the cluster whose range they fall. Assume there exist 2 clusters within a feature and they range between 1-5 and 20-30. Every data point within the feature that is between 1-5 are replaced with the ID of this cluster, which is 1, and every data point that is between 20-30 are replaced with the ID of 2, which allows SILEA to extract more general rules. This transformation, however, is stored in the model file to be able to map the data points to clusters when running the algorithm on a testing dataset. However, the k-means algorithm requires the number of clusters to be manually set. A different number of clusters provided as an input may generate different results which could yield different classification accuracy. The reason for that is, for certain features, having too general rules does not always help catch uniqueness. Therefore, selecting a suitable number of clusters as input when k-means is run can be very crucial in terms of achieving high classification accuracy. One possible way is for the user to try different values manually until the desired accuracy is obtained. However, this can be very time-consuming, especially if there exist many numeric features in the dataset. If the user would like to test m number of clusters for each feature and if there are n number of features in the dataset, there could be up to m^n possible number of clusters to run the algorithm, which is not very practical. Therefore, we employed a genetic algorithm (GA) both to increase the speed and accuracy of searching within such a large domain.

8.2 Genetic Algorithm Optimization

In order for a genetic algorithm (GA) to be able to validate the contribution of a potential solution to the classification of the dataset, we need a portion of the training data to be used for validation. We initially randomly sort the examples within the dataset provided. Then, we split the dataset into two parts, where the first portion is used to train, and the second portion is used to validate the GA. The percentage of the dataset to be used to train and validate is a parameter and can be adjusted by the user. After trying several percentages of the data for training and validation, we observed nice results when we used 60% for training and 40% for testing, which also assures an acceptable amount of data for validation for GA to rely on.

In our GA implementation, we used a population size of 50 chromosomes, a uniform rate of 0.5 and a mutation rate of 0.05. A chromosome is a series of 0's and 1's. Since the necessary information for the number of clusters to be used is embedded into a chromosome, the number of bits within a chromosome is proportional to the number of numeric features within the dataset. Depending on how large or small the search space is desired to be kept, the number of bits dedicated to one feature can be adjusted by the user. However, it is essential to keep in mind that if this number is too small, it may not be possible to find a fine-tuned set of number of clusters for the dataset and if it is too large, it may take a very long time for GA to converge to a solution. To find a more fine-tuned and information-gaining solution for the dataset we used, we dedicated 5 bits in a chromosome for each numeric feature. Therefore, the length of a chromosome in our implementation is $5n$, where n is the number of numeric features in the dataset. As mentioned earlier, this parameter can be adjusted by the user. Five bits for a feature means that GA can select a cluster number ranging between 0 and $2^5 - 1 = 31$, which means that for each numeric feature, a value between 0 and 31 can be used as the number of clusters to be fed to the k-means algorithm when a solution is being tested. However, we observed that a difference of 1 for the number of clusters does not help with the accuracy much since there exist features in the dataset with a large range of possible data points. Therefore, rather than using these values as the number of clusters with such small increments, we multiplied the decimal value represented by 5 bits with 10, which means that the possible number of clusters used for each numeric feature ranges between 0 and 310 with an increment of 10.

Every potential solution selected by GA is tested using a fitness function to determine how much contribution they make to the classification of the dataset using SILEA. In our implementation of the fitness function, every time GA tests a potential solution; we split the chromosome into 5 bits of chunks. For each chunk, we determine the value they represent as a decimal number. Then, we multiply these values by 10. After this process, there exist n number of clusters. Then, for each numeric feature, we run the k-means algorithm initiated with the number of clusters determined from the chromosome. After determining the clusters and their ranges for each numeric feature, SILEA replaces the data points in the training portion of the dataset with the ID of the clusters and extracts the rules. Then, we use the rules extracted on the validation portion of the dataset and determine the accuracy achieved when classifying the examples in this portion. The fitness function returns the accuracy retrieved as the fitness of the solution provided as an input. GA utilizes the fitness values for each chromosome to find as optimal solution as it can.

GA is not guaranteed to converge to the optimum solution that would yield the highest fitness value. Therefore, we limit the number of generations that is run by GA and terminate GA to prevent an infinite loop. We make sure that the GA keeps running as long as the highest fitness value achieved with a generation is higher than the previous one, which means that GA is most likely to keep increasing its accuracy by running new generations. However, if we observe repetitive occurrences of the same accuracy with consecutive generations, we terminate GA. The critical point here is to determine how many consecutive occurrences of the same accuracy should make GA terminate. We chose to terminate GA after five consecutive occurrences of the same accuracy. It is essential to choose this value carefully since a low value may terminate GA before reaching a higher accuracy yielding solution, and a high value may take a very long time for GA to converge. We observed in our previous work that selecting 5 for this value yields both high accuracy results and for GA to converge to a solution in an acceptable amount of time [27], [29].

