CAN THE BEST DEFENSE BE A GOOD OFFENSE? EVOLVING (MIMICRY) ATTACKS FOR DETECTOR VULNERABILITY TESTING UNDER A 'BLACK-BOX' ASSUMPTION

by

Hilmi Güneş Kayacık

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
March 2009

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "CAN THE BEST DEFENSE BE A GOOD OFFENSE? EVOLVING (MIMICRY) ATTACKS FOR DETECTOR VULNERABILITY TESTING UNDER A 'BLACK-BOX' ASSUMPTION" by Hilmi Güneş Kayacık in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dated: March 20, 2009

External Examiner: Dr. Anil Somayaji

Research Supervisors: Dr. Nur Zincir-Heywood

Dr. Malcolm Heywood

Examining Committee: Dr. Qigang Gao

Dr. Denis Riordan

DALHOUSIE UNIVERSITY

DATE: March 20, 2009

AUTHOR: Hilmi Güneş Kayacık

TITLE: CAN THE BEST DEFENSE BE A GOOD OFFENSE? EVOLVING (MIMICRY) ATTACKS FOR DETECTOR VULNERABILITY TESTING UNDER A 'BLACK-BOX' ASSUMPTION

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: PhD CONVOCATION: May YEAR: 2009

Table of Contents

List of Tables

List of Figures

Abstract

This thesis proposes a 'black-box' approach for automating attack generation by way of Evolutionary Computation. The proposed 'black-box' approach employs just the anomaly rate or detection feedback from the detector. Assuming a 'black-box' access in vulnerability testing presents a scenario different from a 'white-box' access assumption, since the attacker does not posses sufficient knowledge to constrain the scope of the attack. As such, this thesis contributes by providing a 'black-box' vulnerability testing tool for identifying detector weaknesses and aiding detector research in designing detectors which are robust against evasion attacks.

The proposed approach focuses on stack buffer overflow attacks on a 32-bit Intel architecture and aims to optimize the various characteristics of the attack. Three components exist in a common stack buffer overflow attack: the shellcode, NoOP and return address components. Therefore, automation of attack generation is realized in three stages: (1) identifying the suitable NoOP and return address components, (2) designing the shellcode at the assembly level, and (3) designing the shellcode at the system call level. The first and second stage address the evasion of misuse detectors by employing obfuscation, whereas the third stage addresses the evasion of anomaly detectors by employing mimicry attacks.

In short, the proposed approach takes the form of a 'black-box' search process where the attacks are rewarded according to two main criteria: (a) their ability to carry out the malicious intent, while (b) minimizing or eliminating the detectable attack characteristics. Furthermore, it is demonstrated that there are two parts to buffer overflow attacks: (i) the preamble and (ii) the exploit. Therefore, the anomaly rate of the whole attack is calculated on both parts. Additionally, the proposed approach supports multi-objective optimization, where multiple characteristics of attacks can be improved. The proposed approach is evaluated against six detectors and four vulnerable applications. The results show that attacks which the proposed approach generates under a 'black-box' assumption are as effective as the attacks generated under a 'white-box' assumption adopted by previous work.

List of Abbreviations and Symbols Used

EC	Evolutionary Computing
EIP	Extended Instruction Pointer
FSA	Finite State Automata
GA	Genetic Algorithm
GCC	GNU C Compiler
GDB	GNU Project Debugger
GE	Grammatical Evolution
GP	Genetic Programming
IDS	Intrusion Detection System
NoOP	No OPeration instruction
pH	Process Homeostasis
pHsm	pH with a schema mask
SVM	Support Vector Machine

Acknowledgements

I would like to thank my supervisors Dr. Nur Zincir-Heywood and Dr. Malcolm Heywood for their continuous guidance and support throughout my graduate studies. I consider myself to be tremendously lucky to have worked with them. Although my PhD studies may be coming to an end, they will always remain as my mentors.

Furthermore, my sincere thanks are due to my committee members, Dr. Denis Riordan, Dr. Qigang Gao and my external examiner Dr. Anil Somayaji for their valuable feedback, which undoubtedly improved the quality of this thesis.

I was fortunate to be funded by numerous organizations. In particular, I wish to express my gratitude to Canada Foundation for Innovation (CFI), Killam Trusts, Mathematics of Information Technology and Complex systems (MITACS), Natural Sciences and Engineering Research Council of Canada (NSERC), Pre-Competitive Advanced Research Network (PRECARN), SwissCom Innovations AG and Telecom Applications Research Alliance (TARA).

My final and most heartfelt thanks go to my mother Rüveyde Kayacık and my father Zeki Kayacık for always being there for me. Words can't express my gratitude to them for their love, encouragement and support. This thesis is dedicated to them.

Chapter 1

Introduction

Software vulnerabilities have been around since the existence of software. One of the more memorable consequences of software vulnerabilities was the Morris Worm in 1988, which exploited various vulnerabilities in UNIX TCP/IP software [71] and brought down over 6000 servers across the Internet and caused a major disruption [85]. On the SecurityFocus (former Bugtraq) website, the vulnerabilities which the Morris worm exploited were the first to be documented[1]. As of this writing (January 2009), SecurityFocus vulnerability archives have over 33,000 posts (based upon the Bugtraq IDs), which implies that tens of thousands of software vulnerabilities have been found (and some have been widely exploited) over the last twenty years.

In general, attacks have one to one relationships with software vulnerabilities since the exploit mechanism depends on the vulnerability type. However, if the software vulnerability is present on multiple platforms (e.g. same web server running on different operating systems), multiple versions of an attack can be developed to exploit the same vulnerability on different platforms.

Various software vulnerability types exist such as design flaws, buffer overflows, format string attacks, input validation errors and race conditions [80]. Among these, buffer overflow is one of the major causes of vulnerabilities [12] and is of interest in this thesis. Buffer overflows are the result of software deficiencies due to programmer errors such as the improper handling of memory, inputs and outputs. For the attackers, buffer overflows are an important tactic which leads to various serious threats such as vandalism, fraud, identity theft, intellectual property theft, or many other types of crime.

Software testing [78] [42] is crucial for identifying and addressing the problems

[1]The SecurityFocus vulnerability archives can be accessed at `http://www.securityfocus.com/`. A search for the vulnerability with Bugtraq ID 1 will bring up the vulnerability which the Morris worm exploited.

in the software before it is released. Software testing can be defined as an empirical investigation conducted to gather information on the quality of the software. It involves various testing methods such as volume testing, stress testing, storage testing or vulnerability testing. In particular, vulnerability testing aims to find and eliminate as many vulnerabilities as possible within the limitations of time and resources. However, finding and eliminating every vulnerability in software is very difficult because the tester would have to provide *every* possible input and follow *every* execution path [78]. Even though the test eliminates numerous vulnerabilities, it does not guarantee that other vulnerabilities unknown to the tester do not exist [74], therefore the prudent strategy is to deploy defensive techniques to prevent attackers from exploiting a vulnerability, should they find one.

Today's Defensive Techniques

Depending upon their function, defensive techniques can be categorized into two groups as discussed below.

1. **Static defenses:** Static defense techniques are analogous to the fences around the premises of a building. In other words, static defensive techniques are intended to provide barriers to attacks. Keeping operating systems and other software up-to-date and deploying firewalls at entry points are examples of static defense solutions. Frequent software updates can remove the software vulnerabilities which are susceptible to exploits. Firewalls provide access control at the entry point to keep intruders out rather than catching them. Therefore, they function in much the same way as a physical gate on a house. Static defense techniques are the first line of defense; they are relatively easy to deploy and provide significant defense improvement compared with the initial unguarded state of the computer network. Moreover they act as the foundation for more sophisticated defense techniques.

