THESIS

EVALUATION OF ANOMALY DETECTION APPROACHES USING SYSTEM CALL TRACES ON EMBEDDED LINUX SYSTEMS AND COMPARISON WITH PROCESS WHITELISTING APPROACHES

Submitted by
Payam Samimi

am Fachgebiet Agententechnologien in betrieblichen Anwendungen und der Telekommunikation (AOT)

In partial fulfillment of the requirements
For the Degree of **Master of Science**

Technische Universität Berlin
Spring 2018

Gutachter:
Prof. Dr.-Ing. habil. Sahin Albayrak
Prof. Dr. habil. Odej Kao

Payam Samimi
Matrikelnummer : 367099
Mollwitz Str. 3, 14059, Berlin

Siemens AG
Mobility Division
Mobility Management
IT Security

This thesis is originated in cooperation with IT Security department in Mobility Division of SIEMENS AG in Berlin.

I would like to thank Dipl. Alexander Markus Wischy Hernandez, as well as the entire IT-Security team in the Mobility division at SIEMENS for giving me the opportunity to carry out state-of-the-art research in this field.

Abstract

In recent years, embedded systems have become increasingly popular and progressively used in various domains. Modern cars, equipment in smart homes, cell phones and critical infrastructures are instances of domains that have increasingly employed these systems. Notwithstanding their diverse advantages, protecting these systems against security threats prompts many challenges. It is believed, that owing to the interactive operation of embedded systems with the real world, security attacks on these systems can cause physical and (in some cases like car crashes) irreversible after-effects, containing serious harm to persons, or even death.

Anomaly-based intrusion detection systems have been widely adopted in many domains as a countermeasure against security attacks. They can detect anomalies through observing the normal behaviour of the system. However, applying these systems in embedded applications prompts challenges such as handling the limited computation resources of embedded systems, as well as the high rate of false positives in anomaly detection systems.

This thesis primarily tackles building an anomaly detection system for embedded applications with considering the aforementioned challenges. First, it provides an overview of a classification of intrusion detection systems. Next, characteristics of embedded systems and requirements in building IDS for these systems are analyzed. Additionally, an overview of related approaches from other research studies is provided and their achieved results are compared.

In the interest of overcoming these challenges and satisfying the requirements, this thesis introduces a host anomaly-based intrusion detection system called Linux Intrusion Detection System (LIDS), which is divided into two sub-components and combines two different techniques of monitoring process behavior: application whitelisting (AWL) and system calls analysis. AWL are performed by the sub-component Watchdog on embedded devices and is implemented based on the Linux Security Module framework. The second component - Secure Homeland - operates on a server and performs system call trace analysis using classification-based data mining techniques.

This work discusses various data representation methods of system call traces and evaluates anomaly detection methods based on two of these representation techniques: sequence-based and pattern frequency-based. Accordingly, this thesis explains and uses the Australian Defence Force Academy Linux Dataset (ADFA-LD), which includes labeled system call traces for normal and abnormal behavior of the system. For the evaluation, Weka tool has been used, which is a set of machine learning algorithms for data mining tasks.

The results of this thesis show that the addressed approach in this work handles the computation constraints of embedded systems efficiently and improves detection accuracy.

Zusammenfassung

In den letzten Jahren kamen eingebettete Systeme zunehmend in unterschiedlichen Bereichen zum Einsatz. Als Beispiele sind selbstfahrende Kraftfahrzeuge, intelligente Haustechnik und kritische Infrastrukturen für den Anwendungsbereich solcher Systeme. Trotz der vielen Vorteile der Anwendung dieser Systeme ist deren Sicherung eine große Herausforderung. Insbesondere wegen ihrer direkten Interaktion mit Menschen können Sicherheitsangriffe in diesen Systemen irreparable Schäden verursachen.

Anomalieerkennungssysteme werden bereits in vielen Bereichen gegen Sicherheitsangriffe eingesetzt. Diese Systeme können durch Überwachung des normalen Verhaltens eines Systems Anomalien erkennen. Anomalieerkennungssysteme beanspruchen jedoch große Rechenkapazitäten, über die eingebettete Systeme nicht verfügen.

Aus diesem Grund liegt der Fokus dieser Abschlussarbeit auf der Entwicklung von Anomalieerkennungssytemen für eingebettete Systeme. In einem ersten Schritt werden die Anomalieerkennungssysteme klassifiziert. Anschließend werden wesentliche Merkmale von eingebetteten Systemen und die entstehenden Anforderungen bei der Entwicklung eines Anomalieerkennungssystems für diese Systeme analysiert.

Unter Berücksichtigung dieser Merkmale und Anforderungen untersucht diese Arbeit ein Anomalieerkennungssystem für eingebettete Anwendungen, das auf Basis der Prozessüberwachungsmethode zwei unterschiedliche Ansätze kombiniert: das Anwendungs-Whitelisting und die Analyse der Systemaufrufe unter Einsatz von maschinellem Lernen.
Das Anwendungs-Whitelisting wurde in Form des Linux Security Module Framework implementiert und funktioniert auf den Endgeräten. Die Systemaufrufanalyse wird hingegen auf einem Server durchgeführt.

Für die Durchführung dieser Analyse befasst sich diese Arbeit mit den bekanntesten Darstellungsmöglichkeiten der Systemaufrufsequenzen und evaluiert die Performanz der Algorithmen des maschinellen Lernens auf Basis zweier dieser Darstellungen: der Reihenfolge der Aufrufe und der Wiederholung der Systemaufrufe. Dementsprechend wurde im Rahmen dieser Arbeit der Australian-Defence-Force-Academy-Linux-Datensatz (ADFA-LD) verwendet, um die nötigen Datendarstellungen zu generieren.

Die im Rahmen dieser Arbeit ermittelten Ergebnisse beweisen, dass die Kombination der zwei Ansätze (Anwendungs-Whitelisting und Systemaufrufanalyse) eine effiziente Methode für den Einsatz der Annomalieerkennungssysteme in eingebetteten Anwendungen ist. Darüber hinaus wird mithilfe dieser Ansatzkombination die Erkennungsgenauigkeit erhöht.

Contents

List of Figures

List of Tables

1 Introduction

In recent years, embedded systems have become increasingly popular and progressively used in various domains. Modern cars, equipment in smart homes, cell phones and critical infrastructures are instances of domains that have increasingly employed these systems [4]. Dozens of electronic control units are used in modern cars to implement highly-intelligent driver assistant systems. Smart meters are employed in smart homes for measuring electricity consumption and provide information for utility servers. Critical domains [5] such as energy supply, transport and traffic are the sectors that likewise employ embedded systems. For instance, signals and track vacancy detection systems in German railways observe and signal whether line sections are clear [6].

Embedded systems enable cost-efficient implementation. They are advised when physical size and weight, long life-cycle, real-time functionality and reliability have to be considered thoughtfully during the development process [7]. Notwithstanding their diverse advantages, protecting these systems against security threats prompts many challenges. Over recent years, numerous research works have been published that have introduced vulnerabilities in many classes of embedded devices [4]. For instance, Peter Shipley and Simson L. Garfinkel discovered a vulnerable modem line that was connected to a system responsible for controlling high voltage power transmission lines [8]. According to Koopman [8], owing to the interactive operation of embedded systems with the real world, security attacks on these systems can cause physical and (in some cases, like car crashes) irreversible after-effects involving serious harm to persons, or even death. Moreover, employing embedded devices in internet-based applications has progressively increased, which consequently demands a lot of effort to secure these systems. He believes that connecting these systems to the internet reveals embedded applications to intrusions and malware attacks.
Over recent years, there has been an immense range of companies and organizations around the world that have been targeted by various cyberattacks. One of the first massive cyberattacks - called Stuxnet - was in Iran in 2010 against the Iranian nuclear program, which aimed at industrial control systems operating in the Natanz uranium enrichment plant [9]. Recently, the WannaCry ransomware attack targeted IT infrastructure in hospitals in the United Kingdom, which caused shutting down work at several hospitals [10], [11]. The German railroad company Deutsche Bahn AG became the latest high-profile casualty of the WannaCry attack. Consequently, these cyberattacks demonstrated a huge demand for developing security solutions for embedded applications. Furthermore, traditional security mechanisms considered for desktop computers may not fulfill the requirements of these systems [8].

Intrusion detection systems (IDS) have been widely adopted in many domains as a countermeasure against security attacks [12]. IDS may be software- and/or hardware-based, which monitors activities on a host system or in a network. IDS can detect malicious activities by comparing current system behaviour with a model of normal/abnormal behaviour of the system. Once an anomaly is detected, an alarm will be generated. Considering the safety-critical environment and operation of embedded systems, developing IDS for these systems is reasonable [4]. However, building and adapting these detection systems according to the characteristics of embedded applications is significantly challenging. In most cases, the operation of IDS demands high memory capacity and computing power, although, the majority of embedded devices have constraints on these resources.

1.1 Objective

The goal of this work is to achieve an approach to build an intrusion detection system for embedded systems that can handle their computing and memory constraints. Furthermore, this work will study the feasibility of various anomaly detection approaches and their potential detection accuracy.
This section will specify the basic skeleton structure of the IDS, considering the challenges and characteristics of embedded applications and IDS.

1.1.1 Challenges

This section focuses on the challenges in building intrusion detection systems for embedded applications. The challenges discussed in this work are based on [4] and [8]. It should be noted that the discussed challenges have been collected according to their priority from the author's perspective.

- Memory capacity: Limited memory capacity introduces a considerable challenge for building IDS. The majority of embedded devices are equipped with small memory capacity and thus most of the memory space is allocated for the system itself. On the other hand, IDS detects intrusions based on models of the system behavior (normal and/or anomaly), which are then loaded into the memory at run-time. This requires additional memory space for IDS. Therefore, finding an approach for intrusion detection despite the memory constraints is challenging [4].

- Overall system performance: Embedded devices are used in production to control processes. They interact with the real word. Disabling them or even causing delay can cause irreversible after-effects like financial loss due to factory downtime or even injuries [8]. Thus, IDS must not under any circumstances affect the performance of these systems [4].

- False positives (FP): Issuing *False positive* alarms is the mostly commonly-known challenge of IDS. FPs are normal patterns that are erroneously detected as anomalies. These alarms in an embedded environment are identified as a major challenge. The reason behind this challenge is that generally embedded devices operate in network-based applications. If each device in this network generates even a small rate of FPs, these alarms overload the utility server/s and the network completely [4].

1.1.2 Taxonomy of IDS

Designing an IDS requires thoughtful consideration of IDS characteristics. This section aims to explain the characteristics of IDS based on its classification according to [13][12][14][15][12]. IDS can be classified relying on five main concepts:

- *Audit source location*: IDS can be network- or host-based. Host-based intrusion detection systems (HIDS) analyze the characteristics of one machine to detect suspicious operations. Network-based intrusion detection systems (NIDS) observe and analyze the network traffic for detecting malicious activities. In the recent past, NIDSs have been subject to more research studies than HIDSs. The reason behind this is as follows: first, traditional HIDS methods (e.g. offline integrity examination, log analysis, etc.) cause adverse impacts on system performance; second, at present most applications are designed network-based; and third, the primary strength of NIDSs is their ability to have a global vision over the network, which enables them to detect distributed network attacks.
  However, researchers believe that this situation will change. They believe that deploying fast hardware enables researchers and developers to implement more complex methods in HIDSs. Moreover, HIDSs can issue precise alarms, including all information concerning anomalous processes.

- *Detection method, anomaly or misuse*: IDS may be *misuse-* or *anomaly-based*. *Misuse-based* detection systems use a database of attack vectors to detect intrusion. These detection systems compare the system behavior with that database. Once a match is detected, an alarm will be issued. However, these systems are intrinsically unable to recognize unknown attack vectors. Furthermore, they could generate a high rate of false positives (FP) when attack vectors are analogous to both normal and anomalous patterns. Additionally, in an embedded environment generating and updating attack vectors on a large number of embedded devices is another challenge.
  In contrast to *misuse-based* IDSs, *anomaly-based* ones make use of normal behavior of the system to build the detection model. The current system behavior is then compared with that model. Once deviation from normal behavior is recognized, an alarm will be issued. Modeling approaches vary from statistical methods to artificial intelligence techniques. The *anomaly-based* approaches overcome *misuse-based* methods in detecting novel attacks. Nevertheless, all changes in the system normal behavior require updating the model, otherwise changes may cause false alarms.

- *Behaviour on Detection*: Kruegel [13] classified IDS based on their behaviour on detection into *passive* and *active*. *Passive* IDS generates alarms, in contrast to *active* IDS, which takes decisions and responds to attacks; for instance, changing *firewall* rules.

- *Usage frequency*: IDS may detect anomalies in real-time or offline. Real-time IDSs are usually implemented as operating system hooks. These systems provide real-time detection, although implementing these hooks requires proper OS programming knowledge. By contrast, offline IDSs observe the system behavior periodically. While, their implementation is considerably simpler than OS hooks, they may affect the overall system performance during periodical checks. Moreover, determining the time period that could prevent fast attacks is challenging.

- *Architecture*: IDS may operate in three modes: *stand-alone, distributed or hierarchical*. Bukac et.al [14] published a survey on *stand-alone* IDSs. *Stand-alone* IDSs observe the system behavior locally. These systems do not share alarms or information with other intrusion detection systems. By contrast, *distributed* IDSs collaborate on intrusion detection. Each IDS is considered as an agent and cooperates with other agents distributed over the network. While agents share information and alarms between each other over the network, this may cause a rapid rise in network traffic, and consequently network overloading. A survey on collaborating IDSs has been published in [16]. The last group is *hierarchical* IDSes. These systems operate similarly to *distributed* IDSs, although the cooperation between agents is layered. In this fashion, each agent is allowed to cooperate with specified agents in its cluster. Although this architecture may avoid the network overload, these systems do not have global vision over the network, and thus they are unable to detect distributed network attacks.

1.1.3 Contribution

In order to overcome the aforementioned challenges and based on IDS taxonomies, this section characterizes the IDS that will be developed in this work as follows:

- ***Audit source location - host-based***: As previously discussed, host-based IDS are increasingly becoming the target of research studies and the industry. This is due to their capability in terms of issuing precise alarms and facilitating network attack detection. Moreover, in many applications such as in-vehicle systems, analyzing the system behavior can only be performed on the host. Thereby, this work will focus on a host-based intrusion detection system (HIDS) for being as part of a current flow of research. This HIDS will operate on a Linux-based distribution "debian", which has a large market share in embedded applications. Thus, throughout this work it will be called "Linux Intrusion Detection System (LIDS)".

- ***Detection method - anomaly-based***: Embedded systems are largely designed to perform specific operations that have a static nature. For instance, they accept and process data packets from determined IP addresses, or they are allowed to run a list of known processes. Moreover, there is no straight interaction between humans and these systems; rather, they are part of a network of devices, or they communicate to server(s) [4]. Considering this static nature of tasks in embedded environments, building the intrusion detection model based on normal behavior is more reasonable than generating a database of attack vectors. Thus, this work will propose an anomaly-based IDS. Host-based IDSs can use various methods or a combination of them for intrusion detection, including monitoring process behavior, an integrity check of system files, monitoring network packets, and log data analysis [14]. The *LIDS* will focus on monitoring

process behaviour for intrusion detection. Details concerning the advantages and disadvantages of these approaches are discussed in Section 2.
Techniques for process observation vary from rather straightforward methods, e.g. *Whitelisting of examined applications, CPU and memory consumption measurements*, *file access monitoring*, to fully complex approach, e.g. *monitoring of system call traces*. The more complex is the method, the greater the computing resources required [14].
For the purpose of achieving a trade-off between method complexity and computing constraints, LIDS will be divided into two separate sub-components. The first component, called Watchdog - which operates on the host embedded device - takes responsibility for process whitelisting.
The second component, called Secure Homeland - which operates on another machine with significantly high computing power - performs complex analyses of system call traces. The system call traces are generated by devices and collected by a server periodically. This collected audit data will then be analyzed by Secure Homeland using data mining approaches.

- ***Behaviour on detection - passive***: LIDS is classified as a passive host anomaly-based IDS, given that the majority of embedded devices carry out time-critical tasks. Interrupting these tasks would cause safety-critical impacts on that domain. The alarms issued by LIDS will be gathered by a utility server.