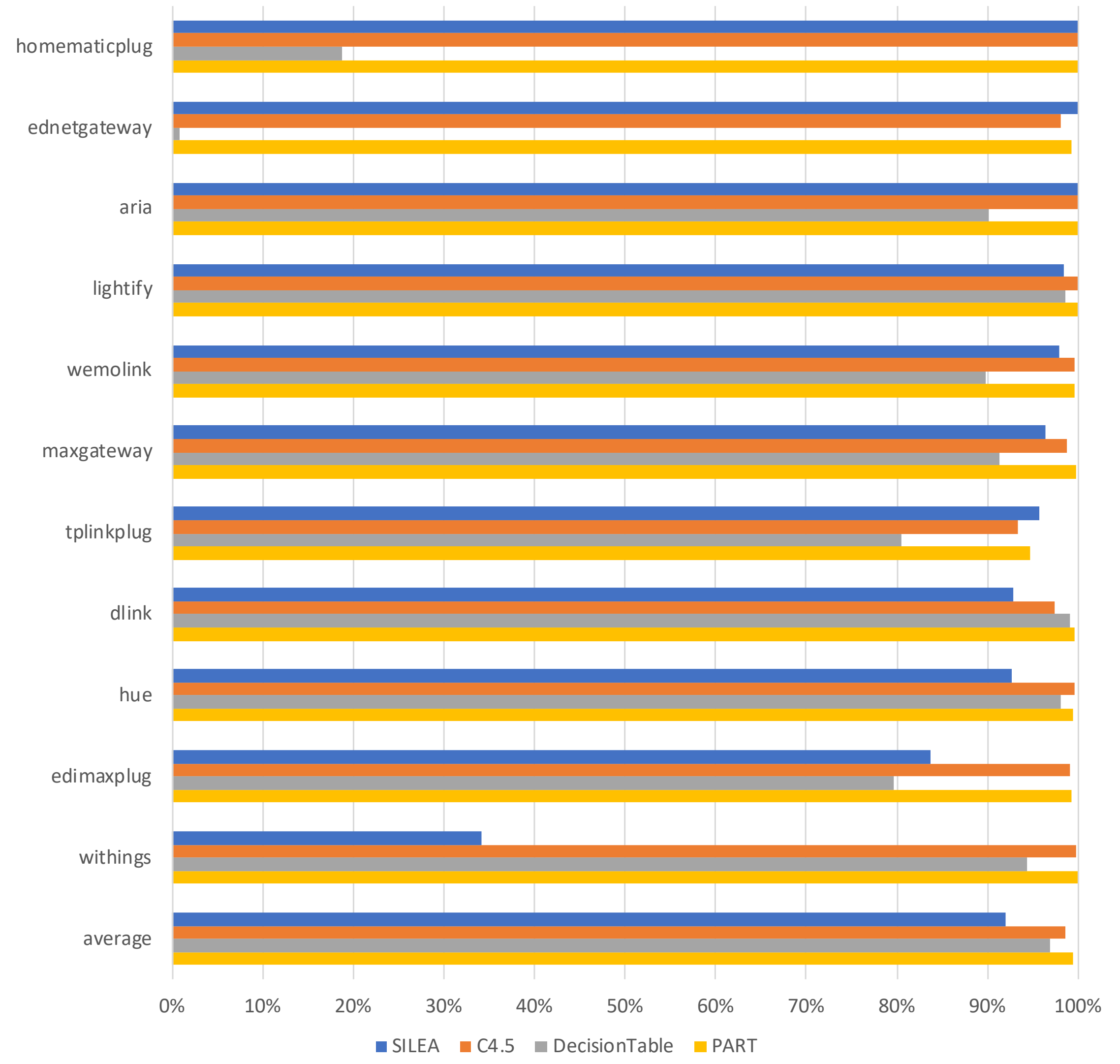


FIGURE 8.1: Device genre classification performance

8.3 Classification

After determining the best possible number of clusters to be used for each numeric feature, we determine how well our classifier performs with a testing dataset. As mentioned earlier, the genetic algorithm (GA) initially uses a portion of the training dataset to train and the other portion to validate. However, after GA completes searching and selecting a suitable solution, we use the entire examples in the training set to train our model. We run the k-means algorithm on numeric features using the number of clusters selected by GA and extract rules. Then, we use the rules extracted to classify the examples on a testing dataset which is never used until this point.

8.4 Experimental Results

In this section, we present the classification accuracy of SILEA with automated quantization using data measurements from 23 IoT devices. These devices consist of home sensors, coffee makers, power switches, and light bulbs. We also compare our results with some of the state-of-the-art inductive learning algorithms.

8.4.1 Data

In this study, we used the dataset collected by [77]. The authors perform device fingerprinting by extracting a fingerprint for each device from their network traffic. Among the packets collected, we utilized the data for 23 IoT devices. For consistency reasons, data belonging to 4 devices were removed since they did not have as many number of measurements as the rest. The devices whose data we used are: *Aria*, *D-LinkCam*, *D-LinkDayCam*, *D-LinkDoorSensor*, *D-LinkHomeHub*, *D-LinkSensor*, *D-LinkSiren*, *D-LinkSwitch*, *D-LinkWaterSensor*, *EdimaxPlug1101W*, *EdimaxPlug2101W*, *EdnetGateway*, *HomeMaticPlug*, *HueBridge*, *HueSwitch*, *iKettle2*, *Lightify*, *MAXGateway*, *SmarterCoffee*, *TP-LinkPlugHS100*, *TP-LinkPlugHS110*, *WeMoLink*, and *Withings*.

There are 20 measurements for each of the devices we used in our analysis. We dedicated 70% of these runs for training and validation, and the remaining 30%

for testing the accuracy of our approach. For each packet, we extracted features for the following protocols: DNS, HTTP, ICMP, IP, SSL, TCP, and UDP. If a packet does not contain header fields for any of these protocols, we add a 'null' value. To avoid bias, we removed the IP address and the IP checksum from every packet.

8.4.2 Classification Accuracy

We perform single-packet IoT device identification using SILEA with an automated quantization approach. We extract rules with the SILEA algorithm from TCP/IP packet headers which are later used to identify which specific vendor and version the device which sent this packet belongs.

We initially tested the accuracy when k-means was used with a preset number of clusters for every numeric feature in the dataset. Then, we compared our results when the number of clusters is automatically selected using a genetic algorithm (GA). For consistency reasons, when we set the number of clusters for k-means to try the accuracy without a GA, we used the maximum possible number of clusters which would have been selected by the GA, which is 310. As seen in 8.6, when detecting the genre of the IoT devices, we observed a maximum of 87% accuracy. However, when GA is used, the accuracy increases to 92%. We observed an increase from 65% to 76% in the accuracy when determining the TP-LinkPlug device version and almost 10% increase from 52% to 61% when classifying the EdimaxPlug devices. For D-Link device identification, we observed the least increase in accuracy from 31% to 33%, which is due to the similarities of libraries used by the D-Link devices. We observed the highest increase in classification when classifying Smarter devices. We were able to increase the classification accuracy almost twice as much from 54% to 96%. There is also a case where GA was not able to generate the highest accuracy. When classifying Hue devices, although the accuracy is still high, we observe a 2% reduction in the accuracy from 98% to 96%, which shows that there are cases which might require a better fine-tuning and a more prolonged execution of the GA. One of the parameters to achieve higher accuracy is to increase the number of repetitions in the occurrences of generations before termination. With a higher termination point set, it would allow GA to explore longer, which could help achieve a higher accuracy yielding solution. Especially if the data points in the dataset are further apart from each other, selecting a higher number of clusters would be a requirement. Therefore, another parameter that needs to be optimized is the maximum number of clusters that the chromosome can represent.