2. **Dynamic defenses:** It is safe to assume that no system is totally foolproof and that attackers are always one step ahead in finding security holes in current systems. Furthermore, although software updates eliminate some of the

existing software vulnerabilities, they may introduce new vulnerabilities to the software. Therefore, dynamic defenses accompany the static defenses to provide comprehensive information about the state of the computer networks and connected systems. Dynamic defense techniques are analogous to burglar alarms, which monitor the premises to find evidence of break-ins. Built upon static defense techniques, dynamic defense operations aim to catch the attacks and log information about the incidents such as the source and nature of the attack. Intrusion detection systems are examples of dynamic defense techniques [20]. An intrusion detection system (IDS) is a combination of software and hardware, which collects and analyses data from networks and hosts to determine if there is an attack [1]. Different detection techniques can be employed to search for evidence of intrusions. To this end, two major categories exist for detection techniques: misuse and anomaly detection.

(a) **Misuse detection:** Misuse detection systems use *a priori* knowledge on attacks to look for traces of attacks. In other words, they detect intrusions by employing a description of the misuse [51]. Signature (rule)-based systems are the most common examples of misuse detection systems. In signature-based detection, attack signatures are sought in the monitored resource. Signature-based systems, by definition, are very accurate for known attacks which are included in their signature database. Moreover, since signatures are associated with specific misuse behaviour, it is easy to determine the attack type. However, their detection capabilities are limited to those within the signature database. As new attacks are discovered, a signature database requires continuous updating to include the new attack signatures.

(b) **Anomaly detection:** Anomaly detectors adopt the opposite approach, which is, to know what is normal, and then find the deviations from normal behaviour. These deviations are considered as anomalies or possible intrusions. Anomaly detection systems rely on knowledge of normal behaviour to detect any attacks. Therefore, attacks – including unknown (0-day) attacks – are detected as long as the attack behaviour deviates sufficiently

from the normal behaviour. However, if the attack is similar to the normal behaviour, it may not be detected. As opposed to misuse detection, it is difficult to associate deviations with specific attacks since the detection is based upon the deviation from normal behaviour, not upon the similarity to an attack signature. As the users change their behaviour, normal behaviour should be redefined to ensure effective detection and low false positives.

Who Defends the defenses?

The defense methods discussed above, including the detectors, are by no means infallible. Software vulnerabilities and hardware faults can cause them to malfunction. In addition to traditional software errors, detectors are also susceptible to detector-specific vulnerabilities such as misconfigurations, blind-spots and deficiencies in detection methodology. Sophisticated attackers try to deploy attacks without getting detected. To this end, they use these detector vulnerabilities and alter their actions to evade detection, rendering the detector ineffective.

In the case of misuse detectors, the attackers analyse the signatures and create evasion attacks to avoid triggering the signatures. Alternatively, they generate smoke-screen activity to increase the false alarms, hence overwhelming the detector and the system administrator. In the case of anomaly detectors, the attackers camouflage their attacks to resemble normal behaviour.

Detector vulnerability testing is important for identifying and eliminating detector weaknesses before attackers can get a chance to exploit them. It is a relatively new area compared with similar tests on cryptographic protocols. Such vulnerability testing efforts can be considered as 'white-hat' (i.e. ethical) hacking, where the objective is to establish the limitations of detection methodologies and to find detector vulnerabilities before the attackers. Knowing the limitations of the defenses at hand enables the defenders to make better decisions on the design of the defenses which protect the networks and the connected hosts.

1.1 Motivation and Objectives

Although the vulnerability testing methodology may differ based upon the target detector to be tested, the common task is to craft the inputs to the detector carefully in order to observe the false positive and detection rates. If the detector generates many false positives when there is no attack, this implies that the detector configuration needs to be revised to reduce false alarms. Similarly, if the detector does not detect the attack when the attack is deployed, this implies that the detection technique should be revised to be more sensitive to the attack. In particular, this thesis focuses on vulnerability testing of the detectors on evasion attacks, where the objective of the 'white-hat' attackers is to modify the attack to make their actions go unnoticed by both misuse and anomaly detectors.

Various techniques exist for generating evasion attacks [100]. The attacker can try to avoid detection by expanding the attack over time or space in the input stream of the detector to make it more difficult for the malicious events to be correlated. Against misuse detectors, the attacker can obfuscate the attack so that the attack payload does not match any detection signatures. Alternatively, the attackers can take advantage of ambiguities in the network protocols to deploy their attacks or deploy a denial of service attack against the detector to overwhelm it. Against anomaly detectors, the attackers can deploy mimicry attacks, in which the attacks mimic the 'normal' definition of the application behaviour or 'valid' network traffic to evade detection. Within this categorization, the 'black-box' framework proposed in this thesis is employed as a technique for obfuscation and mimicry attack generation. The term 'black-box' means that the detector is viewed in terms of its input and output without any knowledge on its internal workings. Conversely, 'white-box' implies that the internal workings of the detector are available.

Previously, several researchers have identified a number of evasion attacks on network-based detectors employing misuse detection [82] [73] [102] [48] and host-based detectors employing anomaly detection [105] [99] [96] [98] [32] [56] [34].

Research on evading misuse detectors aims to minimize or entirely eliminate the components which trigger detection signatures by employing a 'white-box' approach.

Previous efforts [82] [73] [102] on evasion attacks against misuse detectors have employed various techniques such as packet splitting or polymorphism to encrypt the attack payload. This thesis addresses the evasion of misuse detectors by obfuscating the payload and by minimizing the detectable characteristics of the attacks.

In terms of evading anomaly detection, the evasion approach proposed in this thesis is based upon the generation of mimicry attacks. A mimicry attack is an exploit crafted by producing a legitimate sequence of system calls while performing malicious actions, typically by making use of a template denoting the original core attack [56]. In terms of evading anomaly detectors, previous efforts [105] [99] [96] [98] [32] [56] [34] assumed a 'white-box' access to the target detector. This has resulted in very efficient algorithms for designing mimicry exploits. In particular, such research has for the most part concentrated on the Stide open source host-based anomaly detector [30] or its improved versions [89] [27] [103] [110] [26]. The design of exploits then boils down to locating sequences of system calls which both match the contents of an anomaly detector's normal behaviour database whilst reaching the behavioural objectives of the original 'core' exploit. The behavioural objectives of the 'core' exploit can be considered to be general and thus more synonymous with a general behavioural objective. For example, the attacker might have the 'core' objective of opening the password file creating a user account and closing the file. From the perspective of most anomaly detection systems, this process amounts to the detection of an open-write-close sequence of system calls – a process which will be demonstrated later on in this thesis to be rather difficult to achieve without compromising the ability of the detector to avoid misclassification of normal behaviours.

Can an attacker evade detection using a 'black-box' access?

Assuming a 'white-box' approach implies that the 'internal' knowledge necessary to design such an evasion methodology is extensive and may not be available for all detectors, such as in the case of commodity detectors or in the case where the detectors have complex normal behaviour databases (e.g. neural networks with numerous neurons and associated weight values). Even though it is fairly realistic to assume that the attacker can obtain a copy of the detector, obtaining internal knowledge

such as the detection methodology, a snapshot of the normal behaviour database or detector parameters may not always be possible. Furthermore, a 'white-box' analysis places emphasis on the analyser to define the problem based upon his/her knowledge of the detector and its internal data structures. Therefore, 'white-box' testing has the potential to introduce bias into the test methodology [81] [21].

Can an attacker evade detection completely?

The relevant work on vulnerability testing of anomaly detectors [105] [99] [96] [32] [56] [34] assumed that either (1) at the break-in stage, attackers can gain control of the vulnerable application without raising alarms, which reaches to a level which results in detection, or (2) attackers have already gained control of the victim system. Although the attackers can alter their exploit after gaining full control, no consideration was given as to whether the combination of the break-in and exploit would increase the anomaly rate [46] [47]. In this thesis, the break-in stage, during which the attacker tries to gain control of the vulnerable application, is called the preamble. After the attacker gains control of the application, he/she injects a payload, which is called the exploit, to carry out the attack objectives such as spawning a UNIX shell. Therefore, mimicry attacks are comprised of two components: the preamble and the exploit. Thus, the anomaly rate is calculated not only on the exploit but also on the entire attack, which contains the anomalies from the preamble. As such, the performance evaluation reflects the ability of the detector to detect attacks as a whole (as would be the case in practice) as opposed to limiting evaluation to detecting the exploit alone; the latter biasing the evaluation in favour of the attacker and giving a misleading impression of detector vulnerability to mimicry exploits.