- ***Usage frequency - real-time and offline***: This work will profit from both observation modes. Watchdog will observe processes in real-time, while Secure Homeland analyzes the system call traces periodically.

- ***Architecture - stand-alone***: LIDS can be classified as a stand-alone intrusion detection system. The both sub-components of the LIDS - Watchdog and Secure Homeland - will not share the information concerning the monitored devices as well as the issued alarms.

In conclusion of this chapter, this work will focus on building a passive host-based anomaly detection system that observes process behavior based on whitelisting and data mining techniques. Moreover, this thesis will evaluate various anomaly detection approaches.

1.2 Outline

This thesis is separated into seven chapters.

Chapter 2 discusses fundamentals and the state of the art regarding traditional detection methods of host-based intrusion detection systems. Moreover, Australian Defence Force Academy data sets (ADFA) [17] that are used to evaluate IDS are described in this chapter. Furthermore, anomaly modeling techniques are explained here.

Chapter 3 explains the requirements for building a host-based intrusion detection system for embedded devices. This chapter analyzes requirements arising from embedded applications, intrusion detection systems and software engineering.

Chapter 4 demonstrates a high-level description of the architectural structure of the selected approach. Additionally, the single components of the proposed solution are explained in this chapter using images and UML diagrams.

Chapter 5 describes the implementation part of this work. In this chapter, important aspects of implementation including the development environment and software used are explained. In addition, the most significant parts of code used in the implementation are emphasized.

Chapter 6 illustrates methods used for evaluating the proposed solution. Moreover, this chapter includes the results of the evaluation process.

Chapter 7 summarizes the thesis and describes the problems that occurred during this work. Furthermore, it provides an outlook about possible future works.

2 Related work

In this chapter, fundamentals and related research works are discussed, including techniques for host-based intrusion detection, the data set for evaluating IDS and anomaly modeling techniques. This section considers study works that have been published recently and proposed techniques with high detection rates.

2.1 Detection methods

Traditional host-based intrusion detection systems rely on three main methods: enforcing the integrity of critical system files, monitoring leaving and entering network packets, and monitoring the behavior of processes [14]. Integrity checking HIDS makes use of cryptographic hash functions - both in keyed and unkeyed fashion - to calculate hash values of critical system files. Typical instances of hash functions include message-digest algorithm 5 (MD5), secure hash algorithm (SHA) 0/1/2/3, keyed-hash message authentication code (HMAC), etc.

Fundamentally, in this approach a list of hash values of the system files is stored in a database. HIDS uses this database during the control process, in which a new hash value is generated over a system file and then compared with the anticipated value stored in the list. If a matching is found, the file is ignored, otherwise HIDS issues an alarm. This control process may be performed in real-time or periodically. In real-time mode, HIDS uses so-called kernel hooks. More precisely, HIDS implements those kernel hooks, which enables HIDS accessing data structures belonging to the operating system. Accordingly, HIDS can check the integrity once a system file is requested by a program. In this respect, Patil et al. [18] introduced a real-time integrity controller - namely I3FS - which checks the integrity of each file before it is provided for an application. I3FS is a loadable kernel module, which requires security policies that can be dynamically edited by system administrators. A further real-time detection system proposed by Kaczmarek and Wrobel [19] is ICAR (Integrity Checking and Restoring), which is implemented as a loadable kernel module. Once an application requests a file, the integrity of that file is checked. Under conditions where checking fails, the file is restored from back-up, and consequently the anomalous event is logged by ICAR. In contrast to real-time integrity checking, in periodical mode the hash values of system files are calculated periodically and compared with the expected values stored in the list. Tripwire is a periodical integrity checking mechanism that provides detecting of unauthorized alterations [20]. Tripwire uses the security policy for files and carries out scheduled integrity controls relying on a checksum analogy. A further periodical checker is Osiris, which monitors the integrity of one or multiple hosts [20]. Osiris issues logs concerning modifications in the file system, user and group lists, resident kernel modules, etc. These logged events

can also be sent to the administrator.
Nevertheless, it is believed that both approaches - real-time and periodical - have disadvantages [14]. In a real-time detection approach, accessing the list from a second storage can affect the overall system performance. This can cause disabling embedded devices of performing time-critical tasks. In addition, implementing the kernel hooks requires high skills in the operating system programming field. Furthermore, any updates in the kernel version requires adapting those changes in the implementation of the hooks. On the other hand, the periodical checking approach suffers from the complication of determining the time period for checking. For instance, if HIDS controls hash values every five minutes, faster attacks that occur in a few seconds remain undiscovered, or they can be detected after they have caused damages [8]. Finally, apart from using the real-time or periodical approach, the main disadvantage of integrity checking systems is their incapability in detecting network-based attacks. For instance, during a javascript-based attack the hash value of the browser's binary will not change. Thus, implementing HIDS relying on integrity checking requires thoughtful consideration of various aspects.

HIDS based on network traffic analysis observes entering and leaving packets for clues of undesired traffic, which is discussed in this work according to Bukac et al. [14]. In comparison to NIDS, whereby data encryption affects their capabilities, HIDS is capable of observing packets on the application layer (having access to decrypted packet contents). This enables HIDS to carry out more specific detection techniques such as deep packet analysis of the network traffic. In addition, HIDS can facilitate recognizing Botnets by observing the behavior of the host in the network. Barbhuiya et al. [21] proposed a detection system based on network traffic analysis, an active probing mechanism that mitigates three ARP-based attack vectors: the ARP spoofing, malformed ARP packets and the ARP denial-of-service attack in local networks. For instance, against the ARP spoofing attack, this mechanism verifies each received ARP request and response through broadcasted validation requests. In case of already-authenticated IP and MAC addresses, ARP packets are processed. By contrast, for a simple validation process all hosts are requested and responses are gathered. The spoofed ARP packets (nonuniform responses) are then detected by comparing ARP responses. A further detection method based on network traffic have been proposed by Laurens et al. called DDoSniffer, which aims to detect leaving TCP SYN denial-of-service attacks. In particular, its detection method relies on parsing the leaving TCP packets. Once a new TCP connection begins, a new entry is generated in the Newconn and the Conn tables. In case of fully-established TCP connection (a packet counter of a specific connection is not greater than four), the appropriate record is erased from the Newconn. DDoSniffer issues an alarm under two circumstances: first, if the size of the Newconn table is greater than a predetermined threshold; and second, if the number of TCP leaving packets without incoming TCP acknowledgments exceeds a predefined quota.
Nevertheless, applying this network-based HIDS for embedded applications encounters disadvantages. For instance, due to the lack of computation resources in embedded systems, performing high complex methods for analyzing a huge number of network packets is not possible.

A further strategy for HIDS is to monitor process behavior. This may involve monitoring one specific process or multiple processes. Monitoring techniques vary from straightforward methods (e.g. whitelisting, CPU and memory consumption analysis, file access patterns) to rather complicated techniques (e.g., analyzing system call traces) [14]. Generally, whitelisting of trustworthy applications is advised for systems with constraints on computation resources. This is due to achieving a balance between method complexity and the availability of computation resources [14]. Hizver et al. [22] proposed a cloud-based application whitelisting system that proves binary files or libraries versus a whitelist, as soon as a program requests an executable file/library. If checking fails, the execution process is aborted.

Generally, depending on the observation frequency, detection techniques relying on process behavior monitoring can be divided into two main categories: real-time or offline mode. Offline mode analyzes the process behavior in a sandbox environment. By contrast, real-time monitoring is implemented as kernel hooks, which enables HIDS to detect intrusions as soon as the process behavior changes. The idea of allowing the kernel to observe itself grew when Forrest et al. [23] introduced a building detection model relying on normal behavior in 1996. Indeed, malicious activities are detected when the behavior of process varies from the expected normal behavior. In this respect, Reeves et al. [24] proposed an IDS called *Autoscopy* for resource-constrained embedded control systems in the power grid. It has been implemented in the form of a loadable kernel module, providing the detection of process behavior changes. *Autoscopy* makes use of *kprobe* (tracing framework) to monitor process behavior. Probes are placed in specific addresses in the kernel to collect all control-flow information under normal system operation. This information represents the detection model based on normal behavior. In the testing phase, current control-flow information is verified versus the detection model.

In the recent past, applying complex methods such as analyzing system call traces for detecting malicious activities has been targeted by many research studies. Each program according to its code is characterized with a set of system call traces [23]. These traces are generated during the execution of programs. Thus, collecting these traces of a running program during normal operation can be used to represents the model of normal behavior of that program. Accordingly, any variation of this model or system call order is detected as malicious activity. Representation methods of these traces can be categorized into two main groups [25]: *sequence-based* and *pattern frequency-based*. Sequence-based techniques consider a short sequence of normal system call traces for establishing the detection model. Forrest et al. [23], [26] introduced a contribution based on artificial immune system, in which normal system call traces in a running process are divided into shorter sequences with a fixed length, called windows. Let us consider a trace t, and a fixed-length k, then t is divided into system call windows as : $w_0, w_1, w_2, ...$ (w denotes a system call window). Consequently, those windows are gathered in a database, which represents the normal behavior. In the testing phase, each test trace is also divided into fixed-size windows, which are then compared with the database of normal windows. Test windows that exceed a predetermined variation threshold are detected as anomalies. Let us consider the following system call trace and the fixed-length k=3:

open, read, remap, remap, open, getrlimit, remap, close, exit

According to the window-based approach, this trace is divided into windows with a length of k+1=4 as follows:

$$\begin{gathered}w_0 = \text{open, read, remap, remap}\\ w_1 = \text{read, remap, remap, open}\\ w_2 = \text{remap, remap, open, getrlimit}\\ w_3 = \text{remap, open, getrlimit, remap}\\ w_4 = \text{open, getrlimit, remap, close}\\ w_5 = \text{getrlimit, remap, close, exit}\end{gathered}$$

Various methods have been proposed for determining an anomaly score of each test window. This anomaly score is then compared with the determined variation threshold for detecting a window as an anomaly or normal behavior. For instance, an anomaly score calculation approach discussed in [27] divides training traces into k-length windows and preserves the repetitiveness of happening of each unique window in training traces in a dictionary. This dictionary represents the normal behavior of the system. During testing, each test trace is also divided into k-length sequences. Each test sequence is allocated a probability grade $L(w_i)$, which corresponds with the repetitiveness calculated for each window in the training phase. In case $L(w_i)$ is larger than a predefined threshold λ, the window is normal ($L(w_i) > \lambda$), otherwise it is detected as an anomaly ($L(w_i) < \lambda$). Consequently, the anomaly grade for the entire trace is calculated in 2.1, where $\sum L(w_i)$ is number of anomalous windows in the entire trace, and l_t is the length of trace:

$$L(t_i) = \frac{\sum L(w_i)}{l_t} \tag{2.1}$$

The previously-discussed window-based technique has been widely used in various research studies for system call intrusion detection and is called threshold-based Sequence Time Delay Embedding (t-STIDE) [27] [23] [28] [29]. These studies differ in their anomaly score calculation.
Beside t-STIDE, statistical learning methods have shown considerable capabilities in handling short sequences [25]. The main goal of these methods is to obtain invisible linkages behind normal sequences for building a statistical model. Typical instances involve Artificial Neural Networks (ANN) [30], semantic data mining [31], Support Vector Machine (SVM) [32], and Hidden Markov Model (HMM) [33]. The significant advantage of statistical methods is their ability to create a precise model of normal behavior, although the required time for learning is immense [25].
Parallel to anomaly score techniques, several researchers have used classification-based methods to label each trace as an anomalous or normal trace. For instance, in semi-supervised techniques (i.e. training traces cover normal behavior and test sequences contain normal and anomalous patterns), all training instances are labeled as normal. The classifier is then built by applying on those normal instances. Subsequently, in the testing phase each test sequence is analyzed by the classifier. If the test sequence is covered, it is labeled as a normal instance, otherwise it is labeled as anomaly. Typical examples of classifier-based techniques include HMM-based [34], neural networks [35], SVM [36] and rule-based [37].

The prime strength of window-based techniques is detecting anomalies localized in short sequences. However, apart from analyzing statistical or classification-based methods, if anomalies are distributed over the whole trace they are not discoverable by window-based techniques [27].
In contrast to window-based techniques, *pattern frequency-based techniques* have demonstrated computationally-efficient learning and testing results [27]. It should be noted that we consider this approach in summary, which is discussed in more details in [27].
Fundamentally, in this approach a pattern α is detected anomalous when its repetition in the given test trace t varies considerably from its anticipated frequency in a set of normal traces S. Essentially, this problem can be solved by allocating an anomaly score to the pattern α. Accordingly, the anomaly score for the pattern α is calculated based on the variation between the frequency of happening of α in the given test trace t, and the average repetition of occurrence of α in the training set S. More precisely, assuming $f_S(\alpha)$ as the average repetition of occurrence of α in S, and $f_t(\alpha)$ as frequency of happening of α in t, then these frequencies are normalized based on the length of training trace l_s, and test trace l_t, which generate the two following quantities 2.2 and 2.3

$$\bar{f}_t(\alpha) = \frac{f_t(\alpha)}{l_t} \tag{2.2}$$

$$\bar{f}_S(\alpha) = \frac{1}{|S|} \sum_{\forall s_i \in S} \frac{f_{si}(\alpha)}{l_{si}} \tag{2.3}$$

Finally, the anomaly grade of the pattern α results as 2.4:

$$A(\alpha) = |\bar{f}_t(\alpha) - \bar{f}_S(\alpha)| \tag{2.4}$$

Further to this basic *pattern frequency-based* technique, several variations of considering frequency in system call traces have been focused by researchers, such as vector-based methods. In vector-based methods, system call traces are represented in the form of fixed-length vectors. The length of vectors depends on the number of system calls in an operating system (e.g. the Linux distribution Ubuntu has about 342 system calls, then the vector for Ubuntu should have 342 indexes). Each index in the vector corresponds to the same number of a system call; for instance, index 23 characterizes the system call 23.
Borisaniya et al. [3] discussed three various vector-based data representation techniques that are used for building a classifier for labeling test traces: Boolean model, vector space model, and modified vector space model with *n-gram*.
A Boolean model is a simple representation of system call traces in the form of Boolean vector. Let us consider the number of system calls n, a system call trace t, then the vector V for the given system call trace t is shown as $V = \{b_0, b_1, b_2, ..., b_n\}$, where $b_i (0 \leq i \leq n)$ may be 1, if the system call i exists in the trace, otherwise 0.
Accordingly, the training system call traces are converted to Boolean vectors for building the classifier. In the testing phase, the given test trace is first transformed into a Boolean vector and then labeled by the classifier.
A vector space model is another data representation method for system call traces. It is also

called a bag of words method, in which each word is allocated a weight that determines a relativity of that word to a document. Considering this technique in system call representation, similar to a Boolean model a feature vector is issued for each system call trace. Each vector index corresponds to a system call number. Assuming the number of system calls n, a system call trace t, then the vector V of the given trace t is represented as $V = \{m_0, m_1, m_2, ..., m_n\}$, where $m_i (1 \leq i \leq n)$ defines the frequency of occurrence of the system call S_i in the trace t. The training and testing phases are similar to the Boolean Model.
The main weakness of the Boolean model and the vector space model is their incapability of regarding the order of system calls. In case an intruder is able to mimic the system call frequency in a malware, inadequacy of information regarding system call order could mean that the malware remains undetected [38], [39]. For instance, the feature vector for the two following system call traces are equal.

S_1 : *open, read, mmap, getrlimit, close, exit*
S_2 : *open, exit, read, mmap, getrlimit, close*

In order to handle this issue, Borisaniya et al. [3] proposed the *n-gram* with a vector space model to consider both aspects. Accordingly, a short sequence of system calls (with a fix-length) are assumed as a word. For instance, if the length is three, then each word comprises three system calls. Let us consider l as the size of the word, and n as the number of uniform system calls, then this method generates n^l amount of available uniform words in a vector.
In order to present the system call traces based on this method, the vector C is assumed as $C = \{m_1, m_2, m_3, ..., m_r\}$, where $r = n^l$, and $m_i (1 \leq i \leq r)$ defines the times that each word in the given trace occurs.
The performance of this algorithm is described as $O(N * |U^l|)$, which in case of system call traces is a high value. Borisaniya et al. [3] enhanced this method by eliminating dimensions of vectors, which are zero (absence of word in a trace).
Frequency-based approaches, which calculate an anomaly score for patterns, suffer from two main issues [27]. First, computational complexity is an issue given that the time required to compute anomaly grade for a pattern increases linearly according to the length of trace t and the length and count of patterns in s. For instance, if an anomaly grade should be calculated for all neighboring sub-sequences happening in a trace t, the amount of time required is extremely high. Second, the anomaly grade for pattern is a further issue, given that these techniques allocate an anomaly score for each pattern, although they do not declare a pattern as an anomaly or normal pattern. In addition, determining a threshold is challenging in most cases.
A further issue with all frequency-based methods is that they should wait for the full execution of a program to build their vectors. These methods need the entire trace to compute frequencies and anomaly score for patterns. Consequently, attacks like buffer-overflow may be detected after an attack has occurred. Table 2.1 and 2.2 show a list of research works with their detection methods.