As seen, GA can determine a more suitable number of clusters for the dataset at hand and provide higher accuracy than when a preset number of clusters are used. We also compared the classification accuracy of our approach with some of the state-of-the-art inductive learning algorithms. We compared our results with C4.5, DecisionTable, and PART algorithms. Due to the initial randomization of the population in GA, we made sure to run GA at least five times and selected the best result for each of the algorithms.

In Figure 8.1, we provide the accuracy of detecting the vendor of the devices from a single packet classification. We provide the accuracy for each vendor from the dataset and also provide the average accuracy for each classifier. As seen, on average, SILEA was able to distinguish the vendor of the devices correctly at a rate of 92%. C4.5 and PART algorithms were able to classify at a rate of 99%, and DecisionTable was able to classify at a rate of 97%. As seen, for IoT

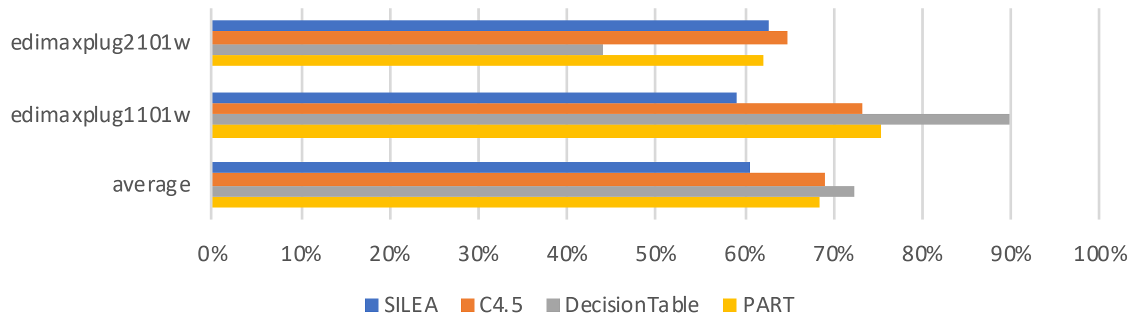


FIGURE 8.2: EdimaxPlug classification performance

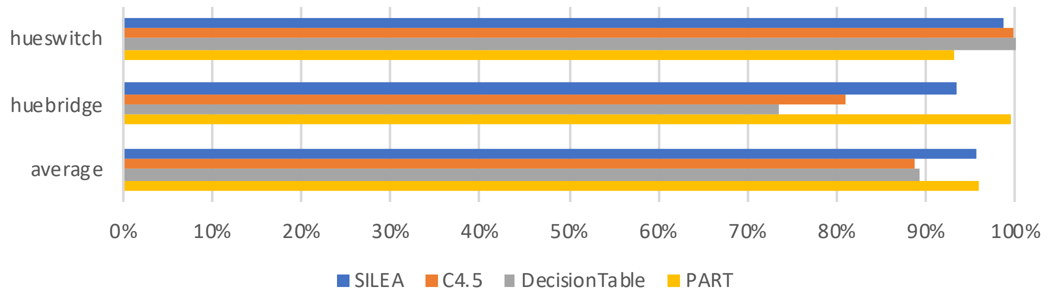


FIGURE 8.3: Hue classification performance

device genre classification, SILEA was not able to achieve as high classification results as the rest.

Similarly, as seen in Figure 8.2, SILEA was not able to perform as well as the others in determining the specific versions of the EdimaxPlug devices. SILEA's classification accuracy was at 61%, C4.5 was at 69%, DecisionTable was at 72%, and PART was at 68%.

However, as seen in Figure 8.3, when classifying the device versions for Hue, Smarter and TP-LinkPlug devices, SILEA was able to perform as high or even higher than the other algorithms. For classifying Hue devices, SILEA and PART were able to classify with the highest accuracy of 96%. The classification accuracy of C4.5 and DecisionTable were at 89%.

As seen in Figure 8.4, when classifying the Smarter devices, all of the algorithms were able to perform classification at the same rate of 96%. One of the reasons why all algorithms perform at the same rate is due to the number of instances