Is it sufficient to optimize a single objective?

The relevant efforts on evading anomaly detectors focus mainly on reducing the anomaly rate of an attack. However, anomaly detectors such as Process Homeostasis (pH) [91] look not only at the number of anomalous events but also at the distribution of the anomalous events. Furthermore, the attacker may have additional constraints

due to the size of the vulnerable buffer and the nature of the vulnerability. Therefore detector vulnerability testing involves optimizing numerous characteristics and achieving multiple attack objectives. The approach proposed in this thesis can support the optimization of multiple objectives of an attack and therefore can lead to a more realistic vulnerability analysis.

The contributions of this thesis discussed in the next section aim to answer the above-stated questions while addressing the prevalent issues with detector vulnerability testing.

1.2 Contributions

As discussed in Section 1.1, the objective of the 'white-hat' attacker is to craft the input to the detector with the purpose of evading detection. If the 'white-hat' attacker can find an input with better detection characteristics (i.e. evading the detector completely or improving the chances of evasion by reducing the anomaly rates and eliminating the detectable attributes), it implies that the detector is susceptible to evasion attacks. Therefore the contributions of this thesis can be discussed in six categories.

An Artificial Arms Race

The proposed approach represents an arms race between artificial 'white-hat' attackers and numerous detectors. The arms race rewards the attacker as it builds successful attacks which can defeat the target detector. In such an arms race, the detector responds to the attacks by providing a detection feedback in the form of anomaly rates or other detection information such as number of alerts. Consequently, the attacker utilizes the detection feedback to build evasion attacks which achieve the objectives of the attacker while minimizing the detection from the target detector. The main product of the arms race is a set of evasion attacks which can evade the target detector. The resulting attacks provide the defenders crucial information which can be utilized to eliminate the weaknesses of the target detector.

Access to the Detector

As opposed to the previous research which assumed a 'white-box' approach against anomaly detectors [105] [99] [96] [98] [32] [56] [34], this thesis assumes a 'black-box' approach to evasion attack generation. As such the feedback from the detector is limited to the detector outputs such as the anomaly rate and the delay, which are readily provided to the user as part of normal operation (typically for the establishment of thresholds to optimize false positive and detection rates under real world conditions). Hence no use is made of the internal data structures or algorithmic details specific to a particular detector. Design of exploits takes the form of a search process in which the anomaly rate or the detection information from the detector is used as the only guide to the effectiveness of the exploit. To do so efficiently requires a greater emphasis on the deployment of suitable stochastic search processes, whereas under the 'white-box' model, an exhaustive (greedy) search is sufficient given the availability of suitable *a priori* information to direct the search process against, for example, a specific data structure internal to the detector.

Consequently, the proposed 'black-box' approach provides a framework in which numerous detectors can be tested without the need for utilizing internal knowledge of the detector. Thus, this thesis expands the applicability of vulnerability testing beyond the limited scenario in which the internal knowledge of the detector is known. Furthermore, such a 'black-box' approach can be utilized for the analysis of the detector parameterization. In particular, when anomaly detectors are deployed on different platforms, the suitable configuration parameters, which provide the best attack detection with minimum false alarms must be determined. To this end, the proposed 'black-box' approach can be deployed against different detector configurations to determine suitable configuration parameters for deploying detectors, such as the proper sliding window lengths or training sets.

Analysis of 'Normal Behaviour'

In the case of evading anomaly detectors, the attacker aims to camouflage the malicious code by altering the attack so that the anomaly detector recognizes it as normal behaviour. Although the definition of 'normal behaviour' is not a straightforward

task, in simple terms, it is how the application behaves during normal operation. Given that the objective of the attacker is to hide the true intention of the attack within the normal behaviour of the vulnerable application, it is crucial to study the impacts of the application characteristics and vulnerability attributes. Depending upon the application, the normal behaviour model may be concise or extensive. Furthermore, when anomaly detectors are deployed in practice, a detection threshold should be identified to reduce false alarm rates. If the anomaly rate is below the detection threshold, the trace is considered to be 'normal.' The discussion of normal behaviour is fairly brief in the previous work on evading anomaly detectors [105] [99] [96] [98] [32] [56] [34] because finding a vulnerability in a concise normal behaviour model is more difficult than finding a vulnerability in an extensive normal behaviour model.

This thesis contributes to the research field by providing methods for analysing normal behaviour models. Such analysis can assist the identification of the elements in the normal behaviour model which an attacker can exploit, such as the existence of open-write-close sequences. Thus, the defenders can deploy additional security measures proactively around these potentially unsafe elements to prevent them from being exploited in an evasion attack.

Evaluation of the Attacks

The previous work on evading anomaly detectors [105] [99] [96] [98] [32] [56] [34] focused on the design of exploits alone without analysing it as a whole. This thesis argues that, from the perspective of an anomaly detector, there are two stages of an attack: (1) the break-in (i.e. the preamble) and (2) the exploit. Previous work reported exploits which can bypass detection completely (with a 0% anomaly rate) without considering the system calls executed during the break-in stage. However, system calls executed before the exploit (i.e. the break-in) also raise alarms, hence the anomaly rate for an attack should be calculated over the system calls executed during both the break-in and the execution of the exploit. Additionally, previous work did not consider exploit size as a parameter in attack generation; hence attack length was not deemed relevant.

The contribution of this thesis includes an attack evaluation strategy, which takes the preambles into account. By establishing realistic metrics for the success or the severity of attacks, this thesis aims to emphasize the potential trends which the attackers may follow such as deploying longer exploits to reduce the effects of anomalous preambles. By focusing on more realistic threats, researchers can develop methods to detect the attacks before the attack is deployed.

Analysis of the Attacks

In this thesis, the attacks generated against anomaly detectors are tested not only on the detectors which they target (as in previous work [105] [99] [96] [98] [32] [56] [34]) but also on numerous other anomaly detectors employed in this research. Such an analysis aims to reveal whether the generated attacks are detector specific or can evade multiple detectors successfully. Needless to say, an evasion attack which can evade multiple detectors poses a more serious risk than a detector specific evasion attack. Furthermore, given that, in the main, the previous work assumed 'white-box' access to the detector, this thesis provides a comparison between two scenarios, namely both 'black-box' and 'white-box' access to the detector.

The evasion attacks generated with the proposed approach contribute by providing a means for generating 'attack datasets', which can be employed in numerous ways to improve computer system defenses. For example, given a set of evasion attacks against a particular detector, the detector developer can analyse the evasion attacks to identify detector weakness. Based upon the analysis results, the detector developer can adjust the detection technique and retrain the detector to improve detection and reduce false alarms. Consequently, the defenders can be more proactive and develop detectors which are robust against the variants of an attack.

Multi-objective Optimization

This thesis employs Evolutionary Computation, which provides support for optimizing multiple characteristics of an attack besides the anomaly rate, such as the attack length and the dispersion of anomalies. Dispersion of anomalies is employed in anomaly detectors (as a locality frame count) to delay the execution of a process

because clustered anomalies are more likely to be intrusions than scattered anomalies. Future detector vulnerability testing efforts should investigate multiple characteristics of an attack such as the anomaly rate of the break-in and the dispersion of the anomalies throughout the attack.

Support for multi-objective optimization contributes by providing a means for generating attacks which can improve various attack characteristics. This is especially important where the vulnerability testing extends beyond minimizing the detection rate and incorporates additional metrics such as exploit lengths and attack delays. In such vulnerability testing scenarios the tester can examine how the incorporated metrics affect the success of the evasion attacks and improve the detection techniques accordingly.