Table 2.1: Research works concerning various detection methods

Detection Method	Research Works
Real-time Integrity Checking	I3FS, checking integrity of each file before it is provided for an application [18]
Real-time Integrity Checking	ICAR, a loadable kernel module which checks file integrity before an application access it [19]
Periodical Integrity Checking	Tripwire, provides integrity checking using the security policy for system file [20].
Periodical Integrity Checking	Osiris, multiple host monitoring tool, capable of logging modifications to file system, user and group lists, resident kernel modules, and etc.
Network Traffic Analysis	detecting ARP Spoofing based on request/response confirmation [21]
Network Traffic Analysis	Detecting TCP SYN denial-of-service attack [14]

Table 2.2: Research works concerning various process monitoring detection methods

Detection Method	Research Works
Whitelisting	Whitelisting of examined applications [22]
Observation of system behavior	Observing system behavior using the kernel debugging frameworks
Short sequence of system call traces	Forest et al. [26]
Short sequence of system call traces	statistical learning method based on t-STIDE [25]
Short sequence of system call traces	Window-based statistical learning method based on t-STIDE [25], ANN [30], semantic data mining [31], SVM [25], HMM [33]
Short sequence of system call traces	Classification-based approaches HMM [40], ANN [35], SVM [36], rule-based [37]
Frequency-based	Boolean Model [3]
Frequency-based	Vector Space Model Model [3]
Frequency-based	Modified Vector Space Model based on *n-gram* [3]

2.2 Dataset

The evaluation of intrusion detection systems using a dataset is widely used among researchers. The well-known data sets for HIDS include the data set of University of New Mexico (UNM) and the KDD data set used in the Third International Knowledge Discovery and Data Mining Tools Competition. These data sets have been commonly used for training and testing the intrusion detection engines. However, these data sets cover behaviors of operating systems from decades ago. Over time, various new features have been added to operating systems, which has radically changed some of their behaviors. For instance, changing from 32-bit to 64-bit architecture has affected the exploitability of these systems [17]. In 2013, a new generation of data sets including system call traces called Australian Defence Force Academy data sets (ADFA) was introduced [17]. ADFA covers two OS: ADFA-LD (representing Linux Datset) and ADFA-WD (representing Windows Dataset).
The ADFA-LD is collected under completely-patched Ubuntu version 11.04 based on the kernel V2.6.38. A Unix based tool, Auditd [17], is used to collect system call traces for ADFA-LD. In order to consider different attack vectors, various services were installed including a web server (Apache V2.2.17 running PHP V5.3.5), FTP server, database server (MySQL V14.14) and SSH server.
The ADFA-WD dataset includes system calls and DLL requests, which are gathered using the Procmon [41] program under the Windows XP SP2 operating system. This dataset is col-

lected under circumstances whereby the default firewall was activated and Norton AV 2013 was running. In addition, the following services were enabled: file sharing, printing over network, web server, database server, FTP server, streaming media server, PDF reader, etc.
The system call traces in both data sets - ADFA-LD and ADFA-WD - are divided into three groups: training data, validation data and attack data. Training data and validation data both represent the normal traces. By contrast, attack data contains raw abnormal traces. Moreover, attack traces are classified based on the attack methods into six classes covering Hydra-FTP, Adduser, Java-Meterpreter, Meterpreter, and Webshell. Tables 2.3 and 2.4 demonstrate these data sets in detail.

Table 2.3: ADFA data sets in detail [3]

data set	**ADFA-LD**		**ADFA-WD**	
	Traces	**System Calls**	**Traces**	**System Calls**
Training Data	833	308.077	355	13.504.419
Validation Data	4.372	2.122.085	1.827	117.918.735
Attack Data	746	317.388	5.542	74.202.804
Total	**5.951**	**2.747.550**	**7.724**	**25.625.958**

Table 2.4: Attack vectors in ADFA-LD data set [3]

Attack	**Impact**	**Vector**	**Number of traces**
Hydra-FTP	brute force password	FTP by Hydra	162
Hydra-SSH	brute force password	SSH by Hydra	176
Adduser	adds new superuser	client side poisoned executable	91
Java-Meterpreter	meterpreter with java	Tiki Wiki vulnerability exploit	124
Meterpreter	Linux meterpreter payload	client side poisoned executable	75
Webshell	C100 webshell	PHP remote file inclusion	118

According to Aghaie [42], many studies to date have employed these data sets for evaluating anomaly- and misuse-based detection systems. Xie et al. [25] applied one-class SVM (based on a short sequence data representation method) on the ADFA-LD (Linux-based) data set, which achieved an accuracy of 80% and 15% *False Positives*. Chang et al. [43] reported very poor performance of K-Nearest Neighbors and k-Means clustering methods based on a frequency representation technique. More precisely, they reported 50% detection accuracy and 20% FP on average among their three studies. Borisaniya et al. [3] applied a modified vector space model with *n-gram* to build classifiers using *Instance-based* learning with parameter K (IBK), and J48 decision tree. According to the study, they achieved an accuracy of 92% and 20% FP for multiclass, and 96% and 19% for the binary class. Consequently, the most of research works showed a poor performance of different data mining methods on these data set.
The ADFA data sets have introduced a new benchmark in evaluating host-based intrusion detection systems, although from the author's perspective they suffer from three main issues. First, system call traces have not been recorded in combination with their issuer program. It could possibly enhance detection accuracy if the programs together with their traces could

have been recorded; Second, no time information concerning traces as well as the period of time, during which data sets have been gathered, is available. Considering time in many cases could enhance detection rate. For instance, considering that a specific trace should not happen during nighttime or the weekend; Third, the system call traces in the ADFA-LD data set have been gathered using a Unix based tool Auditd [17], although no information concerning the configuration of this tool has been published.

2.3 Anomaly modeling techniques

This section explains various anomaly modeling techniques. Chen et al. [44] classified anomaly modeling techniques as follows:

2.3.1 Statistical models

Statistical techniques detect anomalous instances according to probability regions, which are generated by stochastic models. While anomalies belong to low-probability regions, normal instances appear in high-probability regions. In particular, a statistical model is built based on normal behavior. This model is then used in a testing phase, in which a probability for each test instance is computed. Normal instances mostly have high probability, although anomalies have low probabilities [45]. Typical instances of statistical methods or pattern recognition techniques are kNN, Bayesian Classifiers, and SVM [42].

2.3.2 Immune system approach

In host-based intrusion detection systems, an immune system approach models application-behavior based on system call traces. Forrest et al. [23] proposed analyzing system call paths during the normal execution of a program as a very consistent resource for building detection models. This technique is called the STIDE algorithm. This approach divides normal system call traces into fixed-length short sequences and gathers them in a database, which represents a normal model. Test traces are first divided into the fix-length short sequences, then compared with the database. If a short sequence of the test instance is not covered by this database, the instance is detected as an anomaly. Hofmeyr et al. [46],[47] proposed variation of this method, which uses the Hamming distance to detect anomalies. Accordingly, the Hamming distance between two sequences is computed to detect the variation degree.

2.3.3 Markov process model

Markov models are popular to mimicry system behavior based on state transitions. These techniques consider the state transitions for system calls individually. Anomalies differ from normal instances based on the probability of events, whereby this probability is determined relying on preceding state and allocated values in the state transition matrix. Anomalous instances have low probabilities, whereas normal instances have high probabilities. Chandola et al [27] proposed a survey of studies that applied variants of the basic Makrov model such as fixed markovian, varaible markovian, sparce markovian, hidden markov model [40], and etc. Although these techniques have shown high detection accuracy rates, the learning phase could take days [47].

2.3.4 Data mining techniques

In the recent past, data mining has been widely employed in intrusion detection systems [48]. Data mining techniques help to realize summarizing a large amount of data in a systematic manner [49]. Classification is one of the most commonly-used data mining methods. The prime strength of classification-based methods is their fast test phase in comparison to other data mining methods such as clustering-based approaches [45]. In the training stage, the labeled training dataset is used to build the classifier. In the testing stage, the classifier is applied to classify testing instances. Typical examples of classification-based techniques include neural networks, Bayesian networks, support vector machines (SVM), decision trees and rule-based methods [45] [49].

2.4 Summary

Throughout this chapter, the fundamentals and state-of-the-art concerning building host-based intrusion detection have been discussed. We considered detection methods of HIDS precisely, including integrity checking, network-based HIDS and process behaviour monitoring. Moreover, the recently-generated ADFA data sets have been introduced, which have been used by many recent research works for testing HIDSs. Finally, we have discussed various anomaly modeling techniques covering statistical models, immune system approaches, Markov process models and data mining techniques. Consequently, to the best of our knowledge, no HIDS has been introduced that besides considering characteristics of embedded systems can observe process behaviors on these devices.

3 Requirements

This chapter discusses the fundamental requirements for LIDS. The author strongly believes that building IDS for embedded applications requires considering a combination of requirements coming from software engineering, intrusion detection systems, and embedded systems.
In the recent past, researchers have divided system requirements into two main classes, namely non-functional and functional requirements. Non-functional requirements demonstrate a high-level view of how a system should operate. Functional requirements determine the processes in a system. According to this classification, this chapter has been divided into two sections, in which these requirements are discussed separately.

3.1 Non-functional requirements

The aim of this section is to discuss non-functional requirements for LIDS. These requirements have been regarded from the standpoint of embedded systems, software engineering and intrusion detection systems, with the help of several research works [50][51][13][52]. Table 3.1 summarizes these requirements according to their objectives and rationales.

Availability

In the author's view, in case of embedded systems this requirement has a high priority, given that availability can be considered from two various angles: first, regarding the availability of embedded systems; and second, the availability of the LIDS itself.
In most domains, embedded systems have a high ratio of operation time (almost 24 hours), in which critical tasks should be carried out. Correspondingly, the failure of these systems could cause significant damages in those domains.
Perceiving this requirement from the standpoint of the availability of LIDS itself, determining the proportion of time for which LIDS should operate is inevitable. More precisely, it is needed to consider the number of intrusions and the size of damages when LIDS is down during the operation time of the system.
Requirement: The LIDS shall under no circumstances cause system failure. Moreover, the LIDS shall be available 99.99% alongside the embedded device.

Accuracy

Accuracy defines that IDS should not detect normal behavior of the system as malicious activity (normal activity classified as an intrusion is called a *false positive (FP)*) [13]. From

the author's view, regarding this requirement in embedded applications is much more important than other applications such as desktop computers. The reason behind this is the network-based architecture of embedded applications. Typically, embedded systems are utilized in distributed applications. In most cases, these systems communicate with the utility server(s), which collects the information from devices and monitors their operation. Based on the assumption that HIDS operating on each individual device would generate a small rate of FPs, this could certainly overload the network or clog up the utility server. Thus, the aggressiveness of LIDS should be carefully considered.
Requirement: The LIDS shall issue false positive alarms with a probability below 0.01%

Completeness

This determines the capability of IDS in detecting all intrusions occurring in a system (undetected malicious activities are called *false negative (FN)*) [13]. Kruegel et al, [13] believe that satisfying this requirement is relatively challenging, due to the impossibility of having knowledge of past, present and future attacks. However, regarding the critical nature of tasks in embedded applications emphasizes the significance of considering this requirement. More precisely, uncovered intrusion can cause safety-critical impacts on this system.
Requirement: The LIDS shall detect intrusions with a probability of 99.99%.

Performance

This requirement can be divided into four sub-requirements: *time to detect intrusions*, *stored knowledge about intrusion/normal behaviour*, *time to response*, and *identity trace back analysis* [52].
First, time to detect intrusions defines the time between the occurrence of intrusions and their detection. The detection time mainly varies depending on the observation frequency of IDS. In offline-based detection systems, the detection time may be approximately days, weeks or possibly never. By contrast, this time in online-based detection systems is anticipated to be about minutes or hours. Consequently, the amount of damages to the system can be reduced. Accordingly, this requirement has an extremely high priority for LIDS due to the critical nature of the operation environment in embedded applications.
Requirement: The LIDS shall detect intrusions below one second.
Second, *stored knowledge about intrusion/normal behaviour* determines the completeness and expandability of IDS knowledge. The incompleteness of IDS knowledge can strongly affect IDS performance. Anomaly-based IDS requires knowledge including all patterns of normal behavior. If a detection model does not include entirely normal behavior, uncovered normal patterns can cause detecting the normal behavior of a system as malicious activity (*FPs*). By contrast, misuse-based IDS requires exhaustive knowledge concerning intrusions. Uncovered attack vectors can mean that intrusions remain undiscovered (*FNs*).
In addition to completeness, the knowledge of IDS should be updated as soon as changes

in normal behaviour occur (anomaly-based), or new attack vectors are detected (misuse-based). The result of neglected updating of even small changes in normal behavior of the system or new attack methods is having an incomplete knowledge, which causes issuing *FPs* or *FNs*.
Nevertheless, from the author's standpoint, in embedded applications in respect with the completeness and expandability of knowledge, its size and updating frequency should also be taken into account.
Generally, embedded systems have limited memory capacity, which limits increasing the size of knowledge. In addition, the updating frequency of the IDS knowledge on thousands of embedded devices distributed over the network demands careful attention.
Requirement: The stored knowledge of LIDS shall cover all normal processes, as well as its knowledge shall also be expandable
Third, *time to response* defines the time between the detection of intrusion and IDS response. Similitude to the requirement of "time to detect", this requirement has a high priority, given that the shorter the response time, the lower the damages that can be caused by intrusion.
Requirement: The LIDS shall issue an alarm below one second as soon as intrusion is detected.
Forth, *identity trace back analysis* determines the capabilities of IDS in tracing and identifying attackers. This varies from rather straightforward methods (e.g. storing intrusions with time stamping and network addresses) to highly-complex approaches (e.g. using several tools to detect source of attacks in real time). The significance of this requirement is due to the fact that having sophisticated knowledge regarding motivated intruders facilitates determining the level of efforts and costs that should be invested in securing a system.
Requirement: The LIDS shall generate information concerning malicious processes including the owner and the leaving and incoming network packets of the malicious processes.

Security

This identifies the extent to which resources of IDS are defended against security attacks. The magnitude of this requirement is due to the dependency of detection accuracy on the unalterability of IDS resources. Based on the assumption that an attacker could adapt IDS resources to hide his/her malicious activities, intrusion may remain undetected for days, weeks or perhaps forever.
Requirement: The resources of LIDS shall under no circumstances be altered during its operations, as well as during the development process simply from a defined developer, who takes the responsibility for it.

Scalability

Considering this requirement plays an important role in specifying an IDS. Scalability is defined as the capability of IDS in analyzing the worst-number of events, without missing

intrusions or issuing *FPs* [13]. Thus, precisely analyzing the operation environment of LIDS is necessary to determine the worst-number.
Requirement: The LIDS shall be able to analyze worst-number of events that can occur in the system.

Maintainability

This requirement has its roots in software requirements analysis. It focuses on identifying the modifiability, adaptability and extensibility of a software [53]. Intrusion detection systems are largely developed as software projects; therefore, defining this requirement for IDS is inevitable.
IDS should be developed maintainable given that attack methods vary as time passes. The adaptability, modifiability and extensibility of IDS enables monitoring new resources or changing detection approaches to detect novel attack methods.
This requirement is also considerable from the standpoint of embedded applications. Embedded systems are advised for the cost-effective implementation of applications. However, developing unmaintainable software projects can prompt creating new projects in the future, which eliminates the cost-effectiveness of embedded systems.
Requirement: The LIDS shall be implemented in such a manner that it will be adaptable, modifiable and extensible for adding future features. More precisely, the functions shall be implemented small and perform one task. The name of variables shall be defined descriptive. Moreover, it shall be considered to write clear and readable code and comment.