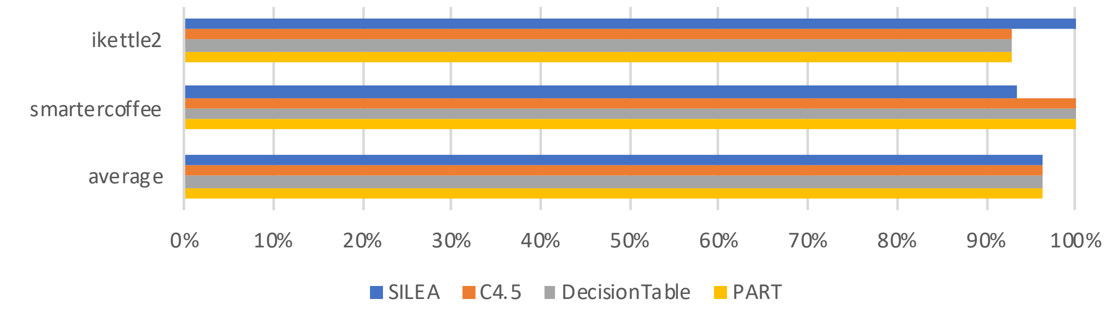


FIGURE 8.4: Smarter classification performance

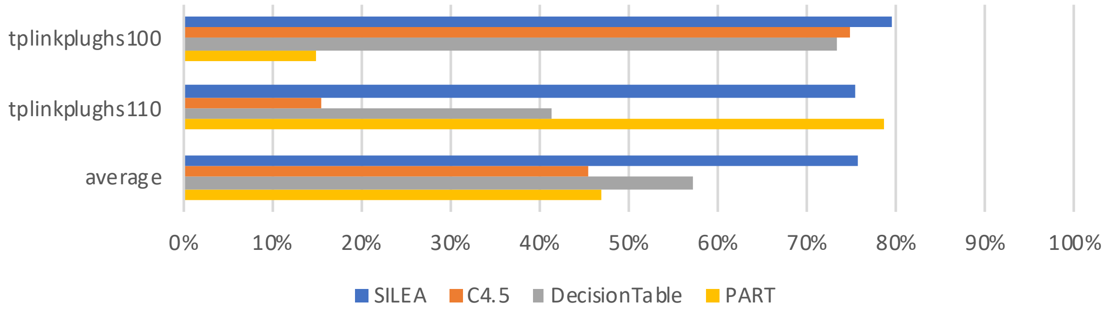


FIGURE 8.5: TP-LinkPlug classification performance

belonging to Smarter devices. Smarter devices, when the data was collected, seems not to have created as much traffic as the others. Therefore, Smarter devices have the least amount of packets in the dataset. Also, the devices were easily distinguishable using merely the UDP checksum. Therefore all of the algorithms were easily able to capture this uniqueness.

The classification of TP-LinkPlug devices is where SILEA excels all the other algorithms. As seen in Figure 8.5, on average, SILEA was able to achieve 76% accuracy. The DecisionTable algorithm achieved the closest accuracy at 57%. C4.5 and PART algorithms, however, performed the worst at the rates of 45% and 47%, respectively.

We observed that, although there are cases when SILEA performs as well and even better than the other algorithms, there are cases where it does not perform as well as the others. One of the reasons why SILEA cannot outperform the

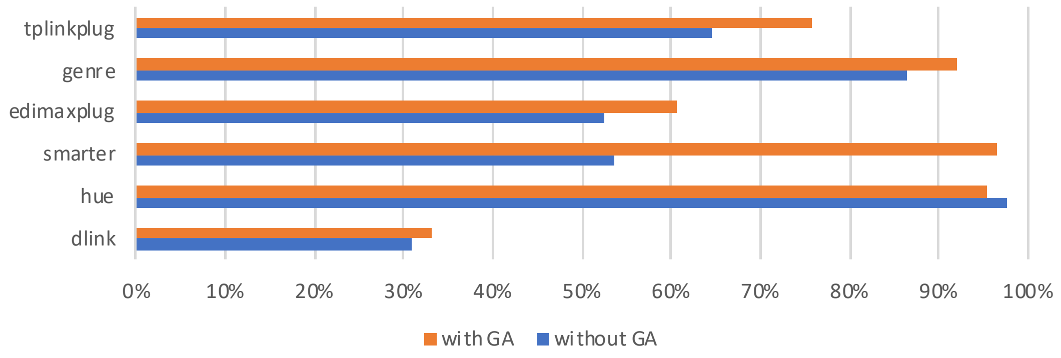


FIGURE 8.6: Classification accuracy with and without GA

other algorithms is that SILEA makes sacrifices in considering certain combinations of features to reduce the complexity. However, we expect the classification accuracy to increase when the size of the chromosome to represent a more comprehensive set of ranges for numeric features is increased. Also, by increasing the number of repetitions allowed for the termination point, it is expected for GA to be able to search for a better solution. Although SILEA with k-means might require more time to run, it has the advantage of generating the model that is fine-tuned for the dataset.

Chapter 9

Conclusion

In this dissertation, we first presented SILEA, a simple yet accurate inductive learning algorithm. SILEA tries to minimize the enormous amount of possible consideration of instances, i.e., O($n.n!$), to a reasonable number, i.e., O(n^2), without sacrificing from its accuracy. Two factors employed in the algorithm recovers the expected accuracy drop from minimizing the number of combinations of attributes to be considered. The first is that the algorithm, for the given combination, extracts all the rules and selects those that can classify the most examples. During the selection phase, the algorithm also discards any rules that are made obsolete by other rules with higher classification capabilities. The second is that, since the algorithm favors certain attributes over others when performing combination reduction, it is made sure that these biased attributes are those which excel over all others based on their entropy values. SILEA, however, does not guarantee the most general rules for a given dataset. Even though the algorithm assures the generation of the most general rules for the considered combination, it can miss more general rules regarding the overall extraction since it eliminates certain combinations in each iteration. We compared the classification accuracy of SILEA to some of the well-known algorithms in the inductive learning field using five different datasets.

On average, SILEA was 8.8% better than RULES3, 12.1% better than RULES3-Plus, 4.2% better than C4.5, 5.8% better than CN2, 9.8% better than RIPPER, 14.4% better than RIDOR, 4.4% better than PART, 13.7% better than DecisionTable, and 8.0% better than RandomTree algorithms.

We then presented OSID which is an entirely automated, machine learning, and Genetic Algorithm (GA) techniques dependent operating system (OS) identification system. OSID can perform any single-packet OS identification with high accuracy. It uses GA to select smaller and more efficient sets of protocol header features to perform OS identification. It generates signatures (i.e., models of protocol header characteristics) with the help of various machine learning algorithms. The use of GA and machine learning algorithms allows OSID to adapt itself to new OS implementations. Unlike current tools for OS identification, OSID does not necessarily depend on specific types of packets to perform identification. Although the system can be restricted to process specific protocols to assure the highest possible identification results, it can also perform high identification rates when considering packets belonging to various protocols. A single packet approach allows OSID to perform identification on any packet.