1.3 Organization of the Thesis

The 'black-box' approach proposed in this thesis generates stack buffer overflow attacks automatically. Therefore, stack buffer overflow attacks are introduced in Chapter 2, with the emphasis on identifying the components of stack-based buffer overflow attacks and the key characteristics which the attackers aim to enhance in order to evade detection. Furthermore, Chapter 2 provides an overview of the characteristics that the proposed approach aim to improve to evade detection.

Chapter 3 details the relevant work in two categories: the previous research on detectors, Section 3.1, and the previous research on evading detectors, Section 3.2. The research in detectors is categorized further in terms of detection methodologies, namely, misuse detection in Section 3.1.1 and anomaly detection in Section 3.1.2. Previous research on evading detectors is discussed in Section 3.2.

Chapter 4 discusses the detectors which are utilized in the experiments for this thesis and includes both misuse and anomaly detectors. Snort was employed as the misuse detector in this thesis, Section 4.1. Anomaly detectors employed in this thesis monitor the sequences of system calls which an application makes to detect intrusions, namely, Stide, pH, pH with a schema mask (pHsm), the Markov Model and Neural Network detectors (Section 4.2).

Chapter 5 introduces the Evolutionary Computation concepts with a generic Evolutionary Computation model in Section 5.1. Furthermore, it provides an overview of the Evolutionary Computation algorithms, which are employed in the experiments, (namely, Grammatical Evolution in Section 5.2 and Linear Genetic Programming in Sections 5.3 and 5.4), without focusing on a particular problem. The algorithm discussions provide an overview of the representation, training schemes, stochastic search operators and fitness calculation, while the problem-specific details are left to Chapters 6, 7 and 8.

Based upon the buffer overflow characteristics provided in Section 2.3, Chapter 6 focuses on improving the buffer overflow characteristics, namely, NoOP and return address components, using Grammatical Evolution (GE). Furthermore Chapter 6 provides the definition of the GE grammar, fitness evaluations of the malicious buffers and a discussion of search space size for the given problem.

Chapter 7 expands the proposed framework by focusing on improving the shellcode of the buffer overflow attack. The improvements are made at the assembly level by inserting obfuscation code and discovering alternate ways for achieving the attack goals of evading misuse detectors . For this purpose, a Linear Genetic Programming (GP) approach is employed for evolving the buffer overflow attack payload. Furthermore, Chapter 7 addresses the identification of an appropriate instruction set, fitness calculations and provides a discussion of the search space size.

Even though the evasion attacks generated in Chapters 6 and 7 evade the misuse detectors, an anomaly detector monitoring application behaviour can detect the attack by observing the deviation in application behaviour during the attack. Chapter 8 continues on improving the shellcode of the buffer overflow attack but the improvements are made at the system call level. Thus, the objective is to craft a shellcode which executes a sequence of system calls which conforms to the normal database of the detector while achieving the attack goals. These attacks are also called mimicry attacks since the attack mimics the legitimate use of the application, as defined by the normal behaviour model. In particular, Linear Genetic Programming with Pareto Ranking is employed to evolve sequences of system calls which (1) contain the malicious sequence and (2) minimize the anomaly rate and improve other characteristics.

In addition to the instruction set details, discussion of fitness calculations, detector training set analysis and search space analysis, Chapter 8 also demonstrates that there are two parts to a buffer overflow attack and the anomaly rates should be calculated for the attack as a whole.

Chapter 9 investigates the anomaly rate of the mimicry attacks when they are trained against a specific detector and deployed against other detectors. The purpose of such an investigation in Section 9.1 is to determine whether the mimicry attacks generated against one detector can generalize to other detectors sharing similar detection techniques. Furthermore, Section 9.2 provides in-depth analysis of the mimicry attacks generated by not only the proposed methodology in Chapter 8 but also the 'white-box' methodologies discussed in relevant mimicry attack work. In Section 9.3, the mimicry attacks generated with the 'black-box' approach proposed in this thesis are compared against the mimicry attacks generated by a representative 'white-box' approach with the purpose of identifying similarities and differences in the resulting attacks.

Chapter 10 analyses the mimicry attacks generated by the proposed 'black-box' approach from the perspective of vulnerable application characteristics. In Section 10.1, the vulnerable applications and the normal databases generated by the anomaly detectors are examined to identify the impact of 'normal behaviour' on the generated mimicry attacks. Moreover, in Section 10.2, mimicry attacks generated by the proposed 'black-box' approach are examined to discern system calls which the attacks employ to hide the malicious intent, which – as Chapter 10 establishes – depends upon the vulnerable application.

Finally, conclusions are drawn and future research directions are discussed in Chapter 11.

Chapter 2

Stack Overflows

This thesis focuses on stack overflows, mainly because (1) stack overflow attacks are fairly established in the evasion attack research and (2) the ground truth of these vulnerabilities are available in detail in the current literature [2] [3] [4] [5]. It is worth emphasizing that the proposed approach relates to a wider scope of attacks, where the objective of the attacker is to improve the code which he/she injects to evade detection.

2.1 Definition

In programming languages such as C, data integrity checks are minimal for performance reasons. Although this increases efficiency and provides more control to the programmer, it also means that the programmer is responsible for making sure that the memory allocated for a variable is sufficient. If the data copied into a variable is more than the variable can hold, the excess data spills into the unallocated memory space or into other allocated variables. If a critical variable is overwritten, the program will crash or behave unexpectedly. The techniques for exploiting a buffer overflow vulnerability depends upon the operating system, system architecture and where the vulnerable variable resides in the memory such as the stack or the heap. This thesis focuses on stack buffer overflows in Intel 32 bit architectures running standard Linux operating systems.

The stack is a first-in-last-out data structure which the programs use to store function variables and relevant information on the caller of the function. A stack frame contains all the variables that the function allocates. Furthermore, the program stores the information on what to do next after the function returns. Specifically, this information is the value of the EIP (extended instruction pointer) register before the function runs, which points to the next instruction in the program. This information

is also called the return address, since it determines the execution path after the function returns. When the function returns, the variables which are stored on the stack are de-allocated and the program uses the EIP value stored in the stack to determine the next instruction.

The reason that stack overflows are hazardous is that the stack grows toward the lower memory addresses (e.g. the first variable at address `0xAA55` and the second variable at address `0xAA44`) whereas the memory copy operation works toward the higher addresses (e.g. if the first byte is copied to address `0xAA44` the second byte is copied to address `0xAA45`). This implies that an unchecked memory copy can overwrite what is already stored on the stack. If the overflow resulting from an unchecked memory copy overwrites the return address on the stack, the execution can be diverted to an arbitrary code.

2.2 Components of Stack-based Buffer Overflow Attacks

In order to deploy a stack-based buffer overflow attack, the attacker needs to: (1) inject the shellcode into the vulnerable variable and (2) overwrite the EIP value stored on the stack with the address of the shellcode in memory. These tasks are not straightforward since the memory addresses of variables are determined at runtime. Furthermore, dynamic variables make it difficult for an attacker to determine the location of the return address with respect to the vulnerable variable. In order to improve the chances of success, the attacker adds the return address and NoOP (No OPeration) sled components to the shellcode. Although, different forms of stack-based buffer overflows exist [101], Figure 2.1 shows the common buffer overflow exploit where the NoOP sled precedes the shellcode and the back-to-back return address component follows it. Sections 2.2.1, 2.2.2 and 2.2.3 details the shellcode, return address and NoOP sled components respectively.