3.2 Functional requirements

The intrusion detection process comprises three phases: the phase of data collection, data analysis and response [54]. Correspondingly, IDS should provide functional modules that cover these phases. This section discusses functional requirements according to these phases for both components of the LIDS as follows. Table 3.2 summarizes these requirements according to their objectives and rationales.

Collecting

Defining this requirement in the form of questions facilitates a comprehension of its priority. These questions are as follows: How should IDS collect data? How are events generated from input data and prepared for analysis? Where should sensors of IDS be placed? Is the data collection through just one sensor adequate? Is there any feature extraction or data fusion required?
Considering the answers of these questions enhances the conception of the collecting functional module.
LIDS - Requirement: The LIDS shall have a sensor in the kernel to observe processes. As soon as a process is created, LIDS shall find the executable file of that process to determine

Table 3.1: Non-functional requirements

Objectives	**Requirements**	**Rationale**
Availability of system	LIDS shall not cause system failure.	in safety-critical domain cause physical after-effects.
Availability of IDS	LIDS shall be available 99.99% alongside the embedded device.	during down time of LIDS intrusions can harm the system.
Accuracy	LIDS shall issue false positive alarms with a probability below 0.01%.	defining inaccurate threshold causes either *FP*, or missing intrusions.
Completeness	LIDS shall detect intrusions with a probability of 99.99%.	not detected intrusions can cause safety-critical impacts on the system.
Time to detect intrusion	LIDS shall detect intrusions as soon as they occur.	delayed detection of intrusions can cause significant damages on system.
Knowledge completeness	The knowledge of LIDS shall cover all normal processes.	either *FP* are issued or intrusion are not detected.
Knowledge expandability	The knowledge of LIDS shall be expandable.	changes in system behavior are detected either as *FP* or intrusions are not detected.
Response time	LIDS shall issue alarm below one second as soon as intrusion is detected.	delayed detection of intrusions can cause significant damages on system.
Identity trace back analyze	LIDS shall provide information concerning malicious processes including the owner and the leaving and incoming network packets of the malicious processes.	sophisticated knowledge regarding intruder facilitates determining the ration of effort to secure the system.
Security	The resources of LIDS shall not be modifiable during run-time.	intruder can hide malicious activities by altering IDS resources.
Scalibility	LIDS shall be able to analyze worst-number of events that the system can process.	IDS misses intrusions, or generates *FP*.
Maintainability	LIDS shall be implemented adaptable, expandable and modifiable.	Non-maintainable software projects increase the likelihood of creating new projects for even small changes, which cause high costs for companies.

the admissibility of that process. In addition, it shall collect system call traces to observe process behavior more precisely.

Analysis

Events issued from collecting engines are ready to be analyzed in this module. This requirement regards two aspects: first, how to analyze the collected events, which may involve analyzing using highly-complex machine learning algorithms or a rather simple database query; and second, it defines where to place the analyze functional module. More precisely, are IDS sensors allowed to analyze the events themselves locally or do they simply perform collecting and other entities are responsible for the analysis?
LIDS - Requirement: In this phase, the LIDS shall determine whether an application is allowed to run on the device. Moreover, it shall analyze system call traces generated by the programs using machine learning algorithms for detecting highly-intelligent attacks.

Response

Activation of this module depends on the analysis functional module. In case of detecting an anomalous event, the response module will be activated. This requirement defines the reaction type of IDS, whether it should issue an alarm concerning an anomalous event or invoke a decision engine for specific actions.
In addition, it is required to determine in case of distributed IDSs which entity is responsible for issuing an alarm (e.g. whether each IDS is allowed to issue an alarm, or a central management takes the decision after collecting alarms from all IDSs).
LIDS Requirement: In case of detecting anomalous programs, LIDS shall issue an alarm concerning the malicious program including the owner and the network connections of the malicious process.

Table 3.2: Functional requirements

Objectives	**Requirements**	**Rationale**
Collecting	LIDS shall have a sensor in the kernel to observe processes. As soon as a process is created, LIDS shall find the executable file of that process to determine the admissibility of that process. In addition, it shall collect system call traces to observe process behavior more precisely.	Inadequate analysis of this requirement may cause missing intrusions.
Analysis	LIDS shall determine whether an application is allowed to run on the device. Moreover, it shall analyze system call traces generated by the programs for detecting highly-intelligent attacks.	Analyzing the events in an improper place or based on an approach which may cause impacts on the system performance can cause missing intrusions or affecting the overall system performance.
Response	In case of detecting anomalous programs, LIDS shall issue an alarm concerning the malicious program.	In a safety-critical environment such as in-vehicle systems disturbing the system functionality can cause irreversible after-effects.

4 Concept

This chapter introduces the architectural design of LIDS. Considering the requirements discussed in the previous chapter and the characteristics of embedded systems, it can be seen that a simple HIDS that operates on an embedded device is not capable of meeting our requirements. Therefore, this work introduces a new HIDS concept comprising two sub-components - namely Watchdog and Secure Homeland - that combines one simple and one complex approach. Watchdog is responsible to carry out the simple approach of whitelisting, whereas by contrast the second sub-component Secure Homeland performs the complex technique of analyzing system call traces using machine learning algorithms on another machine. Fig. 4.1 shows the abstract model of the LIDS.
Based on the focus of this work on evaluating anomaly detection techniques, communication protocols between embedded devices and the server are not discussed in this work.

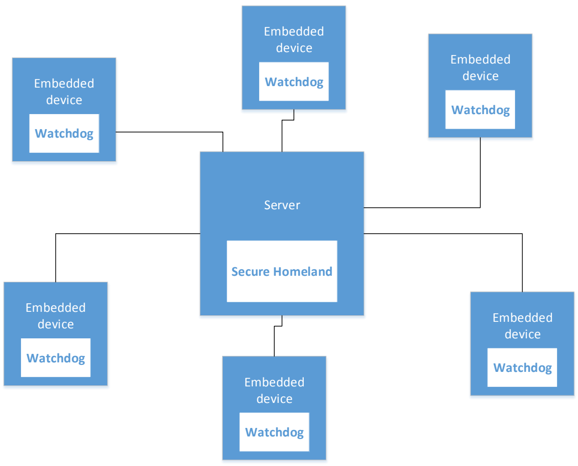


Figure 4.1: Abstract model of LIDS

4.1 Watchdog

This sub-component is designed to perform application whitelisting (AWL) on an embedded host. This is due to the following characteristics of embedded systems that propose application whitelisting as a good candidate for satisfying requirements discussed in the previous

section.
First, the majority of embedded systems have restricted computation resources, as well as bounded memory capacity. Therefore, implementing highly-complex detection techniques may exceed their computation limitations, and consequently can have adverse impacts on the performance of these systems. Second, operation environment in embedded applications is mostly static. More precisely, processes, network protocols, IP addresses for sending and receiving packets, etc. are clearly predefined. Third, infrequent changes and the restricted number of applications that operate on embedded devices simplifies building the detection model based on the system's normal behavior rather than generating IDS knowledge based on malicious applications.
Among the detection techniques researched by various studies, AWL conforms to those characteristics and defined requirements (in chapter 3). It performs a simple but sufficiently powerful detection process with consideration of computation constraints, a static operation environment and restricted number of applications in embedded systems. Moreover, through adding AWL as the final step of the development process of software products that should operate on an embedded host, the requirements "completeness, accuracy and completeness of stored knowledge" can be satisfied. More precisely, this final step specifies a condition under which a software has been developed and is ready for operating on an embedded host (before migrating to the host), whereby it must be added to the whitelist. Without meeting this condition, no software is allowed to operate on the device. Consequently, the failure to detect normal behavior as malicious activity (*FPs*) and the completeness (and expandability) of Watchdog resources is guaranteed.
Generally, application whitelisting (AWL) comprises three main phases [55]: generating a whitelist of executable files during the normal operation of system (learning), checking the checksum of executable files (e.g. hash values) at run-time (testing), and deny or report process with a modified executable file. The prime strength of AWL in contrast to "blacklisting" (i.e. covers malware information in a list) is the absence of a continual updating process of intrusion knowledge on hosts [55]. Particularly in some domains in which thousands of devices must be updated concurrently, frequent updating reflects a major challenge. Furthermore, in embedded applications the size of HIDS resources should be carefully considered. For instance, the size of a whitelist for an embedded device with five applications is much smaller than a blacklist including a large list of various malwares. Thus, intrusion detection in an embedded environment based on AWL is more reasonable than blacklisting.
According to AWL, Watchdog operates in three phases: learning, testing and reporting. In order to separate the learning phase from testing and reporting phases, Watchdog has been separated into two various concepts. A user-space software has been designed for the learning phase, and a Linux security module integrated to the Linux kernel has been designed for the testing and reporting phases. The reason behind this decision will be discussed more precisely in explaining the design concept of the test software.
The user-space software carries out creating the whitelist of executable files existing in the system. It takes a given path to a container of executable files as input, and creates a key-value-based list (whitelist). The key in each element of this list is the name of the executable file, and the value is the hash value of that file. This software is able to generate hash values

using various hash functions such as SHA1 and SHA256. In this respect, it should be noted that Watchdog does not use the known hash function MD5, which has been recently targeted by various security research, and in some cases successfully broken such as in [56].
Next, the issued whitelist will be added to the test software manually. The reasons behind this decision are as follows: first, in most domains, changes in normal behavior of the system rarely occur and thus frequent changes in the whitelist are not required; second, generally the number of applications that operate on an embedded device is small and thus a small list should be moved to the test software; and third, moving the whitelist to the test software manually guarantees protection of HIDS resources, whereby Watchdog can meet the "security" requirements. More accurately, the resource of Watchdog is hardcoded in the kernel binary file and thus their alteration requires recompiling the kernel on the embedded device whereby its occurrence has a probability around zero.
The test software of Watchdog has been designed to be integrated into the kernel space, given that integrating Watchdog into the kernel enables it to meet several requirements (discussed in chapter 3) including "availability of IDS, time to detect intrusion, and time to response".
Availability of IDS: Being part of the kernel enables Watchdog to run concurrent with the embedded device. Accordingly, all processes during the run-time of the system will be monitored and it will be guaranteed that the entire system behavior is observed.
Time to detect intrusion and time to response: Due to the integration into the kernel, Watchdog can monitor processes online, which reduces the time to detect anomalies and the time to respond.
The Linux kernel provides various methods for integrating extra functions into the kernel (although without altering the kernel itself). The most commonly-known method is the Linux dynamically loadable kernel modules, which proposes an appropriate possibility for extending kernel functions and reduces the development time [57]. Nevertheless, the main disadvantage of these modules is that users with root privilege (or internal intruders) are able to remove these modules from the kernel space. This feature of loadable kernel modules is contrary to the availability requirement.
A further possibility for integrating Watchdog into the kernel space is by using the Linux security module framework (LSM). Wright et al. [1] defined the LSM framework as a lightweight access control framework that enables utilizing various security models on the same kernel without affecting the kernel performance. Thanks to this framework, security functions are independent from the kernel and adaptable to various security requirements such as those in embedded systems. In this framework, security modules have access to the kernel objects using so-called *hooks*, whose architecture is shown in Fig. 4.2 [1]. These *hooks* are categorized into five main groups: task hooks (controlling process operation), program loading hooks (checking programs, once executed), IPC hooks (access to IPC object), filesystem hooks (controlling file operation) and network hooks (controlling sockets and network packets). The most significant set of hooks appropriate for AWL is program-loading hooks, which enables Watchdog to access the executable file of programs before their execution. Having all of this in mind, in the author's view LSM is the most appropriate method to implement AWL in Watchdog.
The program-loading hooks are designed in the interest of checking specific user-defined

conditions within the kernel before executing a program. Accordingly, once a program is executed, before allocating system resources to the program, one specific hook implemented in Watchdog will be invoked. The hook implemented by Watchdog can control various features of that program, such as the page table structure, name, executable file, etc. However, in case of AWL, only the executable file of the program will be considered.
Watchdog performs AWL using program loading hook in two steps: First, Watchdog searches the whitelist for the name of the executable file. If the given file cannot be found in the list, an alarm will be issued. Otherwise, the next step will be performed, in which Watchdog calculates a new hash value of the executable file using the Linux Crypto API and compares it with the anticipated value stored in the whitelist. If a match is found, the program is ignored; otherwise, the program is detected as an anomaly and an alarm will be issued immediately.

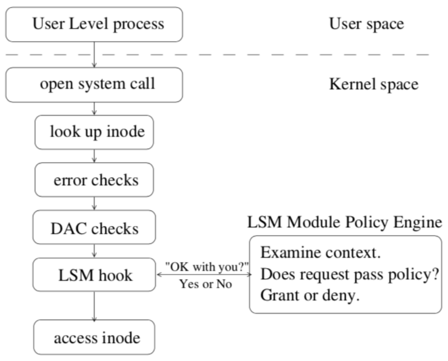


Figure 4.2: LSM hook architecture [1]

Based on implementing the kernel hooks in Watchdog, the time to detect anomalous programs is significantly diminished. Furthermore, as soon as a malicious program is detected, Watchdog will issue alarm, and correspondingly the time to respond to anomalies will also be diminished.

4.2 Secure Homeland

The second sub-component of LIDS is Secure Homeland, which is designed to analyze process behaviour much more precise than AWL. This is due to the limitations of AWL [55] including:

- Generally, AWL is able to detect executable file changes. However, vulnerabilities in network-based applications such as remote authentication dial-in user service (RADIUS) applications can be exploited without any changes in the executable file of the application. Thus, AWL is inadequate to detect/prevent intrusions issued by whitelisted applications.

- Any uncovered file by the whitelist - even if it is not a malware - will be flagged as an anomaly. This could cause issuing *FP*s, which can increase workload in the system.

- AWL cannot detect modified executable files that generate the same hash values. Recent studies have presented attack methods that can break hash algorithms such as MD5 successfully [56].

Keeping all of these considerations in mind, applying more complex anomaly detection approaches for embedded systems is inevitable. In the recent past, data mining approaches have been widely employed in anomaly detection systems [48]. These approaches help detection systems to summarize a large amount of data in a systematic manner, as well as gaining a model of that [49]. Classification is one of the most commonly-used data mining approaches regarding anomaly detection systems. Classification-based techniques have two prime phases, namely training and testing [45]. In the training phase, a labeled training data set is used to build the classifier. Next, this classifier is used in the testing phase for labeling testing instances. Accordingly, the capabilities of the classifier in detecting anomalies will be tested.
The prime strength of classification anomaly detection techniques is their fast test phase in comparison to other data mining approaches such as clustering-based methods [45]. From the author's view, this feature nominates classification-based data mining approaches for application in anomaly detection systems in embedded applications. Thus, Secure Homeland has been designed to apply classification-based anomaly detection approaches.
In this respect, it should be noted that in this thesis the anomaly detection will be performed in supervised mode, in which both training and testing data sets contain labeled instances including normal and anomalous behavior. Typically, selecting the appropriate anomaly detection mode is based on the extent to which labeled data set is available. The data set used in this work (ADFA-LD) comprises labeled instances for both normal and anomalous behavior. Therefore, the supervised mode has been selected to test capabilities of the classifier.
According to the supervised mode, the author has divided this data set into two separate sets, one for training and another for testing. The training data set contains two-thirds of the ADFA-LD data set including normal and abnormal instances, which have been selected completely randomly. The testing dataset covers the remaining one-third of the data set (normal and abnormal instances), which has never been seen by the classifier.
Secure homeland has a feature extraction engine, which uses those two data sets (two-thirds for training and one-third for testing) to convert them into the attribute-relation file format (ARFF) [58] format. The conversion mode depends on the data representation method, which may be sequence-based or frequency-based. Fig. 4.3 shows the abstract model of engines in Secure Homeland. Further aspects concerning the data representation method and selected machine learning algorithms are discussed as follows.

4.2.1 Data representation

This work considers two main representation methods of system call traces - sequence- and frequency-based - which are discussed as follows.