In our analysis, we observed that OSID could achieve up to 99% average accuracy when detecting the OS family using ICMP packets, 99% when detecting the Linux distros using ICMP packets and 98% when detecting the Mac OS versions using SSL packets. We also compared our results to one of the most prominent passive OS identification tools, p0f. We observed that OSID was able to distinguish the OSes and their versions more reliably with lesser restrictions.

Next, we present SysID, a completely automated single-packet IoT device classifier using machine learning and GA. GA helps in determining relevant features in packet headers to both increase classification accuracy and reduce complexity.

In most cases, we were able to use less than half of the features in packet headers to classify devices at a rate higher than when all the features were used. GA also helps eliminate noisy features that negatively affect classification accuracy. We also analyzed multiple machine learning algorithms to determine the best classifiers to fingerprint devices at each layer of classification. Overall, SysID was able to classify devices at a minimum accuracy rate of 95%. In some cases, SysID could distinguish between devices from the same vendor due to the similarity of their network behaviors.

We also presented the contribution levels of popular protocols to classify the OS of hosts and IoT vendors from which the packets originated. We examined how well certain machine learning algorithms performed for classification from TCP/IP protocol headers. By using GA to select the most distinguishing features, we demonstrated the contribution levels of various features in classifying OSes and IoT devices while reducing computation overhead. Since classification was performed individually on the packets, the results obtained are not bound to restrictions such as classification of certain packet types only (e.g., SYN packets in TCP).

In general, it is time-consuming to generate signatures due to the complexity of GA. However, feature selection can be performed offline on High Performance Computing (HPC) platforms. Also, the training is performed once and only needs to be repeated with the introduction of a new OS or device. After generating the signatures, the models for the machine learning algorithms can be shared across users. Another possible restriction is the lack of representative data. Since OSID and SysID extract models from the data, the data needs to be as representative of real-life scenarios as possible. Although our data for OS classification was not too large, we were still able to observe very high classification results.

Chapter 10

Future Work

The presented SILEA inductive learning algorithm can be improved for different scenarios by incorporating other approaches. For instance, the algorithm can employ a better range calculation technique to handle continuous attributes more accurately. Another improvement can be made on the rule extraction process by introducing error tolerance in the rules it generates. Every time new training data is used to train the model, the entire process is required to be repeated with the previous data. An incremental version of SILEA could help address this issue by allowing the models to be updated without the need to regenerate rules from the data that has already been processed. For future work, we will investigate these improvements to SILEA.

Although OSID can classify OSes with very high accuracy using a single-packet, an improved packet sequence classification approach could help increase accuracy even further. An important area of research to perform classification with packet sequence is to determine weights for each protocol's packets. In this study, we observed that UDP protocol packets do not contribute much to the classification of OSes, but TCP protocol does. Therefore, regardless of how many packets belonging to the UDP protocol are observed within a sequence, we need a weight-based mechanism which will give higher precedence to protocols such as TCP over others. We also believe that by considering textual features in packet headers, we can increase our classification accuracy. We would also like to test the classification accuracy of background traffic and user-generated network traffic. We will include mobile devices and other versions of the OS families in the future. Note that even though we have not tested our approach on IPv6, we believe that it can be applied to perform classification on IPv6 packets as well.

We also would like to analyze packet sequences for IoT device fingerprinting, which would possibly allow higher classification accuracy. We would also like to extend our analysis to optimize GA. As we observed, some of the selected features were not utilized by machine learning algorithms; we believe that SysID could have achieved higher accuracy. We want to analyze time-related features to detect actual behaviors of devices rather than merely performing packet classification. We believe such an approach will provide more accurate fingerprints for IoT devices and also provide the ability to extract unique fingerprints even when numerous devices are introduced.

Bibliography

[1] M. H. Gunes, "Complex network discovery: Router-level internet topology mapping," AAI3323614, PhD thesis, Richardson, TX, USA, 2008, ISBN: 978-0-549-75832-7.

[2] J. Naruchitparames, M. H. Gunes, and C. Y. Evrenosoglu, "Secure communications in the smart grid," in *2011 IEEE Consumer Communications and Networking Conference (CCNC)*, 2011, pp. 1171–1175.

[3] H. A. J. Narayanan and M. H. Gunes, "Ensuring access control in cloud provisioned healthcare systems," in *2011 IEEE Consumer Communications and Networking Conference (CCNC)*, 2011, pp. 247–251.

[4] D. T. Pham, S Bigot, and S. S. Dimov, "Rules-5: A rule induction algorithm for classification problems involving continuous attributes," *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, vol. 217, no. 12, pp. 1273–1286, 2003.

[5] D. T. Pham and A. A. Afify, "Rules-6: A simple rule induction algorithm for supporting decision making," in *31st Annual Conference of IEEE Industrial Electronics Society, (IECON 2005)*, Nov. 2005, pp. 2184–2189.

[6] D. T. Pham and M. S. Aksoy, "A new algorithm for inductive learning," *Journal of Systems Engineering*, vol. 5, no. 2, pp. 115–122, 1995.

[7] J. R. Quinlan, "Learning logical definitions from relations," *Machine learning*, vol. 5, no. 3, pp. 239–266, 1990.

[8] A. Aksoy and M. H. Gunes, “Silea: A system for inductive learning,” in *2016 7th International Conference on Information, Intelligence, Systems & Applications (IISA)*, IEEE, 2016, pp. 1–8.

[9] A. W. Whitney, “A direct method of nonparametric measurement selection,” *IEEE Transactions on Computers*, vol. 20, no. 9, pp. 1100–1103, 1971.

[10] L. Burrell, O. Smart, G. K. Georgoulas, E. Marsh, and G. J. Vachtsevanos, “Evaluation of feature selection techniques for analysis of functional MRI and EEG,” in *DMIN*, 2007, pp. 256–262.

[11] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein, “Passive operating system identification from tcp/ip packet headers,” in *Workshop on Data Mining for Computer Security*, Citeseer, 2003, p. 40.