2.2.1 Shellcode

A shellcode is a short assembly program which aims to execute on the attacker's behalf. The shellcode name is derived from the original objective of the shellcode, which is to spawn a UNIX shell. Although spawning a UNIX shell is still the most

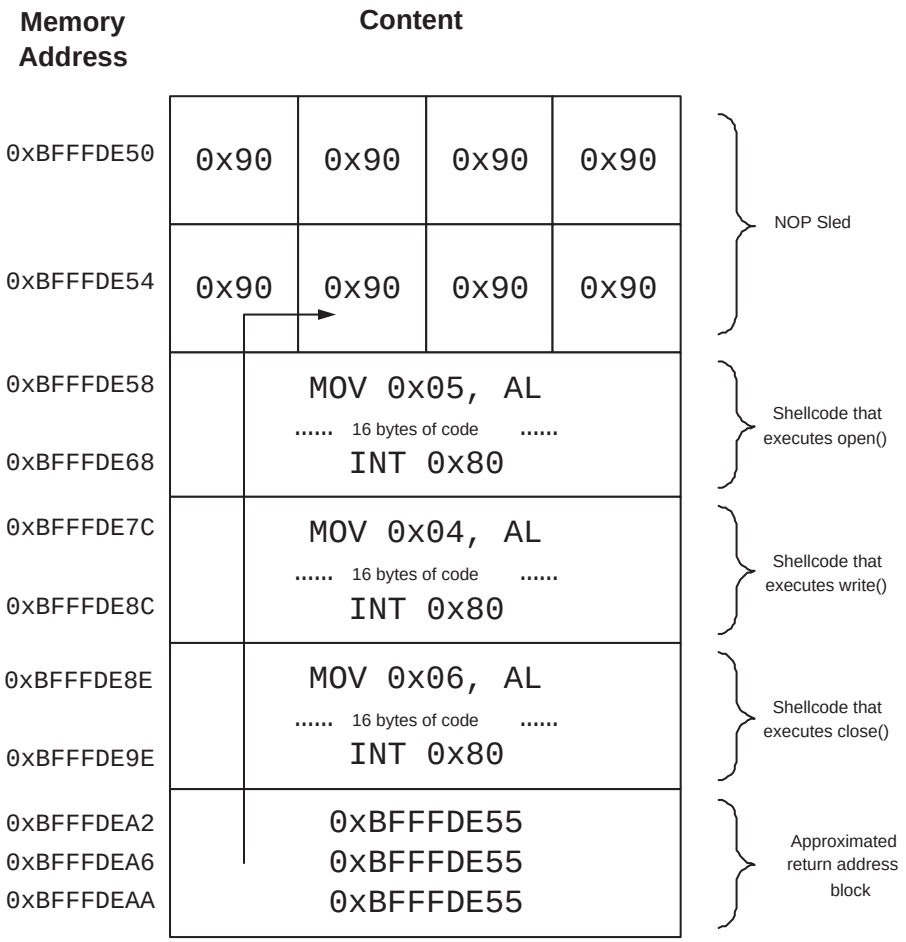

Figure 2.1: Common buffer overflow exploit

common form of shellcodes, attackers use shellcodes to perform various tasks such as modifying critical system files or opening and binding shells to sockets [54] [31].

There are numerous characteristics which differentiate a shellcode from a regular program. First, the regular program allocates memory for variables (in fact, variables are stored in data segments and the program resides in code segments) whereas a shellcode generally works only within its memory space. That is to say, if a variable is required for the shellcode, it is usually explicitly allocated within the shellcode. Second, programs do not usually have size constraints whereas a shellcode needs to be concise since it must fit within the vulnerable buffer. Third, although both programs and shellcodes are stored as hexadecimal numbers, the shellcode cannot contain a null byte (`0x00`) since memory copy functions such as `strcpy` will only copy until the null byte is reached thus failing to copy the entire shellcode.

The objective of the shellcode is to force the vulnerable program to behave in a way which achieves the objective of the attacker. One way to achieve this is to force the vulnerable program to make system calls. A system call is a mechanism which programs use to request a service from the operating system such as input output functionality, running and exiting processes. In protected mode, the operating system kernel runs in a privileged mode, which allows the kernel to access system resources such as disks, networks and hardware. User programs use system calls to request access to these system resources. In theory, separating the kernel mode from the applications prevents applications from modifying the kernel space.

Two methods are presented for calling system calls [54]: the first is to use a C library wrapper or libc [25] and the second is to call the system call with an assembly function (or a shellcode) which copies the system call arguments to corresponding registers [54]. In order to call a system call with a shellcode, the following tasks need to be performed.

1. A system call number should be stored in the EAX register. The list of system calls and their corresponding numbers are operating system dependent and are usually included in the operating system code and/or documentation. For example, Red Hat 9.0 Linux defines the system call numbers in: `/usr/include/asm/unistd.h`.

2. The arguments to the system call should be stored in the remaining registers. (first argument in EBX, second in ECX, third in EDX etc.). If there are more than three arguments or if the argument is too large to fit in the register (in the case of a C/C++ data structure), the registers contain the pointers to the system call arguments.

3. Interrupt `0x80` should be executed.

After the interrupt is executed, the CPU switches to kernel mode and the system call is executed based upon the parameters supplied in the registers. Therefore the general objective of the shellcode is to set the register values and call the interrupt to execute the system calls. Some of the system calls commonly used by attackers are `execve`, which executes a program the location of which is passed as a parameter; `open`, `write` and `close`, which access and modify the file which the user requests. The shellcode can call one or more system calls depending upon the objective of the attacker and the obfuscation method used.

2.2.2 Return Addresses

In order to direct the execution to the shellcode, the approximate location of the vulnerable buffer which stores the shellcode must be known. Attackers approximate the location of the vulnerable buffer by subtracting a suitable offset from the stack pointer (ESP) [25]. The stack pointer can be obtained by analysing the application with a debugger such as the GNU Project Debugger (GDB) [106]. Subtracting an offset from the stack pointer effectively moves the return address value toward the variables stored in the stack. Naturally, a larger offset is subtracted if the vulnerable buffer is farther from the stack pointer. Once an approximation is made, the attacker creates a block of back-to-back approximated return addresses and appends it to the end of the shellcode. As long as (i) one of the return addresses overwrites the actual return address stored on the stack (Figures 2.1 and 2.2) and (ii) the approximated back-to-back return address is accurate enough to jump to the NoOP sled, the execution will be directed to the shellcode.

One issue related to the return addresses is alignment. Even though the return address is approximated accurately enough to jump to the first instruction of the

shellcode, if the stored return address on the stack is not overwritten with a properly aligned return address, the misaligned return address will direct to the wrong memory location causing it to crash or behave unexpectedly. For example, if approximated address `0x1234` is misaligned by 1 byte, the value which overwrites the actual return address can be `0x3412`.

Furthermore, in Complex Instruction Set Computer (CISC) architectures such as Intel x86, the length of the instruction is not fixed. Therefore if the return address jumps to anywhere other than the first byte of the instruction, the instruction misalignment can cause the shellcode to crash, even though the shellcode is valid. This increases the importance of the NoOP sled since the length of the NoOP instruction is 1 byte and therefore it does not create misalignment problems.

2.2.3 The NoOP sled

Estimating the address of the first instruction in the shellcode is crucial since jumping elsewhere in the shellcode will have an undetermined outcome. In order to eliminate the need for estimating the exact location of the vulnerable buffer, the beginning of the shellcode is padded with a special purpose single byte instruction called 'no operation' or NoOP, which is used to waste computational cycles (generally, for the purpose of timing). Sequences of NoOP instructions are referred to as the NoOP sled. As long as the desired return address is accurate enough to direct execution to the NoOP sled, the EIP will be incremented after the execution of each NoOP sled until the execution reaches the shellcode.

A longer NoOP sled increases the chance of the shellcode deploying successfully since more than one return address value will allow the shellcode to execute. However, the NoOP sled usually manifests itself as a long sequence of `0x90` bytes, hence presenting a very obvious detection signature. Therefore, from the attacker's point of view, shorter NoOP sleds are desirable, which implies that the original stack pointer offset (indirectly, the location of the shellcode) must be estimated more accurately.