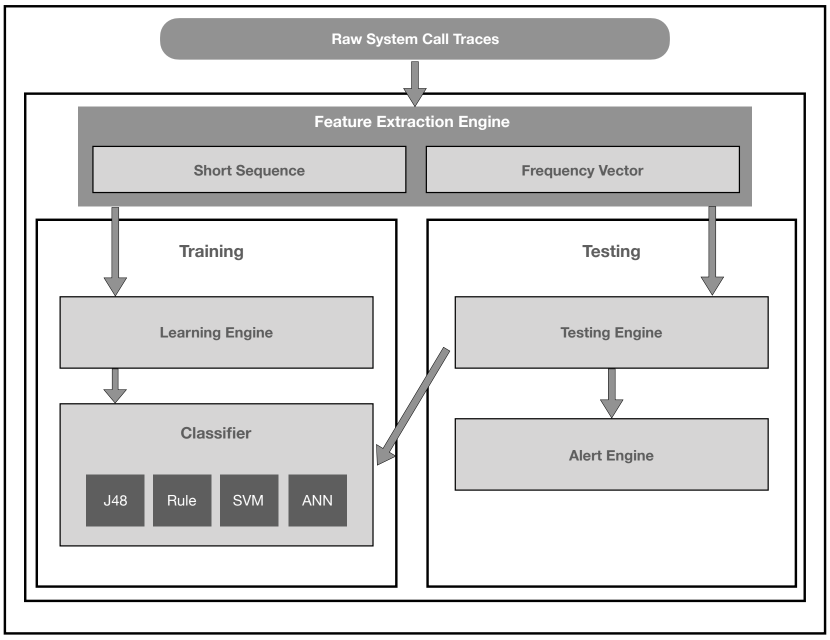


Figure 4.3: Architecture of learning and testing process in Secure Homeland based on [2]

Sequence-based

The sequence-based data model in this work is similar to the window-based model discussed in [23]. Accordingly, each system call trace is divided to fixed-length short sequences. Let us assume the following system call trace and the fixed length 3 (k=3),

2,0,9,2,2,97,9,3

then this trace is divided into short sequences with length of 3 as follows (it should be noted that according to this method the first system call in the window is not counted):

- 2,0,9,2
- 0,9,2,2
- 9,2,2,97
- 2,2,97,9
- 2,97,9,3

In addition, for each trace the repeated sequences have been removed from the data set and before building and testing the classifier the dataset has been randomized. For the sequence-based method, the author considered three various lengths: 3, 5, and 10. The window-sizes 3 and 5 have been researched in many research works, however the window-size 10 have been discussed rarely. Therefore, the author selected these sizes. Short sequences of normal traces are labeled as normal instances, and those from attack traces as anomalies.

Frequency-based

Frequency-based model in this work is based on the vector space model discussed in [3]. Consequently, instances have been generated in the form of feature vectors, whose index matches frequency of system call with the same number in that trace. In the ADFA-LD data set, there are 341 various system call numbers, and thus each feature vector (instance) has 341 attributes. Similar to the sequence-based model, vectors of normal traces have been labeled as normal, and anomaly traces as anomalies. For instance, consider the following trace,

2,0,9,2,2,8,9,3

then based on the assumption that OS has ten various system calls, the feature vector based on the vector space model is as follows:

1,0,3,1,0,0,0,0,1,2,0, normal

4.2.2 Machine Learning Algorithms

One of the targets of this thesis is to study the detection accuracy and feasibility of classification-based anomaly detection methods. In order to evaluate these techniques, Weka workbench [58] has been used, which is a set of machine learning algorithms for data mining tasks, among various classification-based techniques. For this thesis, the author employed the following machine learning algorithms: Bayesian networks, rule-based methods, decision trees, instance-based knowledge methods and artificial neural networks.
Although this thesis will not focus on analyzing classification-based algorithms but rather on their application in anomaly detection in embedded applications, the classification techniques are briefly discussed as follows:
The *Bayesian network* algorithm is one of the most universal and widely-used classification-based techniques. The base of this algorithm is the *Bayes theorem*, in which based on the calculation of conditional probability on each node a Bayesian network is generated. This network is a directed graph with no occurrence of cycles [59].
Rule-based anomaly detection methods apply a rule learner engine that covers the system's normal behavior. Each test instance is then classified depending on whether it is covered by any such rules (normal) or not (anomaly) [45]. JRip - proposed by William W. Cohen - is an instance of rule-based methods, which provides a rule learner and repeated incremental pruning to produce error reduction (RIPPER) for classifying instances efficiently [59].
Decision trees - a further rule learning method - is another widely-used classification technique in which the classifier is built in the form of decision trees based on the concept of information entropy [59].
In contrast to rule-based algorithms, the instance-based knowledge (IBK) technique builds the classifier by employing only specific instances [60]. These algorithms do not consider abstractions from data such as rules or decision trees; rather, they employ the instances themselves [59]. More specifically, in the learning phase training instances are memorized. Next, in testing, each test instance is searched in the memory. If it is found, the instance is

labeled as a normal instance, an otherwise as an anomaly [45].
Similar to other classification-based approaches, neural network-based methods operate in two various methods, namely learning and testing. In the training phase, the training data including normal behavior is used to train a neural network. By contrast, in the testing phase the neural network takes each test instance as input. Refused test instances are declared as anomalies; otherwise, if the test instance is accepted by the network, it is declared as normal behavior [45].
In the interest of improving the detection accuracy of basic classification methods and taking advantage of various classification methods, another classification method called ensemble learning was proposed [61]. Accordingly, instead of using only one basic classifier, a collection of basic classifiers is used for labeling test instances. Boosting is one of ensemble learning methods that have been developed recently. It is an approach that covers an iterative training set-generator method, as well as an algorithm that combines basic classifiers [62]. In this thesis, we also consider stacking and bagging as instances of ensemble learning. Overall, ten well-known classification-based algorithms - from the author's view - have been selected to be evaluated in this work. These algorithms are in different categories in Weka software, which are demonstrated in Table 6.3.

Table 4.1: List of chosen algorithms and their options

Category	**Algorithm**	**Options**
Bayes	Naive Bayes	–
Function	LibSVM (Support Vector Machine)	Radial Basis Function (RBF) Kernel
	MultilayerPerceptron	–
Lazy	IBk (k-nearest neighbors)	k=3
Rules	JRip(RIPPER)	Folds=3
	PART	unpruned = true, confidenceFactor = 0.25, numFolds = 3
Trees	J48(C4.5)	minNumObj = 3, confidenceFactor = 0.25, unpruned= true
Meta	Stacking	Classifiers = J48, AdaBoostM1, Bagging
	Bagging	Classifiers = RandomTree
	AdaBoostM1	Classifier = DecisionStump

4.3 Outcome

Dividing LIDS into two sub-components enables providing a balance between method complexity and computation constraints. The first sub-component called Watchdog, which operates on embedded host, performs application whitelisting, which is integrated within the Linux kernel. The second sub-component operates on a server and analyzes system call traces using data mining techniques for detecting highly-complex attack methods. The combination of these two sub-components within one concept allows LIDS to meet the requirements discussed in section 2. To my best knowledge, LIDS is the first intrusion detection system that combines a Linux kernel-based method with machine learning algorithms for protecting embedded systems and is evaluated based on the ADFA dataset.

5 Implementation

This chapter describes the implementation of two sub-components, Watchdog and Secure Homeland. Two different frameworks have been selected for the implementations: a Linux security module and a Java command-line software. Details concerning the implementation of the components are discussed in the two following sections.

5.1 Watchdog

In this section, the project structure and development environment of Watchdog are explained.

5.1.1 Environment

The following software and operating system were used for the implementation of two parts of Watchdog:

- Test part
  - The Linux distribution Debian 8, kernel V4.4.9
  - Linux Kernel Programming IDE (LinK+ IDE)
  - Required packages for compiling the kernel
- Train part
  - Xcode IDE V.8.0[63]
  - Mac OS High Sierra
  - GCC, the GNU Compiler Collection V.6.4 [64]

5.1.2 Project structure

As previously discussed in chapter 4, Watchdog comprises two separated components, one for learning and another for testing. A user-space software carries out the learning phase, which is implemented in c++ programming language. This software is a command-line tool, which contains one main function, whose input argument is a path, where executable files are gathered. Through executing this software, the given path is searched for binaries and for each binary its name and hash value is calculated and stored in the whitelist. When calculating hash values for all binaries is accomplished, the whitelist is written to a file.
By contrast, the testing phase is implemented based on the Linux security module framework.

Accordingly, for compiling Watchdog, the kernel should also be compiled again. The build process of the Linux kernel includes choosing various numbers of compiling options, which in some cases could be very complex. However, this step is enhanced by the kernel build system (kbuild), in a way that in order to be part of the kernel, each entity should have two main files: kconfig and Makefile. Kconfig declares config symbols and their attributes (e.g. type, description and dependencies) and makefiles are normal GNU makefiles. In addition, the contents of these files must follow the syntax of security modules.
The main file in the test part of Watchdog is *ids.c*, which includes implementation of *hooks*. Moreover, *common.h* contains declaration of helper functions to separate implementation of *hooks* from other functions.

5.1.3 Important implementation aspects

Watchdog implements the *bprm_check_security* hook, which enables accessing executable file of program. The list of *hooks* for task and program execution operations are demonstrated in appendix *A* (attached). Watchdog makes use of two other Linux kernel facilities, *linked lists* (declared in the header file /include/linux/list.h) and the *Linux kernel crypto API* [65]. Once the Linux boots up, Watchdog as part of the kernel is initialized, which causes invoking two functions, *security_add_hooks* (adds implementation of hooks to the kernel) and *ids_init* (initializes the whitelist). During running of OS, once a program is executed, the *bprm_check_security* hook is invoked, which invokes its implementation *ids_bprm_check_security* in Watchdog. This function is given a pointer of *struct linux_binprm *bprm* including the address to the executable file of the program. First, the name of the executable file will be searched in the whitelist, and if it is not found the responsible function for issuing alarm will be invoked. Otherwise, the second step involves calculating a new hash value of the executable file using the Linux crypto API and comparing it with the expected value stored in the whitelist. If matching is found, the program is ignored; otherwise, an alarm is generated. Each time the function for alarm issuing is invoked, the further activities of that program including entering/leaving packets, etc. will be logged for intruder identity tracing back.

Listing 5.1: Hooks and structures in *Watchdog*

```
 /*
  * Using linked list implemented in the Linux kernel for defining the whitelist
  */
    struct ids_digest_list{
        struct list_head list;
        char *name;
        char *path;
        u8 *digest[HASH_BYTES_MAX];
    }

/*
 * The whitelist
 */
    struct ids_whitelist;

/*
 * The function for establishing the whitelist defined in ids.c
 */
    static int ids_init(void);
/*
 * Function of the Linux Crypto API for calculating hash values
 */
    int crypto_shash_digest(struct shash_desc * desc, const u8 * data, unsigned
    int len, u8 * out);

/*
 * This structure is used to hold the arguments that are used when loading
 binaries. Watchdog considers the file pointer in this structure.
 */
struct linux_binprm {
        char buf[BINPRM_BUF_SIZE];
        .
        .
        .
        struct file * file;
        struct cred *cred;      /* new credentials */
        const char * filename;  /* Name of binary as seen by procps */
        .
        .
        unsigned long loader, exec;
} __randomize_layout;
```

5.2 Secure Homeland

In this section, the project structure and development environment of Secure Homeland are explained.

5.2.1 Environment

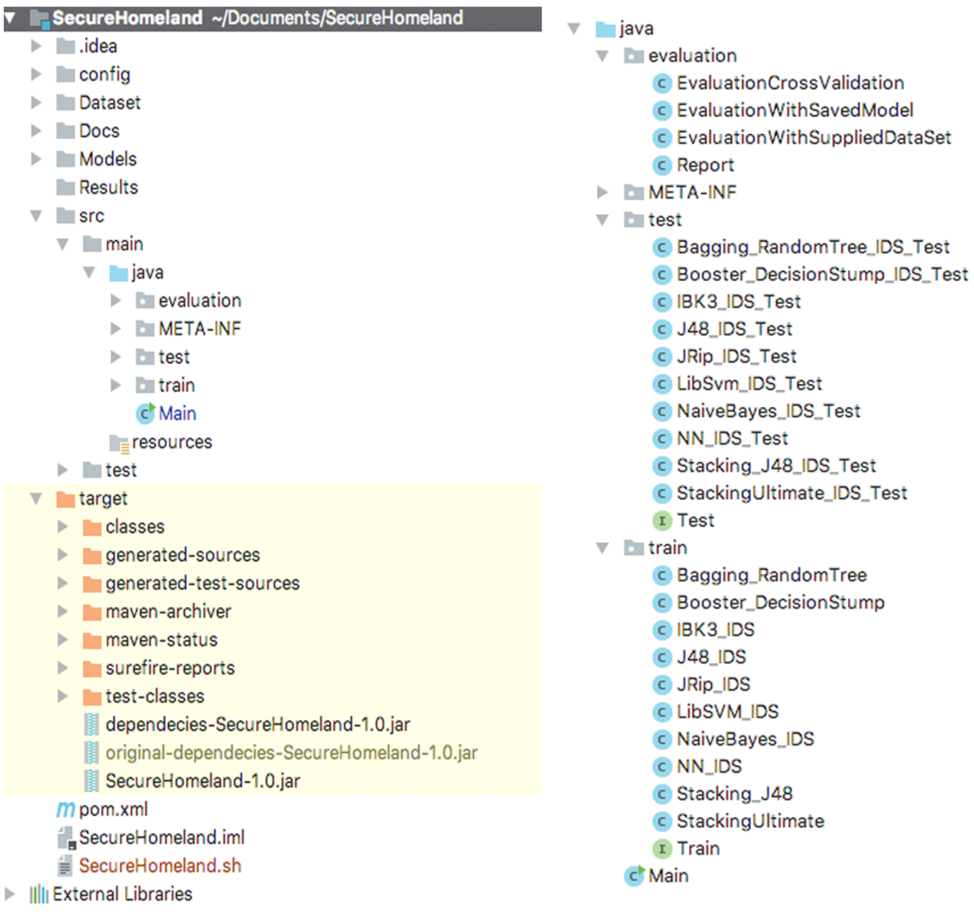
The following software and operating system were used for the implementation:

- macOS High Sierra Version 10.13.2
- Java Development Kit (JDK) 8
- IntelliJ IDEA [66]
- Data mining software Weka 3.7.12 [58]
- Apache Maven 3.5.2 [67]

5.2.2 Project structure

Secure Homeland has been created as a Maven project that enhances managing the complexity of dependencies. The implementation within the IntelliJ project is separated into three distinguished packages - train, test and evaluation - as depicted in Fig. 5.1.
The following listing briefly describes the single packages and folders of the project in alphabetical order to provide an overview of the implementation:

Config
This folder includes a config file, namely configIDS.properties, which contains configuration



```
▼ SecureHomeland ~/Documents/SecureHomeland
  ▶ .idea
  ▶ config
  ▶ Dataset
  ▶ Docs
  ▶ Models
    Results
  ▼ src
    ▼ main
      ▼ java
        ▶ evaluation
        ▶ META-INF
        ▶ test
        ▶ train
          Main
        resources
    ▶ test
  ▼ target
    ▶ classes
    ▶ generated-sources
    ▶ generated-test-sources
    ▶ maven-archiver
    ▶ maven-status
    ▶ surefire-reports
    ▶ test-classes
      dependecies-SecureHomeland-1.0.jar
      original-dependecies-SecureHomeland-1.0.jar
      SecureHomeland-1.0.jar
    pom.xml
    SecureHomeland.iml
    SecureHomeland.sh
▶ External Libraries

▼ java
  ▼ evaluation
      EvaluationCrossValidation
      EvaluationWithSavedModel
      EvaluationWithSuppliedDataSet
      Report
  ▶ META-INF
  ▼ test
      Bagging_RandomTree_IDS_Test
      Booster_DecisionStump_IDS_Test
      IBK3_IDS_Test
      J48_IDS_Test
      JRip_IDS_Test
      LibSvm_IDS_Test
      NaiveBayes_IDS_Test
      NN_IDS_Test
      Stacking_J48_IDS_Test
      StackingUltimate_IDS_Test
      Test
  ▼ train
      Bagging_RandomTree
      Booster_DecisionStump
      IBK3_IDS
      J48_IDS
      JRip_IDS
      LibSVM_IDS
      NaiveBayes_IDS
      NN_IDS
      Stacking_J48
      StackingUltimate
      Train
    Main
```

Figure 5.1: Project structure

properties for the project. The properties of this project includes the path to the training data set, testing data set, path for reading or storing the model, selected machine learning algorithm, etc.