[12] F. Beck, O. Festor, and I. Chrisment, “Ipv6 neighbor discovery protocol based os fingerprinting,” PhD thesis, INRIA, 2007.

[13] B. Li, J. Springer, G. Bebis, and M. H. Gunes, “A survey of network flow applications,” *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 567–581, 2013.

[14] A. Aksoy and M. H. Gunes, “Operating system classification performance of tcp/ip protocol headers,” *Local Computer Networks Workshops (LCN Workshops), 2016 IEEE 41st Conference on*, 2016.

[15] B. Li, M. H. Gunes, G. Bebis, and J. Springer, “A supervised machine learning approach to classify host roles using sflow,” in *HPDC - Workshop on High Performance and Programmable Networking (HPPN)*, ACM, 2013.

[16] F. Veysset, O. Courtay, O. Heen, I. Team, *et al.*, “New tool and technique for remote operating system fingerprinting,” *Intranode Software Technologies*, 2002.

[17] O. Arkin, “A remote active os fingerprinting tool using icmp,” *login: the Magazine of USENIX and Sage*, vol. 27, no. 2, pp. 14–19, 2002.

[18] G. Taleck, “Synscan: Towards complete tcp/ip fingerprinting,” *CanSecWest, Vancouver BC, Canada*, 2004.

[19] A. J. Bennieston, *Nmap-a stealth port scanner*, 2004.

[20] J. Schwartzenberg, “Using machine learning techniques for advanced passive operating system fingerprinting,” Master’s thesis, Jun. 2010.

[21] C. Sarraute and J. Burroni, “Using neural networks to improve classical operating system fingerprinting techniques,” *arXiv preprint arXiv:1006.1918*, 2010.

[22] D. Fifield, A. Geana, L. MartinGarcia, M. Morbitzer, and J. Tygar, “Remote operating system classification over ipv6,” in *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, ACM, 2015, pp. 57–67.

[23] M. Zalewski, *P0f v3 (version 3.07 b)*.

[24] L. Spitzner, “Passive fingerprinting,” *FOCUS on Intrusion Detection: Passive Fingerprinting (May 3, 2000)*, pp. 1–4, 2000.

[25] Y.-C. Chen, Y. Liao, M. Baldi, S.-J. Lee, and L. Qiu, “Os fingerprinting and tethering detection in mobile networks,” in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ACM, 2014, pp. 173–180.

[26] K. Zandberg, “Passive fingerprinting on an ipv6-enabled network,” B.S. thesis, University of Twente, 2016.

[27] A. Aksoy, S. Louis, and M. H. Gunes, “Operating system fingerprinting via automated network traffic analysis,” in *Evolutionary Computation (CEC), 2017 IEEE Congress on*, IEEE, 2017, pp. 2502–2509.

[28] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, “Ddos in the iot: Mirai and other botnets,” *Computer*, vol. 50, no. 7, pp. 80–84, 2017, ISSN: 0018-9162.

[29] A. Aksoy and M. H. Gunes, "Automated iot device identification using network traffic," in *IEEE International Conference on Communications*, IEEE, 2019.

[30] M. S. Aksoy, "A review of rules family of algorithms," *Mathematical and Computational Applications*, vol. 13, no. 1, p. 51, 2008.

[31] J. R. Quinlan, "Generating production rules from decision trees," in *Proceedings of the 10th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'87, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 304–307.

[32] D. T. Pham and S. S. Dimov, "An efficient algorithm for automatic knowledge acquisition," *Pattern Recognition*, vol. 30, no. 7, pp. 1137–1143, 1997.

[33] J. Cheng, U. M. Fayyad, K. B. Irani, and Z. Qian, "Improved decision trees: A generalized version of id3," in *Proc. Fifth Int. Conf. Machine Learning*, 1988, pp. 100–107.

[34] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, ISBN: 1-55860-238-0.

[35] B. Hssina, A. Merbouha, H. Ezzikouri, and M. Erritali, "A comparative study of decision tree id3 and c4.5," *International Journal of Advanced Computer Science and Applications*, vol. 4, no. 2, pp. 13–19, 2014.

[36] R. S. Michalski, "On the quasi-minimal solution of the general covering problem," *In Proceedings of the 5th international symposium on Information Processing (FCIP 69)*, vol. A3, pp. 125–128, 1969.

[37] D. T. Pham and S. S. Dimov, "An algorithm for incremental inductive learning," *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, vol. 211, no. 3, pp. 239–249, 1997.

[38] P. Clark and T. Niblett, "The cn2 induction algorithm," *Machine learning*, vol. 3, no. 4, pp. 261–283, 1989.

[39] D. T. Pham, S Bigot, and S. S. Dimov, “Rules-f: A fuzzy inductive learning algorithm,” *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, vol. 220, no. 9, pp. 1433–1447, 2006.

[40] H. I. Mathkour, “Rules3-ext: Improvements of rules3 induction algorithm,” *Mathematical and Computational Applications*, vol. 15, no. 3, pp. 318–324, 2010.

[41] H ElGibreen and M. S. Aksoy, “Rules-tl: A simple and improved rules algorithm for incomplete and large data,” *Journal of Theoretical and Applied Information Technology*, vol. 47, no. 1, pp. 28–40, 2013.

[42] H. Elgibreen and M. S. Aksoy, “Rules-it: Incremental transfer learning with rules family,” *Frontiers of Computer Science*, vol. 8, no. 4, pp. 537–562, 2014.

[43] C. A. Murthy and N. Chowdhury, “In search of optimal clusters using genetic algorithms,” *Pattern Recognition Letters*, vol. 17, no. 8, pp. 825–832, 1996.

[44] S. Bandyopadhyay and U. Maulik, “An evolutionary technique based on k-means algorithm for optimal clustering in rn,” *Information Sciences*, vol. 146, no. 1-4, pp. 221–237, 2002.

[45] K Krishna and N. M. Murty, “Genetic k-means algorithm,” *IEEE Transactions on Systems Man And Cybernetics-Part B: Cybernetics*, vol. 29, no. 3, pp. 433–439, 1999.