Figure 2.2 shows a simple program which is susceptible to stack buffer overflow attacks. The size of 'char' is assumed to be 1 byte and the size of 'long' is assumed to be 4 bytes. The function argument X is copied onto B without any data integrity

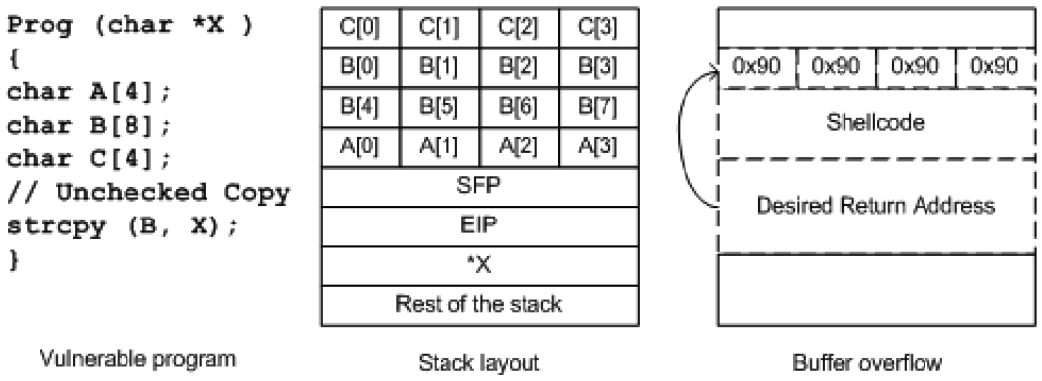

Figure 2.2: Example of a stack buffer overflow

checks. The stack layout shows the variables on the stack after the program is executed and the buffer overflow which overwrites the EIP value stored on the stack with the return address value supplied by the attacker.

If the overwritten return address points to anywhere in the NoOP sled or the first instruction of the shellcode, the attack is deployed successfully provided that the shellcode is injected properly.

2.3 Discussion

In order to evade detection, the attacker needs to enhance numerous characteristics of stack buffer overflow attacks.

1. **Length of the NoOP sled and the accuracy of the return address:** In order to be able to direct the execution to the shellcode, the attacker needs to approximate the address of the vulnerable variable on the stack and create a malicious buffer which will be injected into the vulnerable application. Since the return address is an approximation, the attacker needs to add a sequence of NoOP instructions, which works like a 'safety zone' which improves the chances of the attack. As long as the approximate address directs the execution to the NoOP sled, the attack can deploy successfully. As the approximation of the return address gets more accurate, the attacker can employ shorter NoOP sleds, which in turn enables the attack to bypass the signature detectors. To address this problem, this thesis employs Grammatical Evolution to identify suitable

NoOP and return address components. The proposed methodology and results are discussed in Chapter 6.

2. **Shellcode design at the assembly level:** Another aspect of the attack is the design of a shellcode at the assembly level. Although the attacker can use readily available shellcodes [75], misuse detectors contain signatures which can detect the existence of certain assembly instructions (such as NoOP, which is equal to hexadecimal `0x90`) and parameters (such as `/bin/sh`, which is equal to hexadecimal `0x2f62696e2f7368`). To design shellcode at the assembly level, this thesis employs Genetic Programming, where the code bloat phenomenon creates a possibility of (i) creating equivalent NoOPs or (ii) distributing NoOPs across the shellcode thus masking the original intent of the shellcode. Furthermore, given a suitable instruction set and a fitness function which describes the core objectives of an attack, Genetic Programming can find alternate ways to achieve the attacker's objectives while bypassing the signatures. The proposed methodology and results are discussed in Chapter 7.

3. **Shellcode design at the system call level:** By optimizing the NoOP sled length and return address accuracy, the attacker can avoid using highly detectable NoOP sleds. By mixing the NoOP instructions with the shellcode and finding alternative instructions to carry out the attack, the attacker can also bypass the signatures which monitor detectable shellcode characteristics. However, even though the attacker deploys the attack without getting detected by misuse detectors, an anomaly detector running on the target host which monitors the application behaviour can detect the attack by identifying deviations from normal behaviour. Therefore, the attacker needs to design a shellcode which executes a sequence of system calls which does not produce a detectable deviation from normal behaviour, while achieving the attacker's goals. The above-mentioned evasion attacks, which remain undetected while being deployed successfully, are also called mimicry attacks. The proposed mimicry attack generation methodology and results are detailed in Chapter 8.

Chapter 3

Background on Detectors and Attacks

The proposed approach can be considered as a first step towards creating and artificial arms race between the detectors and the attackers with the ultimate objective of improving the state-of-art in intrusion detection. Within this view, this chapter focuses on the relevant work on intrusion detectors and the evasion techniques developed against the detectors. Therefore, the relevant work discussed in this chapter is divided into two categories: research on detectors and research on evasion methodologies, specifically mimicry attacks research.

3.1 Previous Work on Detectors

Different detection mechanisms can be employed to search for evidence of intrusions. Two major categories exist for detection mechanisms: misuse and anomaly detection. Misuse detection systems, which are discussed in Section 3.1.1, use *a priori* knowledge for detection. In other words, they detect intrusions by knowing what the misuse is or what it causes. On the other hand, the anomaly detectors in Section 3.1.2 adopt the opposite approach, which is to employ the knowledge of normal behaviour, and then find deviations from it. These deviations are considered as anomalies or possible intrusions.

3.1.1 Misuse Detectors

Although anomaly detectors have been the focus of recent detector research, misuse detectors are the basis for most tools used in practice. In general, such tools are based upon the efforts on pattern matching detectors [87] and static prevention mechanisms [15] [108].

Pattern matching (i.e. misuse detection) [87] was the initial method used to detect buffer overflow attacks. In this approach, the defining characteristics of attacks (such

as a sequence of NoOP instructions, or crucial segments in a shellcode) are employed to create attack signatures. The advantage of pattern matching is that it is easy to deploy. On the other hand, even slight changes in attacks allow attackers to camouflage their actions easily [48]. Snort, which is discussed in detail in Section 4.1, is one of the more well-known open source misuse detectors which include shellcode signatures.

On the other hand, static prevention techniques such as Stack Guard [15] and Stack Shield [108] focus on the way attackers crash the stack to gain control. Although alternative methods exist, the most generic type of stack overflow has to overwrite the return address stored in the stack to take control of the execution. Therefore these methods look for abnormal changes in return addresses stored in the stack. Stack Guard inserts 'canary' variables close to the return addresses stored on stack. Since the common attack technique is to write back to back desired return addresses hoping that one will overwrite the return address, it becomes difficult to overwrite the return address without overwriting the 'canary' word. Before the function returns, the 'canary' word is checked. If it is altered, the program exits gracefully without executing the attacker's code. Furthermore Stack Shield inserts control variables before and after the text segment, which holds the executable code in memory. The main idea is that a return should always divert the execution to a point within the code and any diversions outside the code causes the program to exit gracefully. In terms of recent protection mechanisms, Hiroaki Etoh implemented a stack overflow protection mechanism [93] into GNU C Compiler (GCC) version 3. The stack smashing protector in GCC allows the vulnerable program to exit safely by preventing the attacker from overwriting the return address. Moreover, address space randomization efforts [90] aim to make it difficult for the attacker to approximate the return address values.

Although the proposed static prevention methods are easy to deploy and provide effective protection against certain types of buffer overflow attacks, they are computationally demanding (i.e. they introduce a computational overhead between 69% and 16600% [15]) and methods have been published on how to bypass both Stack Guard and Stack Shield [10]. The main method behind bypassing Stack Guard and Stack Shield is to discover the 'canary' variables and deploy the exploits without modifying

them [10]. To overcome these challenges, Dalton et al. [17] proposed a hardware assisted method recently which involves dynamic information flow tracking. In such a scheme, the untrusted data is tagged and tracked while it propagates through the system. If the untrusted data is handled in an unsafe manner, the detector throws a security exception.