Dataset
This folder includes data sets for training and testing, which have been generated by the feature extraction engine.

Evaluation
This package contains classes responsible for the evaluation of classifiers. Depending on the test mode (test with supplied test, cross-validation mode and test with saved classifier), the test method implemented in these classes will be invoked.

Models
This folder is used to store or read classifier that have already been stored.

Test
This package contains all algorithms for evaluating classifiers. For each training classifier, there is a test class. All test classes implement an interface called test. Moreover, it is configurable to test classifiers whether with a saved model, with supplied test data set or in cross-validation mode.

Train
Training algorithms are gathered in the train package. Each training class implements the train interface and invokes the corresponding train algorithm from Weka [58] for building classifiers.

5.2.3 Important implementation aspects

During this project, the author favored composition over inheritance, "Hasing-A relationship over is-A". This design principle enables the modifiability of classes individually. The design model of this project is mainly based on the "strategy design pattern", in which varying parts of code are encapsulated in interfaces. In order to provide a better understanding of the project architecture, its UML diagrams are shown in Fig. 5.2.
This software will be used as a Java command line tool and thus for providing all dependencies for the final Jar file, the *Shade* Maven-Plugin has been used.

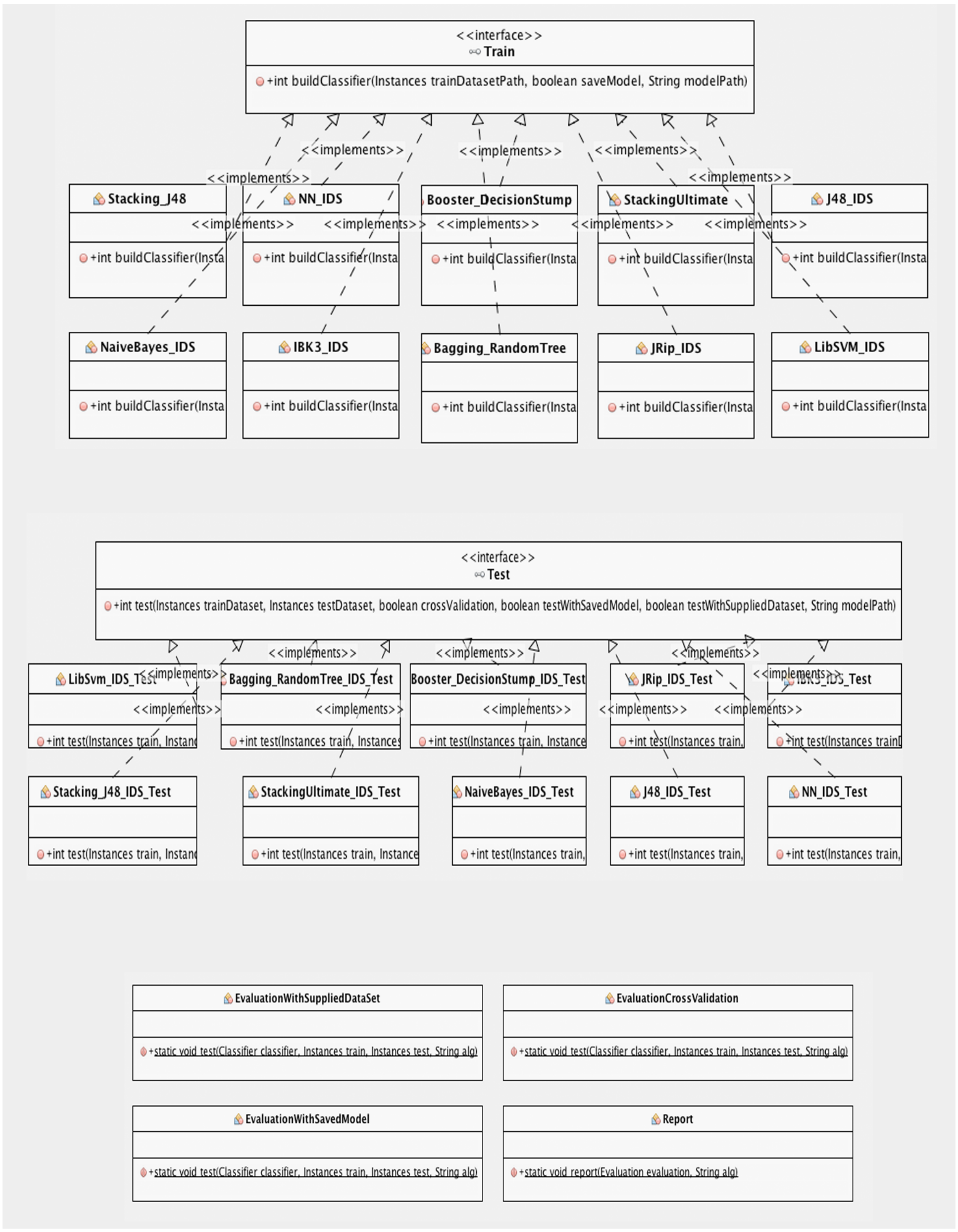

<<interface>>
Train
+int buildClassifier(Instances trainDatasetPath, boolean saveModel, String modelPath)
<<implements>>
Stacking_J48
NN_IDS
Booster_DecisionStump
StackingUltimate
J48_IDS
NaiveBayes_IDS
IBK3_IDS
Bagging_RandomTree
JRip_IDS
LibSVM_IDS
+int buildClassifier(Insta
<<interface>>
Test
+int test(Instances trainDataset, Instances testDataset, boolean crossValidation, boolean testWithSavedModel, boolean testWithSuppliedDataset, String modelPath)
LibSvm_IDS_Test
Bagging_RandomTree_IDS_Test
Booster_DecisionStump_IDS_Test
JRip_IDS_Test
IBK3_IDS_Test
Stacking_J48_IDS_Test
StackingUltimate_IDS_Test
NaiveBayes_IDS_Test
J48_IDS_Test
NN_IDS_Test
+int test(Instances train, Instanc
EvaluationWithSuppliedDataSet
+static void test(Classifier classifier, Instances train, Instances test, String alg)
EvaluationCrossValidation
+static void test(Classifier classifier, Instances train, Instances test, String alg)
EvaluationWithSavedModel
+static void test(Classifier classifier, Instances train, Instances test, String alg)
Report
+static void report(Evaluation evaluation, String alg)

Figure 5.2: UML diagram of all packages in Secure Homeland

6 Evaluation

The following chapter validates the components of LIDS, Watchdog and Secure Homeland against the requirements discussed in chapter 3. According to the separated architectural design of LIDS, the evaluation results will be provided in the two following sections individually.

6.1 Application whitelisting

In this section, the evaluation environment, method and results of evaluating of application whitelisting - which has been implemented in Watchdog - is discussed.

6.1.1 Evaluation environment

The following hardware and operating system were used for the testing of Watchdog.

- VWware Workstation [68] for creating a virtual machine configured as the Raspberry Pi Zero W [69].
  - CPU : 1GHz, single-core CPU
  - RAM : 512MB
  - Storage : 6GB
- Linux distribution Debian 8, kernel V4.4.9

6.1.2 Evaluation method

The evaluation of Watchdog includes four main steps. First, precise analysis of all applications that should operate on the host. In order to avoid *FP*s, normal behavior of the host machine should be covered completely. Therefore, a list of all programs that should run on the system is established. Second, issuing the whitelist by means of the learning software. Subsequently, this list - which comprises 30 entries - is added to the testing part of Watchdog in the LSM module. The last step of the third phase is newly recompiling the kernel. Fourth, running all legal programs and several illegal programs that are not covered by the whitelist, or the executable file of legal programs have been altered. Finally, the alarm results are collected to analyze whether all legal programs detected as normal activity, and illegal programs detected as anomalies.

6.1.3 Results

Throughout four days testing, *Watchdog* could achieve 100% detection accuracy and 0% *FP*s. All running legal programs are detected as normal behavior. Additionally, illegal executable files or binary files whose content had been modified were detected as anomalies. Moreover, required information to trace back the identity of intruder has been audited. The decisive factor for achieving this detection rate is the precise analysis of normal behavior. The more complete the list, the lower the *FP*s rate. The achieved results showed that based on this concept most defined requirements in chapter 2 for a host-based anomaly detection system have been met. The integration of Watchdog within the kernel space enables satisfying two requirements concurrently, namely the availability of the system and availability of IDS. Being part of the operating system enables Watchdog to be available throughout the operating time of the host. Furthermore, the design concept relies on monitoring processes using facilities implemented within the kernel without disabling any services. Thus, Watchdog under no circumstances will affect the system performance or cause system failure. Scalability is a further requirement fulfilled by this integration. In particular, worst-number of events that should be analyzed by Watchdog corresponds with the worst-number of events that the host machine can handle. This concept also satisfies detection and response time requirements. Anomalous programs or legal programs pointing to altered executable files are detected before they are allocated resources by the operating system. This results in detecting intrusions before execution. Correspondingly, the response time is reduced given that alarms concerning anomalous programs are issued before their execution.
This concept also fulfills the security requirement by hard-coding the list within the kernel space. By this means, the modification of IDS resources (the whitelist) requires manipulating the kernel itself, which is to some extent a complicated attack method. In addition to these requirements, this concept satisfies the identity trace back requirement by auditing all information concerning malicious processes including time-stamp, name, path to executable file, and incoming and outgoing network packets. However, meeting requirements concerning knowledge completeness and expandability is conditional upon precisely analyzing use cases and programs running on the host machine.

6.2 System call trace analysis

In this section, evaluation environment, method and results of analyzing of system call traces implemented in Secure Homeland will be focused.

6.2.1 Evaluation environment

The following hardware and operating system were used for testing Secure Homeland.

- MacBook Pro
  - CPU : 2,2 GHz Intel Core i7
  - RAM : 16 GB 1600 MHz DDR3

    - Storage : 256 GB SSD

- macOS High Sierra Version 10.13.2
- Terminal tool on macOS

6.2.2 Evaluation method

Evaluation methodology: In the interest of evaluating Secure Homeland, the selected data set (ADFA-LD) was used to generate two main separate data sets, one for supervised mode and another for cross-validation mode. Both data sets cover the two representation modes, sequence- and frequency-based, discussed in chapter 4. The data sets for sequence-based mode comprise three sub-data sets according to the window size 3, 5, and 10. By contrast, the data set for frequency-based mode includes only one data set, in which system call traces were transformed into modified feature vectors.
The author ran experiments in one-class mode, in which one of two labels - normal or anomaly - were considered for each instance. Correspondingly, each algorithm with the selected options was ran on data sets first through the supplied test data set, and second through the cross-validation method. Fig. 6.1 demonstrates the structure of the above described data sets generated in this thesis.

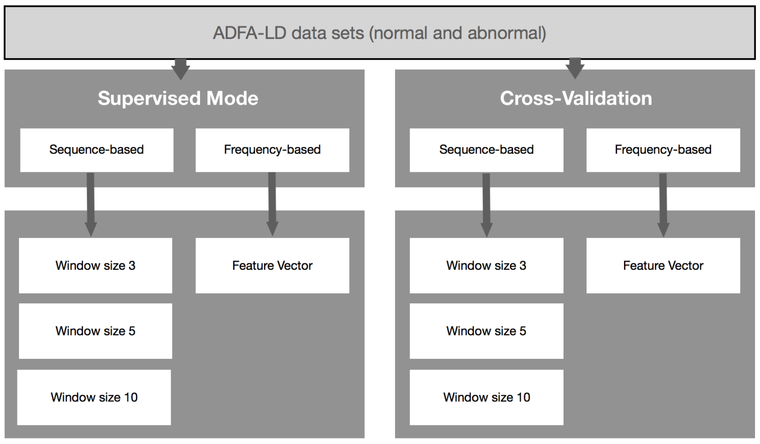


Figure 6.1: Generated data sets for supervised and cross-validation modes in this thesis

Evaluation metrics: In order to evaluate the performance of the classifiers and data representation modes, the following general evaluation metrics have been used, which are commonly used in information retrieval area and discussed in [3].

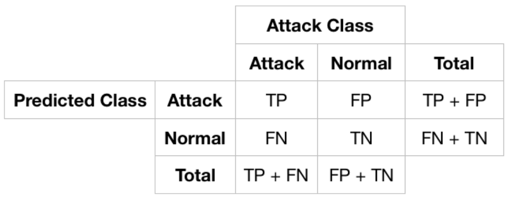
True Positive (TP): Number of anomalous traces detected as anomalies.

False Positive (FP): Number of normal traces detected as anomalies.

True Negative (TN): Number of normal traces noted as normal traces.

False Negative (FN): Number of anomalies detected as normal traces.

Fig. 6.2 demonstrates the confusion matrix used to calculate other quantities as follows:
Precision: It defines the proportion of anomalous traces detected as anomalies among the total number of traces detected as anomalies.



		Attack Class		
		Attack	Normal	Total
Predicted Class	Attack	TP	FP	TP + FP
	Normal	FN	TN	FN + TN
	Total	TP + FN	FP + TN	

Figure 6.2: Confusion matrix

$$Precision = \frac{TP}{TP + FP} \tag{6.1}$$

Recall: It determines the proportion of anomalous traces detected as anomalies among the total number of truly anomalous traces. It is also called the true positive rate (TRP).

$$Recall = \frac{TP}{TP + FN} \tag{6.2}$$

Accuracy: It defines the ratio of correctly-detected traces (normal traces detected as normal trace, anomalous traces detected as anomaly) among the total number of instances.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{6.3}$$

FP rate:It determines the proportion of normal system call traces predicted as anomalous traces by the classifier.

$$FPrate = \frac{FP}{TN + FP} \tag{6.4}$$

F-measure: It is a combination of precision and recall into a single quantity (harmonic mean of precision and recall).

$$F - measure = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}} \tag{6.5}$$

6.2.3 Results

The performance results including the false positive rate, precision, recall, f-measure and accuracy are shown in the following tables (detailed results are presented in appendix B. The achieved results for the windows-based approach are shown in Table 6.3 (for testing with supplied test data set) and Table 6.4 (for cross-validation mode). Moreover, the results

for frequency-based approach are presented in Table 6.1 (for testing with supplied test data set) and Table 6.2 (for cross-validation mode).
In addition to these tables, Figure 6.3 compares performance results of algorithms including the FP rate and accuracy in the form of a graphical presentation. As shown in Figure 6.3, there has been a distinct increase in the detection accuracy and a considerable decrease in the FP rate in the frequency-based approach in comparison to the window-based approach. In the window-based approach, the results show that raising the window size from 3 to 5 and 10 follows reduces of the FP rate and Accuracy, while increasing the train time. However, considering the FP rate and Accuracy, among algorithms Naive Bayes and Stacking achieved good performance with window size 10. In contrast to window-based, most algorithms achieved good performance in the frequency-based method, although Stacking and AdaBoostM1 had the best rates.