[46] Y. Lu, S. Lu, F. Fotouhi, Y. Deng, and S. J. Brown, “Fgka: A fast genetic k-means clustering algorithm,” in *Proceedings of the 2004 ACM symposium on Applied computing*, ACM, 2004, pp. 622–623.

[47] H.-x. Guo, K.-j. Zhu, S.-w. Gao, and T. Liu, "An improved genetic k-means algorithm for optimal clustering," in *Sixth IEEE International Conference on Data Mining-Workshops (ICDMW'06)*, IEEE, 2006, pp. 793–797.

[48] J. Xiao, Y. Yan, J. Zhang, and Y. Tang, "A quantum-inspired genetic algorithm for k-means clustering," *Expert Systems with Applications*, vol. 37, no. 7, pp. 4966–4973, 2010.

[49] M. Laszlo and S. Mukherjee, "A genetic algorithm using hyper-quadtrees for low-dimensional k-means clustering," *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, no. 4, pp. 533–543, 2006.

[50] M. A. Rahman and M. Z. Islam, "A hybrid clustering technique combining a novel genetic algorithm with k-means," *Knowledge-Based Systems*, vol. 71, pp. 345–365, 2014.

[51] K. R. Žalik, "An efficient k-means clustering algorithm," *Pattern Recognition Letters*, vol. 29, no. 9, pp. 1385–1391, 2008.

[52] H. He and Y. Tan, "A two-stage genetic algorithm for automatic clustering," *Neurocomputing*, vol. 81, pp. 49–59, 2012.

[53] D. K. Roy and L. K. Sharma, "Genetic k-means clustering algorithm for mixed numeric and categorical data sets," *International Journal of Artificial Intelligence & Applications*, vol. 1, no. 2, pp. 23–28, 2010.

[54] I.-S. Oh, J.-S. Lee, and B.-R. Moon, "Hybrid genetic algorithms for feature selection," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 26, no. 11, pp. 1424–1437, 2004.

[55] J. Naruchitparames, M. H. Gunes, and S. J. Louis, "Friend recommendations in social networks using genetic algorithms and network topology," in *2011 IEEE Congress of Evolutionary Computation (CEC)*, IEEE, 2011, pp. 2207–2214.

[56] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of machine learning research*, vol. 3, no. Mar, pp. 1157–1182, 2003.

[57] B. Xue, M. Zhang, W. N. Browne, and X. Yao, "A survey on evolutionary computation approaches to feature selection," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 4, pp. 606–626, 2016.

[58] H. Kashyap, S. Das, J. Bhattacharjee, R. Halder, and S. Goswami, "Multi-objective genetic algorithm setup for feature subset selection in clustering," in *Recent Advances in Information Technology (RAIT), 2016 3rd International Conference on*, IEEE, 2016, pp. 243–247.

[59] R. Leardi and A. L. Gonzalez, "Genetic algorithms applied to feature selection in pls regression: How and when to use them," *Chemometrics and intelligent laboratory systems*, vol. 41, no. 2, pp. 195–207, 1998.

[60] S. Oreski and G. Oreski, "Genetic algorithm-based heuristic for feature selection in credit risk assessment," *Expert systems with applications*, vol. 41, no. 4, pp. 2052–2064, 2014.

[61] J. D. Kelly Jr and L. Davis, "A hybrid genetic algorithm for classification.," in *IJCAI*, vol. 91, 1991, pp. 645–650.

[62] F. Hussein, N. Kharma, and R. Ward, "Genetic algorithms for feature selection and weighting, a review and study," in *Document Analysis and Recognition, 2001. Proceedings. Sixth International Conference on*, IEEE, 2001, pp. 1240–1244.

[63] F. Gagnon and B. Esfandiari, "A hybrid approach to operating system discovery based on diagnosis," *International Journal of Network Management*, vol. 21, no. 2, pp. 106–119, 2011.

[64] P. Auffret, "Sinfp, unification of active and passive operating system fingerprinting," *Journal in computer virology*, vol. 6, no. 3, pp. 197–205, 2010.

[65] L. G. Greenwald and T. J. Thomas, "Toward undetected operating system fingerprinting.," *WOOT*, vol. 7, pp. 1–10, 2007.

[66] M. H. Gunes and K. Sarac, "Analyzing router responsiveness to active measurement probes," in *Passive and Active Network Measurement*, S. B. Moon, R. Teixeira, and S. Uhlig, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–32, ISBN: 978-3-642-00975-4.

[67] H. Kardes, M. H. Gunes, and K. Sarac, "Graph based induction of unresponsive routers in internet topologies," *Computer Networks*, vol. 81, pp. 178 –200, 2015, ISSN: 1389-1286.

[68] R. Beverly, "A robust classifier for passive tcp/ip fingerprinting," in *Passive and Active Network Measurement*, Springer, 2004, pp. 158–167.

[69] C. Mavrakis, "Passive asset discovery and operating system fingerprinting in industrial control system networks," Master's thesis, Technische Universiteit Eindhoven University of Technology, Oct. 2015.

[70] D. Chang, Q. Zhang, and X. Li, "Study on os fingerprinting and nat/tethering based on dns log analysis," *IRTF & ISOC Workshop on Research and Applications of Internet Measurements (RAIM)*, 2015.

[71] A. Ornaghi and M. Valleri, *Ettercap*, 2003. [Online]. Available: http://ettercap.sourceforge.net/index.php?s=home.

[72] F. Gagnon, B. Esfandiari, and L. Bertossi, "A hybrid approach to operating system discovery using answer set programming," in *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, IEEE, 2007, pp. 391–400.

[73] R. Reiter, "A theory of diagnosis from first principles," *Artificial intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

[74] G. F. Lyon, *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.

[75] K. Gao, C. Corbett, and R. Beyah, "A passive approach to wireless device fingerprinting," in *2010 IEEE/IFIP International Conference on Dependable Systems&Networks (DSN)*, IEEE, 2010, pp. 383–392.