3.1.2 Anomaly Detectors

Anomaly detection systems attempt to build models of normal user behaviour and use this as the basis for detecting suspicious activities. This way, known and unknown (i.e. new) attacks can be detected as long as the attack behaviour deviates sufficiently from normal behaviour. When a buffer overflow attack is deployed, a vulnerable privileged program is exploited to do something which it is not supposed to do. This implies that it is possible to observe a change in program behaviour. Anomaly detectors are based upon this assumption. Unfortunately, should the attack be sufficiently similar to the normal behaviour, it may not be detected.

In terms of host-based anomaly detection, Forrest et al. [30] employed a methodology motivated by immune systems. This characterizes the problem as distinguishing self from non-self (normal and abnormal behaviours respectively). An event horizon is built from a sliding window applied to the sequence of system calls made by an application during normal use. The sequences formed by the sliding window are stored in a table which establishes the normal behaviour model. During the deployment (detection) phase, if the pattern from the sliding window is not in the normal behaviour database, it is considered a mismatch. Forest et al. [30] used the sendmail service, which is a part of the University of New Mexico Computer Immune Systems Data Sets [109], in their experiments because it has sufficiently varied normal behaviour and there were many documented attacks against sendmail. In order to create normal behaviour, the authors used a suite of artificially-created messages which contains as many variations on normal behaviour as possible. Variations in message length, encoding, content and senders/receivers were considered. Once the normal behaviour was generated, normal behaviour database was developed on the sequences of system calls which sendmail made while processing the messages. To do so, they used a

sliding window of sizes 5, 6 and 11 in order to maintain a record of both current and past system calls. After the database was created, a sliding a window of the same length was employed on the new trace. The percentage and numbers of mismatches were calculated, where a mismatch was defined as the system call sequence which does not appear on the normal database. The authors used mismatch metrics to formulate the degree of anomaly on a given trace. The results [30] indicated that the normal behaviour database distinguished both successful and attempted attacks from the normal execution of the code. Their approach was implemented later as a detector called Stide [109]. Stide is accompanied by the University of New Mexico Computer Immune Systems Data Sets, which contains the system call traces for UNIX applications such as sendmail, ftp, lpr, inetd and xlock. The traces provided in the University of New Mexico Computer Immune Systems Data Sets are preprocessed by mapping the system calls to discrete numbers and by eliminating the system call parameters.

Relevant work which has extended Stide can be categorized further into two groups, (1) research which includes more information from the system state such as the program counter values and the stack state and (2) research which utilizes the system call sequence information by focusing on representation, clustering of anomalies and learning methodology.

Extending Stide with Employing Additional System State Information

In terms of research which extends Stide by employing more information from system state, Sekar et al. [89] extended the work of Forrest et al. [30] by utilizing finite state automata (FSA) to represent the normal behaviour database. Their objective was to develop an efficient method which could (1) work in real time (given that building a FSA is computationally expensive) and (2) capture both short and long term correlations. The main contributions of their approach are listed below:

1. As opposed to fixed window length approaches [30], it does not have a limitation on the length of system call sequences.

2. It can capture the behaviour of branching and looping structures by using program counter information, which should reduce false positives (i.e. a better defined model).

3. It learns the behaviour of the program and leaves out the behaviour of library functions. For instance, if the program calls a shared libc function such as printf, resulting system calls would provide no useful information about how the main program works.

The authors used nfs, ftpd and httpd services in their experiments where the proprietary system call data sets are generated by "using training scripts that attempt to simulate the requests likely to be handled by each of these servers" [89]. For ftpd and nfs, they developed training scripts where each script generates a random sequence of valid commands as well as invalid ones which were introduced deliberately whereas the system calls of the httpd service were collected from a live web server. Although no further information was given, the authors claim that the distribution of the commands was set to be similar to the distribution in the normal operation of the services. In addition to the system call sequence information, two further observations were made. The first observation is the program counter which stores the memory address of the code, from which the system call was executed. The second observation is the stack information which allows shared library code to be tracked back to the main program. The results were expressed in terms of a convergence rate (model length over the number of system calls in the model), a false positive rate and computational complexity. The results indicate that compared with the sliding window approaches [30], FSA converges quickly and provides a better false positive rate. It has been found effective in detecting a wide range of attacks including stack buffer overflows. In particular, stack buffer overflows are detected because stack information is used to track back from the shared library to the main program. If the back track fails, it means the stack is corrupt and therefore it is likely that a buffer overflow attack has occurred.

Feng et al. [27] [26] substituted the FSA [89] for a virtual path table which not only keeps track of where the system call is executed from, but also employs the point where the execution is returned. The authors assumed that all the functions

which the program calls are statically linked, hence the return addresses observed on the stack are consistent over different runs of the same program and therefore the approach may not be applicable to a common case where shared libraries are linked dynamically. An execution path is generated between the two execution points in the program by comparing the list of return addresses on the stack. In summary, a virtual path contains the list of functions which the program needs to execute or return from in order to get from one execution point to another. The authors employed httpd and ftpd services in their experiments. The proprietary httpd system call data was generated by using a web benchmarking suite. The proprietary ftpd system call data was generated by their automated scripts which mimic common user activities such as downloading and uploading. After training is conducted on the proprietary data sets, anomaly detection occurs as follows.

Given the input trace:

1. If the stack history is corrupt (inconsistent return addresses), it is a stack anomaly.

2. If there is an unexpected system call, it is a system call anomaly.

3. If the given trace does not match any known virtual paths, it is tagged as a virtual path anomaly. In other words, the program never used that execution path before.

4. If a new return address is observed, it is a return address anomaly.

Item 1 can be detected by the previous work by Sekar et al. [89] and Item 2 can be detected by the previous work by Forrest et al. [30] and Sekar et al. [89]. The last two employs the return address information – the main contribution of the work by Feng et al. [27]. In their more recent work, Feng et al. employed static analysis to extract an automaton with call stack information [26]. They discussed that the models based on push down automata are inefficient due to non deterministic stack activity. The stack of the process is analysed to extract the return addresses, which in turn reveals the context information and enables a deterministic model.

Similar to Feng et al. [27] [26], Wagner et al. [103] employed an analysis of the source code to improve the detection rate. As opposed to the work of Feng et al. [27]

[26], which employed program counter information to determine where the system calls are executed, Wagner et al. [103] employed the analysis of the source code. Their main assumption was that it is possible to infer where the system calls occur at the source code. First, an expected program behaviour model is computed from the application source code, and then in runtime, system calls were checked against the model to detect any violations. The authors [103] used a callgraph model by building a control flow graph. Each node in the graph is a point in the source code. Transitions from the node occur when (1) a system call is made, (2) a function is called or (3) a function returns. Such a graph is generated by a context-free grammar with non-terminals as nodes and terminals as system calls. In the detection phase, if the observed system call sequence cannot be achieved by transitioning from one node to another in the graph, it is flagged as an anomaly. The main contribution of their work is the incorporation of source code analysis to intrusion detection, as opposed to employing runtime debugging information to infer program behaviour. This also means that as opposed to runtime observations, which may not necessarily cover all possible program branches (loops and if statements), the graph covers all possible branches. Moreover, mimicry attacks are discussed briefly in their paper, where the authors suggest monitoring system call arguments for more precise detection [103]. Although the authors provide an in depth discussion of the implementation issues of the detector, very little information on the data sets was provided other than the name of the applications (i.e. finger, qpopper, procmail, sendmail).