Table 6.1: Results of experiment for frequency-based method on ADFA-LD data set with one-class labels in testing with supplied test data set mode

Algorithm	**vector-size**	**FP Rate**	**Precision**	**Recall**	**Accuracy**	**F-Measure**
Naive Bayes	342	0,567	0,809	0,970	0,816	0,882
J48	342	0,032	0,995	0,973	0,972	0,984
JRip	342	0,005	0,999	0,963	0,967	0,981
IBK-k3	342	0,051	0,992	0,971	0,969	0,982
Multilayer Perception	342	0,050	0,997	0,891	0,893	0,941
Stacking	342	0,014	0,998	0,987	0,987	0,992
AdaBoostM1	342	0,008	0,999	0,980	0,982	0,990
Bagging	342	0,026	0,996	0,956	0,967	0,981

Table 6.2: Results of experiment for frequency-based method on ADFA-LD data set with one-class labels in cross-validation mode (Fold=10)

Algorithm	**vector-size**	**FP Rate**	**Precision**	**Recall**	**Accuracy**	**F-Measure**
Naive Bayes	342	0,649	0,764	0,980	0,780	0,859
J48	342	0,141	0,981	0,975	0,961	0,978
JRip	342	0,120	0,985	0,969	0,959	0,977
IBK-k3	342	0,168	0,976	0,977	0,959	0,976
Multilayer Perception	342	0,400	0,999	0,875	0,875	0,933
Stacking	342	0,115	0,983	0,988	0,974	0,985
AdaBoostM1	342	0,101	0,986	0,980	0,970	0,983
Bagging	342	0,103	0,986	0,977	0,968	0,982

Table 6.3: Results of experiment for various window-sizes on ADFA-LD data set with one-class labels with supplied test data set

Algorithm	window-size	FP Rate	Precision	Recall	Accuracy	F-Measure
	3	0,368	0,914	0,740	0,723	0,818
Naive Bayes	5	0,299	0,853	0,701	0,701	0,770
	10	0,077	0,853	0,627	0,799	0,723
	3	0,296	0,961	0,719	0,718	0,822
J48	5	0,176	0,947	0,647	0,701	0,788
	10	0,024	0,974	0,454	0,632	0,619
	3	0,343	0,965	0,706	0,702	0,815
JRip	5	0,199	0,944	0,662	0,684	0,778
	10	0,301	1,000	0,733	0,733	0,846
	3	0,315	0,947	0,728	0,723	0,823
IBK-k3	5	0,216	0,945	0,651	0,671	0,771
	10	0,039	0,967	0,402	0,548	0,568
	3	0,683	0,998	0,681	0,680	0,810
Multilayer Perception	5	0,970	0,585	0,978	0,585	0,734
	10	0,035	1,00	0,307	0,307	0,470
	3	0,245	0,966	0,726	0,729	0,829
Stacking	5	0,138	0,976	0,637	0,661	0,771
	10	0,012	0,984	0,527	0,724	0,686
	3	0,322	0,930	0,743	0,733	0,826
AdaBoostM1	5	0,203	1,000	0,586	0,586	0,739
	10	0,090	0,989	0,316	0,338	0,478
	3	0,229	0,968	0,727	0,732	0,831
Bagging	5	0,199	0,924	0,697	0,720	0,794
	10	0,010	0,989	0,468	0,652	0,636

Table 6.4: Results of experiment for various window-sizes on ADFA-LD data set with one-class labels in Cross-Validation mode (Fold=10)

Algorithm	window-size	FP Rate	Precision	Recall	Accuracy	F-Measure
	3	0,444	0,913	0,821	0,782	0,865
Naive Bayes	5	0,469	0,842	0,856	0,773	0,849
	10	0,516	0,693	0,901	0,719	0,783
	3	0,349	0,968	0,796	0,786	0,874
J48	5	0,344	0,925	0,839	0,809	0,880
	10	0,389	0,860	0,857	0,792	0,858
	3	0,345	0,971	0,794	0,785	0,873
JRip	5	0,074	0,999	0,762	0,763	0,864
	10	0,301	1,000	0,733	0,733	0,846
	3	0,655	0,880	0,783	0,721	0,829
IBK-k3	5	0,525	0,868	0,803	0,749	0,843
	10	0,498	0,849	0,800	0,734	0,824
	3	0,621	1,000	0,765	0,765	0,867
Multilayer Perception	5	0,460	0,997	0,759	0,758	0,862
	10	0,536	0,980	0,739	0,731	0,842
	3	0,320	0,974	0,795	0,787	0,875
Stacking	5	0,278	0,951	0,832	0,817	0,887
	10	0,322	0,883	0,883	0,828	0,883
	3	0,473	0,917	0,811	0,772	0,860
AdaBoostM1	5	0,429	0,891	0,836	0,785	0,863
	10	0,383	0,873	0,846	0,790	0,859
	3	0,396	0,954	0,801	0,784	0,871
Bagging	5	0,291	0,945	0,835	0,817	0,887
	10	0,297	0,907	0,863	0,826	0,885

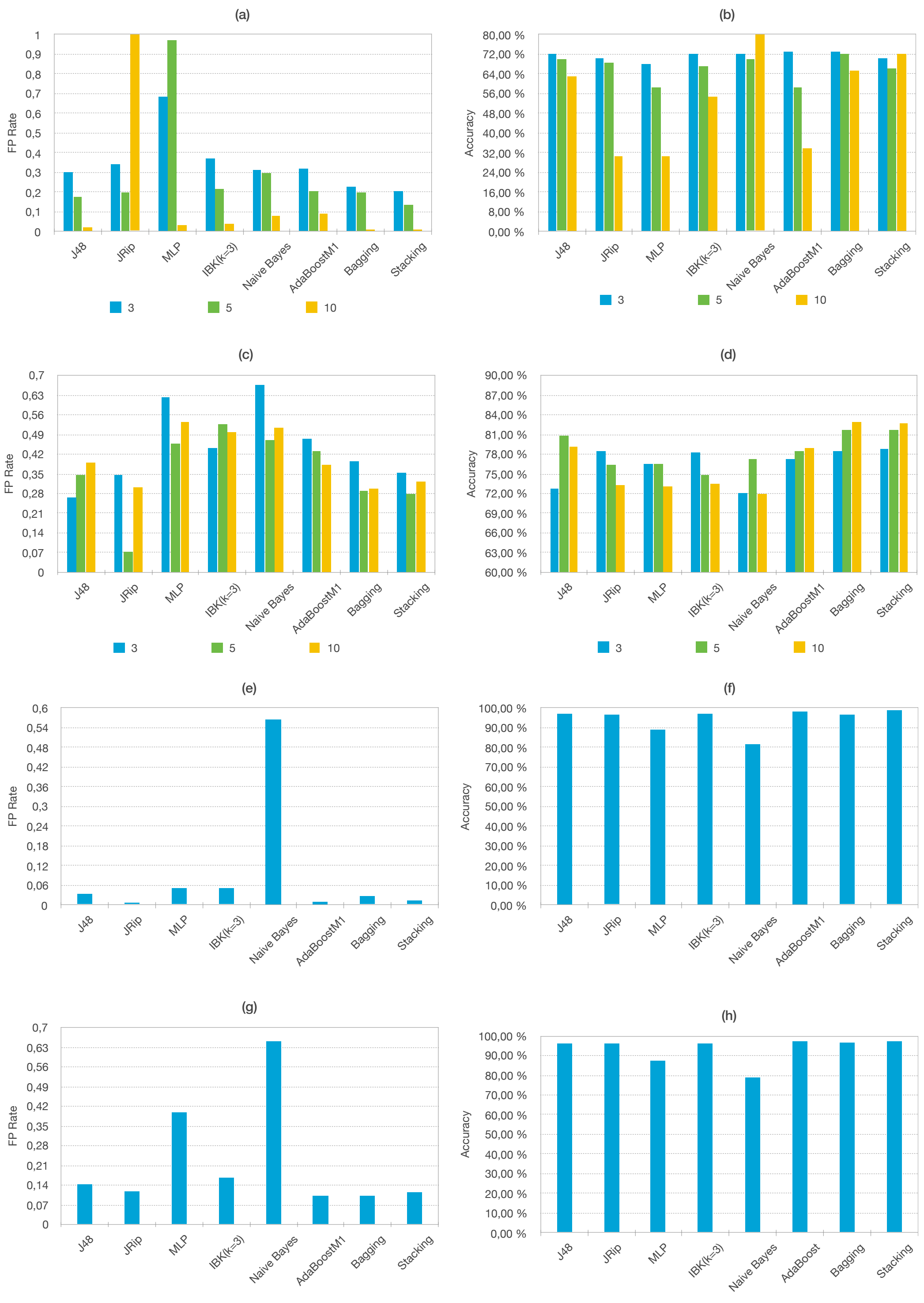


Figure 6.3: Achieved performance results including False Positive Rate and Accuracy on ADFA-LD data set. (a) FP rate for windows-based with supplied test data set; (b) Accuracy for windows-based with supplied data set; (c) FP rate for windows-based in Cross-Validation mode; (d) Accuracy for windows-based in Cross-Validation mode; (e) FP rate for frequency-based with supplied test data set; (f) Accuracy for frequency-based with supplied test data set; (g) FP rate for frequency-based in Cross-Validation mode; (h) Accuracy for frequency-based in Cross-Validation mode

7 Conclusion

7.1 Summary

In chapter 2, fundamentals and state of the art concerning building host-based intrusion detection systems were discussed. We considered two detection methods of HIDS precisely, including application whitelisting and observing system call traces. Moreover, the recently-generated ADFA data sets were introduced, which are used by many recent research works for testing HIDSs. Finally, we discussed various anomaly modeling techniques covering statistical models, immune system approaches, Markov process model and data mining techniques.

In chapter 3, functional and non-functional requirements of building host-based intrusion detection systems for embedded systems were discussed. We discussed the significance of each requirement by considering characteristics of embedded systems. Accordingly, we defined availability, accuracy, completeness, performance, security, scalability and maintainability as non-functional requirements. By contrast, collecting, analysis and response were defined as functional requirements of a host-based IDS.

In chapter 4, the architectural design of a new approach for building a host-based anomaly detection system (LIDS) was proposed. It was presented that the proposed concept comprises two main sub-components that can satisfy the defined requirements in chapter 3. Moreover, this chapter showed that the LIDS combines a simple approach of application whitelisting with another approach that analyses system call traces for detecting highly-intelligent attacks.

In chapter 5, information concerning the implementation of the LIDS was discussed. Accordingly, the project structure, implementation environment and important aspects of implementation of both sub-components were explained. We described the implementation of application whitelisting in the form of a Linux security module. Furthermore, the implementation of the second sub-component based on Java programming language with its UML diagrams was presented.

In chapter 6, the evaluation results of both sub-components were presented. For both approaches, the evaluation environment, evaluation method and results of tests were discussed. Accordingly, the first sub-component - which performs application whitelisting - achieved 100 percent detection accuracy and 0 percent false positives. For evaluating the second sub-component, various anomaly detection techniques based on two data representation methods - window- and frequency-based - were tested and the achieved performance results of various techniques were presented in the form of tables and graphical presentations.

7.2 Problems encountered and outlook

The most challenging issue with this work was the implementation of application whitelisting in the form of a Linux security module. First, it required deep knowledge of the Linux kernel programming, as well as how the Linux operating system deals with various kernel objects. Second, any changes required re-building the kernel, which in some cases was a time-consuming process. In order to solve this problem, the author used the Jprobe framework (Kernel Debugging Framework). Jprobe is a straightforward approach to probe the kernel. Fundamentally, it requires the address of a kernel function, which needs to be analyzed. Using Jprobe, we assigned a handler function that is called each time when the kernel function is invoked. Consequently, we could prove various control techniques without being compelled to re-build the kernel repeatedly.

7.3 Outlook

Future work will enhance both sub-components with new features. Watchdog (performs application whitelisting) will be enhanced to observe incoming and leaving network packets as well as monitoring sockets. More precisely, we will analyze which process creates and listens to which sockets, specify network protocols that are allowed to be used, monitor ports and control the admissibility of processes in sending and receiving network packets.
The second sub-component - Secure Homeland - will be enhanced to combine the two discussed data representation modes of window- and frequency-based to consider both sequence and frequency simultaneously. It should be noted that in chapter 2 we introduced an approach called *n-gram* vector space model [3] that combined both approaches, although according to our experiences this approach is not appropriate for an embedded environment. A Further enhancement for LIDS will be operation of Secure Homeland on the embedded devices. According to the concept that has been introduced in this work, system call traces of embedded devices are gathered in a server and analyzed by Secure Homeland. However, providing a secure communication channel between devices and the server prompts new complex challenges. Thus, from the author's perspective the design concept of Secure Homeland must be enhanced to be able for running on embedded devices.

List of Acronyms

ANN	Artificial Neural Network
AWL	Application Whitelisting
ECU	Electronic Control Unit
FP	False Positives
FN	False Negatives
HIDS	Host-based Intrusion Detection System
HMM	Hidden Markov Model
IDS	Intrusion Detection System
LIDS	Linux-based Intrusion Detection System
LSM	Linux Security Module
NIDS	Network-based Intrusion Detection System
OS	Operating System
SVM	Support Vectore Machine
TP	True Positives
TN	True Negatives

Bibliography

[1] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security modules: general security support for the linux kernel," in *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pp. 213–226, 2003.

[2] E. Viegas, A. O. Santin, A. Franca, R. Jasinski, V. A. Pedroni, and L. S. Oliveira, "Towards an energy-efficient anomaly-based intrusion detection engine for embedded systems," *IEEE Transactions on Computers*, vol. 66, pp. 163–177, Jan 2017.

[3] B. Borisaniya and D. Patel, "Evaluation of modified vector space representation using adfa-ld and adfa-wd datasets," vol. 6, p. 250, 07 2015.

[4] F. M. Tabrizi and K. Pattabiraman, "Intrusion detection system for embedded systems," in *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*, Middleware Doct Symposium '15, (New York, NY, USA), pp. 9:1–9:4, ACM, 2015.

[5] "Sektoren und branchen kritischer infrastrukturen." `http://www.kritis.bund.de/SubSites/Kritis/DE/Einfuehrung/Sektoren/sektoren_node.html`. Accessed: 2017-11-01.

[6] "Track vacancy detection." `http://w3.usa.siemens.com/mobility/us/en/rail-solutions/rail-automation/track-vacancy-detection/pages/track-vacancy-detection.aspx`. Accessed: 2017-11-01.

[7] P. Koopman, "Embedded system design issues (the rest of the story)," in *Computer Design: VLSI in Computers and Processors*, pp. 310–317, IEEE, 1996.

[8] P. Koopman, "Embedded system security," *Computer*, vol. 37, pp. 95–97, July 2004.

[9] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security and Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[10] R. Brandom, "Uk hospitals hit with massive ransomware attack," *The Verge*. Available at `https://www.theverge.com/2017/5/12/15630354/nhs-hospitals-ransomware-hack-wannacry-bitcoin`.

[11] S. Mohurle and M. Patil, "A brief study of wannacry threat: Ransomware attack 2017," *International Journal of Advanced Research in Computer Science*, vol. 8, no. 5, pp. 1938–1940, 2017.

[12] M. Erritali and B. E. Quahidi, "A review and classification of vanets intrusion detection systems," *Security Days (JNS3), 2013 National*, April 2013.

[13] C. Kruegel, *Intrusion Detection and Correlation: Challenges and Solutions*. Santa Clara, CA, USA: Springer-Verlag TELOS, 2004.

[14] V. Bukac, P. Tucek, and M. Deutsch, *Advances and Challenges in Standalone Host-Based Intrusion Detection Systems*, pp. 105–117. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[15] J. McHugh, "Intrusion and intrusion detection," *International Journal of Information Security*, vol. 1(1), pp. 14–35, August 2001.

[16] C. V. Zhou, C. Leckie, and S. Karunasekera, "A survey of coordinated attacks and collaborative intrusion detection," *Computers and Security*, vol. 29, no. 1, pp. 124 – 140, 2010.

[17] G. Creech and J. Hu, "Generation of a new ids test dataset: Time to retire the kdd collection," in *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 4487–4492, April 2013.

[18] S. Patil, A. Kashyap, G. Sivathanu, and E. Zadok, "Fs: An in-kernel integrity checker and intrusion detection file system," in *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, (Berkeley, CA, USA), pp. 67–78, USENIX Association, 2004.

[19] J. Kaczmarek and M. Wrobel, "Modern approaches to file system integrity checking," in *2008 1st International Conference on Information Technology*, pp. 1–4, May 2008.

[20] G. Sivathanu, C. P. Wright, and E. Zadok, "Ensuring data integrity in storage: Techniques and applications," in *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*, StorageSS '05, (New York, NY, USA), pp. 26–36, ACM, 2005.

[21] F. A. Barbhuiya, S. Roopa, R. Ratti, N. Hubballi, S. Biswas, A. Sur, S. Nandi, and V. Ramachandran, *An Active Host-Based Detection Mechanism for ARP-Related Attacks*, pp. 432–443. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[22] J. Hizver and T. c. Chiueh, "Cloud-based application whitelisting," in *2013 IEEE Sixth International Conference on Cloud Computing*, pp. 636–643, June 2013.

[23] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, (Washington, DC, USA), pp. 120–, IEEE Computer Society, 1996.

[24] J. Reeves, A. Ramaswamy, M. Locasto, S. Bratus, and S. Smith, "Intrusion detection for resource-constrained embedded control systems in the power grid," *International Journal of Critical Infrastructure Protection*, vol. 5, no. 2, pp. 74 – 83, 2012.

[25] M. Xie, J. Hu, and J. Slay, "Evaluating host-based anomaly detection systems: Application of the one-class svm algorithm to adfa-ld," in *2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pp. 978–982, Aug 2014.