[76] A. S. Uluagac, S. V. Radhakrishnan, C. Corbett, A. Baca, and R. Beyah, "A passive technique for fingerprinting wireless devices with wired-side observations," in *2013 IEEE conference on communications and network security (CNS)*, IEEE, 2013, pp. 305–313.

[77] M. Miettinen, S. Marchal, I. Hafeez, N Asokan, A.-R. Sadeghi, and S. Tarkoma, "Iot sentinel: Automated device-type identification for security enforcement in iot," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, IEEE, 2017, pp. 2177–2184.

[78] D. Chen, N. Zhang, Z. Qin, X. Mao, Z. Qin, X. Shen, and X.-Y. Li, "S2m: A lightweight acoustic fingerprints-based wireless device authentication protocol," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 88–100, 2016.

[79] Y. Sharaf-Dabbagh and W. Saad, "On the authentication of devices in the internet of things," in *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, IEEE, 2016, pp. 1–3.

[80] J. François, H. Abdelnur, R. State, and O. Festor, "Ptf: Passive temporal fingerprinting," in *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, IEEE, 2011, pp. 289–296.

[81] S. V. Radhakrishnan, A. S. Uluagac, and R. Beyah, "Gtid: A technique for physical device and device type fingerprinting," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 5, pp. 519–532, 2015.

[82] D. Formby, P. Srinivasan, A. Leonard, J. Rogers, and R. A. Beyah, "Who's in control of your control system? device fingerprinting for cyber-physical systems.," in *NDSS*, 2016.

[83] T. Kohno, A. Broido, and K. C. Claffy, “Remote physical device fingerprinting,” *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 2, pp. 93–108, 2005.

[84] L. C. C. Desmond, C. C. Yuan, T. C. Pheng, and R. S. Lee, “Identifying unique devices through wireless fingerprinting,” in *Proceedings of the first ACM conference on Wireless network security*, ACM, 2008, pp. 46–55.

[85] N. T. Nguyen, G. Zheng, Z. Han, and R. Zheng, “Device fingerprinting to enhance wireless security using nonparametric bayesian method,” in *INFOCOM, 2011 Proceedings IEEE*, IEEE, 2011, pp. 1404–1412.

[86] Q. Xu, R. Zheng, W. Saad, and Z. Han, “Device fingerprinting in wireless networks: Challenges and opportunities,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 94–104, 2016.

[87] H. Patel, “Non-parametric feature generation for rf-fingerprinting on zigbee devices,” in *2015 IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA)*, IEEE, 2015, pp. 1–5.

[88] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh, “Mobile device identification via sensor fingerprinting,” *arXiv preprint arXiv:1408.1416*, 2014.

[89] J. Quinlan, “Learning efficient classification procedures and their application to chess end games,” Symbolic Computation, R. Michalski, J. Carbonell, and T. Mitchell, Eds., pp. 463–482, 1983.

[90] R. Yasdi, “Learning classification rules from database in the context of knowledge acquisition and representation,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, no. 3, pp. 293–306, 1991.

[91] M. Pazzani, *UCI machine learning repository*, 1991. [Online]. Available: http://archive.ics.uci.edu/ml.

[92] B. F. Hayes-Roth, *UCI machine learning repository*, 1989. [Online]. Available: http://archive.ics.uci.edu/ml.

[93] G. Gong, *UCI machine learning repository*, 1988. [Online]. Available: http://archive.ics.uci.edu/ml.

[94] R. Fisher, *UCI machine learning repository*, 1988. [Online]. Available: http://archive.ics.uci.edu/ml.

[95] J. Cendrowska, *UCI machine learning repository*, 1990. [Online]. Available: http://archive.ics.uci.edu/ml.

[96] M. H. Gunes, S. Bilir, K. Sarac, and T. Korkmaz, "A measurement study on overhead distribution of value-added internet services," *Computer Networks*, vol. 51, no. 14, pp. 4153 –4173, 2007, ISSN: 1389-1286.

[97] M. A. Canbaz, K. Bakhshaliyev, and M. H. Gunes, "Router-level topologies of autonomous systems," in *Complex Networks IX*, S. Cornelius, K. Coronges, B. Gonçalves, R. Sinatra, and A. Vespignani, Eds., Cham: Springer International Publishing, 2018, pp. 243–257, ISBN: 978-3-319-73198-8.

[98] H. Kardes, M. Gunes, and T. Oz, "Cheleby: A subnet-level internet topology mapping system," in *2012 Fourth International Conference on Communication Systems and Networks (COMSNETS 2012)*, 2012, pp. 1–10.

[99] M. H. Gunes and K. Sarac, "Importance of ip alias resolution in sampling internet topologies," in *2007 IEEE Global Internet Symposium*, 2007, pp. 19–24.

[100] E. Arslan, M. Yuksel, and M. H. Gunes, "Network management game," in *2011 18th IEEE Workshop on Local Metropolitan Area Networks (LANMAN)*, 2011, pp. 1–6.

[101] E. Arslan, M. Yuksel, and M. H. Gunes, "Training network administrators in a game-like environment," *Journal of Network and Computer Applications*, 2015, ISSN: 1084-8045.

[102] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artificial intelligence*, vol. 97, no. 1, pp. 273–324, 1997.

[103] D. Michie, D. J. Spiegelhalter, and C. C. Taylor, "Machine learning, neural and statistical classification," 1994.

[104] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[105] A. Aksoy and M. H. Gunes, "Operating system identification using network packet headers," under revision, 2019.

[106] Y. Vanaubel, J.-J. Pansiot, P. Mérindol, and B. Donnet, "Network fingerprinting: Ttl-based router signatures," in *Proceedings of the 2013 conference on Internet measurement conference*, ACM, 2013, pp. 369–376.

[107] G. Lyon, *Nmap*. [Online]. Available: https://nmap.org.