[26] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Secur.*, vol. 6, pp. 151–180, Aug. 1998.

[27] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection for discrete sequences: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, pp. 823–839, May 2012.

[28] T. Lane and C. E. Brodley, "Temporal sequence learning and data reduction for anomaly detection," *ACM Trans. Inf. Syst. Secur.*, vol. 2, pp. 295–331, Aug. 1999.

[29] T. Lane and C. Brodley, "Sequence matching and learning in anomaly detection for computer security," 05 1997.

[30] A. K. Ghosh, J. Wanken, and F. Charron, "Detecting anomalous and unknown intrusions against programs," in *Proceedings 14th Annual Computer Security Applications Conference (Cat. No.98EX217)*, pp. 259–267, Dec 1998.

[31] G. Creech and J. Hu, "A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns," *IEEE Transactions on Computers*, vol. 63, pp. 807–819, April 2014.

[32] E. Eskin, A. Arnold, M. Prerau, L. Portnoy, and S. Stolfo, *A Geometric Framework for Unsupervised Anomaly Detection*, pp. 77–101. Boston, MA: Springer US, 2002.

[33] A. P. Kosoresow and S. A. Hofmeyer, "Intrusion detection via system call traces," *IEEE Software*, vol. 14, pp. 35–42, Sep 1997.

[34] B. Gao, H.-Y. Ma, and Y.-H. Yang, "Hmms (hidden markov models) based on anomaly intrusion detection method," in *Proceedings. International Conference on Machine Learning and Cybernetics*, vol. 1, pp. 381–385 vol.1, 2002.

[35] F. A. González and D. Dasgupta, "Anomaly detection using real-valued negative selection," *Genetic Programming and Evolvable Machines*, vol. 4, pp. 383–403, Dec. 2003.

[36] M. Wang, C. Zhang, and J. Yu, "Native api based windows anomaly intrusion detection method using svm," in *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)*, vol. 1, pp. 6 pp.–, June 2006.

[37] X. Li, J. Han, S. Kim, and H. Gonzalez, *ROAM: Rule- and Motif-Based Anomaly Detection in Massive Moving Object Data Sets*, pp. 273–284.

[38] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, (Washington, DC, USA), p. 156, IEEE Computer Society, 2001.

[39] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, (New York, NY, USA), pp. 255–264, ACM, 2002.

[40] J. Hu, X. Yu, D. Qiu, and H. H. Chen, "A simple and efficient hidden markov model scheme for host-based anomaly intrusion detection," *IEEE Network*, vol. 23, pp. 42–47, January 2009.

[41] "Process monitor (procmon)." `https://docs.microsoft.com/en-us/sysinternals/downloads/procmon`. Accessed: 2017-11-20.

[42] E. Aghaei, "Machine learning for host-based misuse and anomaly detection in unix environment," Master's thesis, 2017.

[43] M. Xie, J. Hu, X. Yu, and E. Chang, *Evaluating Host-Based Anomaly Detection Systems: Application of the Frequency-Based Algorithms to ADFA-LD*, pp. 542–549. Cham: Springer International Publishing, 2014.

[44] W.-H. Chen, S.-H. Hsu, and H.-P. Shen, "Application of svm and ann for intrusion detection," *Comput. Oper. Res.*, vol. 32, pp. 2617–2634, Oct. 2005.

[45] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys (CSUR)*, vol. 41(3), July 2009.

[46] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Secur.*, vol. 6, pp. 151–180, Aug. 1998.

[47] S. S. Murtaza, W. Khreich, A. Hamou-Lhadj, and S. Gagnon, "A trace abstraction approach for host-based anomaly detection," in *2015 IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA)*, pp. 1–8, May 2015.

[48] R. Moskovitch, S. Pluderman, I. Gus, D. Stopel, C. Feher, Y. Parmet, Y. Shahar, and Y. Elovici, "Host based intrusion detection using machine learning," in *2007 IEEE Intelligence and Security Informatics*, pp. 107–114, May 2007.

[49] P. Kabiri and A. A. Ghorbani, "Research on intrusion detection and response: A survey," *International Journal of Network Security*, vol. 1, pp. 84–102, 2005.

[50] A. Olmsted, "Secure software development through non-functional requirements modeling," in *2016 International Conference on Information Society (i-Society)*, pp. 22–27, Oct 2016.

[51] J. Eckhardt, A. Vogelsang, and D. M. Fernandez, "Are non-functional requirements really non-functional? an investigation of non-functional requirements in practice," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 832–842, May 2016.

[52] E. Amoroso and R. Kwapniewski, "A selection criteria for intrusion detection systems," in *Proceedings 14th Annual Computer Security Applications Conference (Cat. No.98EX217)*, pp. 280–288, Dec 1998.

[53] M. Mari and N. Eila, "The impact of maintainability on component-based software systems," in *2003 Proceedings 29th Euromicro Conference*, pp. 25–32, Sept 2003.

[54] F. L. Gaol, *Recent Progress in Data Engineering and Internet Technology*. Springer-Verlag Berlin Heidelberg, 2012.

[55] A. McIntyre, U. Lindqvist, B. Peterson, and Z. Tudor, "Host protection strategies for industrial control systems," in *2012 IEEE Conference on Technologies for Homeland Security (HST)*, pp. 87–92, Nov 2012.

[56] X. Wang and H. Yu, *How to Break MD5 and Other Hash Functions*, pp. 19–35. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[57] J. M. D. Goyeneche and E. A. F. D. Sousa, "Loadable kernel modules," *IEEE Software*, vol. 16, pp. 65–71, Jan 1999.

[58] "Weka." `http://www.cs.waikato.ac.nz/ml/weka/`. Accessed: 2017-11-20.

[59] H. Chauhan, V. Kumar, S. Pundir, and E. S. Pilli, "A comparative study of classification techniques for intrusion detection," in *2013 International Symposium on Computational and Business Intelligence*, pp. 40–43, Aug 2013.

[60] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Machine Learning*, vol. 6, pp. 37–66, Jan 1991.

[61] V. Singh and S. Puthran, "Intrusion detection system using data mining a review," in *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*, pp. 587–592, Dec 2016.

[62] H. Mitchell, *Ensemble Learning*, pp. 221–240. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[63] "Xcode." `https://developer.apple.com/xcode/`. Accessed: 2017-12-09.

[64] "Gcc." `https://gcc.gnu.org/`. Accessed: 2017-12-09.

[65] "Linux crypto api." `https://www.kernel.org/doc/html/v4.12/crypto/index.html`. Accessed: 2017-12-09.

[66] "Intellij." `https://www.jetbrains.com/idea/`. Accessed: 2017-12-09.

[67] "Apache maven." `https://maven.apache.org/`. Accessed: 2017-12-09.

[68] "Vmware workstation." `https://www.vmware.com/de/products/workstation-pro.html`. Accessed: 2017-12-09.

[69] "Raspberry pi zero w." `https://www.raspberrypi.org/products/raspberry-pi-zero-w/`. Accessed: 2017-12-09.

Annex A - List of the Kernel hooks

```
@Security hooks for program execution operations.

int (*bprm_set_creds)(struct linux_binprm *bprm);
int (*bprm_check_security)(struct linux_binprm *bprm);
int (*bprm_secureexec)(struct linux_binprm *bprm);
void (*bprm_committing_creds)(struct linux_binprm *bprm);
void (*bprm_committed_creds)(struct linux_binprm *bprm);

@Security hooks for task operations.

int (*task_fix_setuid)(struct cred *new, const struct cred *old,int flags);
int (*task_setpgid)(struct task_struct *p, pid_t pgid);
int (*task_getpgid)(struct task_struct *p);
int (*task_getsid)(struct task_struct *p);
void (*task_getsecid)(struct task_struct *p, u32 *secid);
int (*task_setnice)(struct task_struct *p, int nice);
int (*task_setioprio)(struct task_struct *p, int ioprio);
int (*task_getioprio)(struct task_struct *p);
int (*task_setrlimit)(struct task_struct *p, unsigned int resource,
            struct rlimit *new_rlim);
int (*task_setscheduler)(struct task_struct *p);
int (*task_getscheduler)(struct task_struct *p);
int (*task_movememory)(struct task_struct *p);
int (*task_kill)(struct task_struct *p, struct siginfo *info, int sig, u32 secid);
int (*task_wait)(struct task_struct *p);
int (*task_prctl)(int option, unsigned long arg2, unsigned long arg3,
            unsigned long arg4, unsigned long arg5);
void (*task_to_inode)(struct task_struct *p, struct inode *inode);
```

Listing 1: Security hooks for task and program execution operations

Annex B - The performance results of algorithms

Table .1: The performance results of algorithms including FP Rate, Accuracy, Confusion Matrix and the required time for learning and testing in window-based method (window-size=3) using a supplied test data set. Train data set contains 78406 instances, test data set contains 39845 instances

Algorithm	**FP Rate**	**Accuracy**	**TP**	**TN**	**FP**	**FN**	**Train**	**Test**
J48	0,300	0,721	25926	2791	1197	9931	4 s	60 ms
JRip	0,345	0,706	26006	2122	1117	10600	18 s	60 ms
Multilayer Perception	0,683	0,686	27080	20	43	12702	28 s	0,13 s
IBK-K3	0,315	0,723	25696	3099	1427	9623	0,100 s	2 m
Naive Bayes	0,368	0,723	24786	4014	2337	8708	0,41 s	0,09 s
Stacking	0,203	0,706	26776	1360	347	11362	12 m	13 s
AdaBoostM1	0,322	0,733	25228	3983	1895	8739	37,29 s s	2,6 s
Bagging	0,229	0,732	26264	2884	859	9838	2,43 m	3,45 s

Table .2: The performance results of algorithms including FP Rate, Accuracy, Confusion Matrix and the required time for learning and testing in window-based method (window-size=3) in Cross-Validation mode. Train data set contains 90504 instances

Algorithm	**FP Rate**	**Accuracy**	**TP**	**TN**	**FP**	**FN**	**Train**
J48	0,349	0,786	67008	4151	2223	17122	29,8 s
JRip	0,345	0,785	67230	3796	2001	17477	57 s
Multilayer Perception	0,621	0,765	69213	11	18	21262	37 s
IBK-K3	0,655	0,721	60908	4383	8323	16890	2,50 m
Naive Bayes	0,444	0,782	63229	7525	6002	13748	0,2 s
Stacking	0,353	0,787	66860	4337	2371	16936	2 h
AdaBoostM1	0,473	0,772	63465	6436	5766	14837	40 m
Bagging	0,396	0,784	66045	4867	3186	16406	30 m

Table .3: The performance results of algorithms including FP Rate, Accuracy, Confusion Matrix and the required time for learning and testing in window-based method (window-size=5) using supplied test data set. Train data set contains 240135 instances and test data set contains 86085 instances

Algorithm	**FP Rate**	**Accuracy**	**TP**	**TN**	**FP**	**FN**	**Train**	**Test**
J48	0,176	0,701	47706	12651	2695	23033	36,79 s	0,19 s
JRip	0,199	0,684	47577	11347	2824	24337	13,731 m	0,19 s
Multilayer Perception	0,970	0,585	49349	1052	34575	1109	12 m	0,15 s
IBK-K3	0,216	0,671	47607	10131	2794	25553	0,02 s	24,38 m
Naive Bayes	0,299	0,701	43009	17340	7392	18344	2,9 s	0,23 s
Stacking	0,138	0,661	49168	7718	1233	27966	7,9 min	0,65 s
AdaBoostM1	0,199	0,720	46561	15427	3840	20257	7,2 m	0,92 s
Bagging	0,199	0,720	45561	15427	3840	20257	40 s	0,45 s

Table .4: The performance results of algorithms including FP Rate, Accuracy, Confusion Matrix and the required time for learning and testing in window-based method (window-size=5) in Cross-Validation mode. Train data set contains 279612 instances.

Algorithm	**FP Rate**	**Accuracy**	**TP**	**TN**	**FP**	**FN**	**Train**
J48	0,344	0,809	195940	30229	15857	37586	68,19 s
JRip	0,074	0,763	211673	1550	124	66265	70 s
Multilayer Perception	0,460	0,758	211250	643	547	67172	1 h
IBK-K3	0,525	0,749	187646	21885	24151	45930	50 m
Naive Bayes	0,469	0,773	178372	37899	33425	29916	40 s
Stacking	0,278	0,817	201352	27174	10445	40641	2 h
AdaBoostM1	0,429	0,785	188701	30779	23096	37036	6,32 m
Bagging	0,291	0,817	200232	28129	11565	39686	4 m

Table .5: The performance results of algorithms including FP Rate, Accuracy, Confusion Matrix and the required time for learning and testing in window-based method (window-size=10) using supplied test data set. Train data set contains 568527 instances and test data set contains 131380 instances

Algorithm	**FP Rate**	**Accuracy**	**TP**	**TN**	**FP**	**FN**	**Train**	**Test**
J48	0,024	0,632	39279	43763	1062	47276	10 m	4 s
JRip	0,999	0,307	40341	46	0	90993	9 s	5,7 s
Multilayer Perception	0,035	0,307	40339	55	2	90984	9 m	9,6 s
IBK-K3	0,039	0,548	39002	32936	1339	58103	2 s	2 h
Naive Bayes	0,077	0,799	34419	70564	5922	20475	21 s	6,81 s
Stacking	0,012	0,724	39685	55399	656	35640	50,46 m	1,27 s
AdaBoostM1	0,090	0,338	39893	4513	448	86526	68,46 m	1,22 s
Bagging	0,010	0,652	39878	45761	463	45278	4,16 m	1,25 s

Table .6: The performance results of algorithms including FP Rate, Accuracy, Confusion Matrix and the required time for learning and testing in window-based method (window-size=10) in Cross-Validation mode. Train data set contains 668073 instances.

Algorithm	**FP Rate**	**Accuracy**	**TP**	**TN**	**FP**	**FN**	**Train**
J48	0,389	0,792	420965	108270	68807	70031	3,8 h
JRip	0,301	0,733	489704	158	68	178143	4 m
Multilayer Perception	0,536	0,731	480158	8329	9614	169972	1 h
IBK-K3	0,498	0,734	415988	74276	73784	104025	2h
Naive Bayes	0,516	0,719	339235	141006	150537	37295	4 m
Stacking	0,322	0,828	432380	120972	57392	57329	18 h
AdaBoostM1	0,383	0,790	427436	100292	62336	78009	3 h
Bagging	0,297	0,826	444048	108089	45724	70212	2,5 h

Table .7: The performance results of algorithms including FP Rate, Accuracy, Confusion Matrix and the required time for learning and testing in frequency-based method using supplied test data set. Train data set contains 3825 instances and test data set contains 5118 instances

Algorithm	**FP Rate**	**Accuracy**	**TP**	**TN**	**FP**	**FN**	**Train**	**Test**
J48	0,032	0,972	4351	626	21	120	5 s	0,5 s
JRip	0,005	0,967	4369	578	3	168	1,18 s	0,5 s
Multilayer Perception	0,050	0,893	4361	211	11	535	26 m	6,4 s
IBK-K3	0,051	0,969	4339	618	33	128	0,50 s	31,2 s
Naive Bayes	0,567	0,816	3539	635	833	111	0,94 s	1,34 s
Stacking	0,014	0,987	4362	688	10	58	1,5 m	1,4 s
AdaBoostM1	0,008	0,982	4367	659	5	87	30,130 s	0,6 s
Bagging	0,026	0,967	4356	595	16	151	35 s	0,7 s

Table .8: The performance results of algorithms including FP Rate, Accuracy, Confusion Matrix and the required time for learning and testing in frequency-based method in Cross-Validation mode. Train data set contains 5951 instances.

Algorithm	**FP Rate**	**Accuracy**	**TP**	**TN**	**FP**	**FN**	**Train**
J48	0,141	0,961	5401	613	101	133	3,8 s
JRip	0,120	0,959	5126	580	79	166	14 s
Multilayer Perception	0,400	0,875	5201	6	4	740	1,23 h
IBK-K3	0,168	0,959	5078	627	127	119	18 s
Naive Bayes	0,649	0,780	3977	665	1228	81	1 s
Stacking	0,115	0,974	5116	682	89	64	31 m
AdaBoostM1	0,101	0,970	5133	640	72	106	9 m
Bagging	0,103	0,968	5133	626	72	120	9m