Mutz et al. [72] proposed a host-based anomaly detection system where multiple detection models are applied to system call arguments and an overall aggregate score of these models is introduced to determine if an event is an attack or not. They employed a Bayesian modelling approach to combine multiple anomaly scores from system call arguments. Furthermore, they compared their approach with sequence-based methods such as Stide, bag of system calls and clustering algorithms. In their experiments, the authors employed the Solaris Basic Security Mode (BSM) Audit logs of UNIX applications (i.e. eject, fdformat, ps, ftpd, sendmail, telnetd) from the 1999 MIT Lincoln Laboratories Intrusion Detection Data sets [63].

Similar to the work of Mutz et al. [72], Sufatrio et al. [94] provide an extension to the detectors which employs system call sequence information by extending the detector to monitor additional system call attributes such as system call arguments and process privileges. The process privilege information indicates whether the system call is executed with super-user privileges. Thus, if a system call is executed with user privileges during normal operation but gets executed with super-user privileges during deployment, it is identified as an anomaly. In order to evaluate system call arguments, the authors proposed a categorization scheme where each category acts as an abstraction of arguments and specifies the security impact of the parameter on the system state. Such a categorization scheme is user supplied and therefore the effectiveness of the detection relies on the proper specification of the parameters. Sufatrio et al. [94] employed Stide [30] as well as their implementation which extends Stide against traceroute and ftpd applications and the JOE text editor. Their results indicated that incorporating process privileges and system call argument information improves the resistance of the detectors to mimicry attacks without increasing the false alarm rate.

In terms of detection techniques which employ system call parameters, Li et al. [62] proposed an approach to incorporate system call parameters into the detection process. To this end, they proposed a three stage approach. The first step involves discovering the system call sequences using the Teiresias algorithm [86]. Using association rule mining, the second stage generates rules which govern the system calls and system call arguments which perform repetitive tasks whereas the third stage aims to discover rules which govern the system calls from different patterns. The authors employed their approach on a web server which hosts the web pages downloaded from a university department web site. Li et al. [62] provided results in terms of the speed of convergence, false alarm rates and latency. However, since they did not test their approach against an actual attack, no detection results were provided.

Tan established that the length of the sliding window plays an important role in detector performance [97]. Specifically, for Stide to detect the attack, the sliding window length should be longer than the minimum foreign sequence. The minimum foreign sequence was defined as a sequence in which all of its subsequences exist in

the normal database. Tan et al. [97] argued that, by developing an attack variant which increases the minimum foreign sequence length, the attacker can avoid detection. Similarly, Wespi et al. [110] proposed a method to discover variable length sequences from the training data. Variable length patterns are deemed more suitable for anomaly detection because their observations on sequences of events created by a process revealed that there are long, process-specific patterns. Sliding window lengths in fixed length approaches are much shorter so they cannot capture behaviour defined by long sequences.

In order to discover the variable length sequences, Wespi et al. used the Teiresias algorithm [86], which was developed originally to discover "rigid patterns in unaligned biological sequences" [110]. The main objective is to build a table of maximal patterns, which is the longest sequence which covers a large portion of the training set. The table should contain only those patterns necessary to cover the training set, hence reducing the number of patterns. No training occurs in this approach. The authors employed the ftpd application in their experiments in which the proprietary system call data was generated by utilizing an operating system test suite which automatically exercised ftp commands.

Results showed that compared with sliding window approaches [30], the variable length approach [110] needs to store fewer patterns to cover the training set. Furthermore, variable length patterns provided better coverage for the test data (which contains the normal behaviour), which in turn reduces false alarms. The tests on attacks are not very detailed but the authors [110] conclude that the separation between normal behaviour and anomalies is more reliable than with fixed length approaches.

Extending Stide without Employing Additional System State Information

In terms of the relevant work which extends Stide while utilizing only system calls, Somayaji [91] developed a methodology similar to Stide and implemented it as a Linux kernel extension. The resulting detector is called Process Homeostasis (pH) and discussed further in Section 4.2.2. Somayaji employed look-ahead pairs since it is more efficient and faster to converge. Furthermore, Somayaji introduced tolerization and sensitization concepts to the detector. Based upon the assumption that malicious

behaviour will produce anomalies clustered together, tolerization allows the detector to reduce false positive rates by neglecting scattered anomalies, which are likely to be slight changes in normal behaviour. The detector is retrained as the false positive rate increases. Sensitization prevents attacks from leaking into the normal database. If the locality frame count for a process exceeds the limit, the process is sensitized by resetting the look-ahead pairs for the process. Another important feature of pH is that it responds to attacks by slowing down the process. Delay is an exponential function of locality the frame count which aims to identify clusters of anomalies.

The author employed pH on his workstation as well as on another three hosts (namely, a personal workstation, the University of New Mexico mail and web server and the University School of Nashville web server), which were used on a daily basis. The main purpose was to observe pH while building normal behaviour models and employing the resulting models while monitoring normal behaviour. Furthermore, in order to observe the detection of intrusions, the author employed pH on a host with a vulnerable fetchmail program and executed an exploit against the vulnerable program. This thesis employs Process Homeostasis, therefore it is discussed further in Section 4.2.2.

More recently, Inoue and Somayaji compared the look-ahead pairs method against the full sequence method [38]. Furthermore, they extended pH by using random schema masks. The main idea behind schema masks is to maintain a longer sliding window while ignoring some of the calls within the window. That is to say, the schema mask determines which locations within the sliding window are ignored while producing the output from the sliding window. They determined that utilizing longer sliding window lengths improves detection, therefore they maintain a longer sliding window size and take taps from locations determined at the detector configuration. The schema mask method has two advantages. First, the detector monitors a longer time window without increasing computational complexity. Second, if the attacker does not know the schema mask, he/she would have to develop mimicry attacks based upon window sizes. The authors employed lpr-mit, named, sendmail and xlock traces from the University of New Mexico Computer Immune Systems Data Sets [109]. This thesis employs Process Homeostasis with schema masks so it is discussed further in

Section 4.2.3. Finally, for a brief review of the research on system call monitoring, the reader is encouraged to read the literature review paper by Forrest et al. [29].

Machine Learning Approaches to Anomaly Detection

Several researchers have applied machine learning to host-based anomaly detection. According to the learning method employed learning algorithms are typically supervised or unsupervised. In supervised learning, the learning algorithm utilizes a set of classified (or labelled) instances and is expected to identify a way of predicting new unclassified instances [112]. Using a bag of words representation for system call traces, Kang et al. [43] employed supervised learning algorithms such as Naive Bayes, decision trees, RIPPER and SVM on system call data which contains both attack and normal behaviour traces. The authors employed the fourth week data from the 1999 MIT Lincoln Laboratories Intrusion Detection Data set [63] and lpr and sendmail data from the University of New Mexico Computer Immune Systems Data Sets [109].

On the other hand, unsupervised learning involves searching for associations between features without making use of classes or labels. The Markov Model [84] [65] is a particularly suitable machine learning method which can model temporal relations in the input data explicitly [114]. Moreover, such a model has been utilized within the context of intrusion detection systems [98] [114] [33] [83], as a related case of a Finite State Automata representation [89]. Typically, the relevant work on Markov Models employed a sliding window on the sequences of system calls. Yeung et al. [114] and Qian et al. [83] employed left-to-right and fully connected Hidden Markov Models whereas Gao et al. [33] and Tan et al. [98] employed only fully connected Hidden Markov Models. Although there are some variances in representation and the preprocessing of system calls, the common objective is to build a 'normal behaviour' model for a process. Given new behaviour, hypothesis testing is employed to determine whether the new behaviour is anomalous or not. The Markov Model detector which is employed in this thesis is similar to the relevant work and is discussed further in Section 4.2.4. Yeung et al. [114] employed ps, login, named and sendmail system call traces from the University of New Mexico Computer Immune Systems Data Sets [109]. Qian et al. [83] employed the sendmail system call traces from the University

of New Mexico Computer Immune Systems Data Sets [109]. Gao et al. [33] employed the httpd service and utilized a web benchmarking tool to generate the proprietary system call data set. By contrast, Tan et al. [98] employed the sendmail, lpr and ftpd system call traces from the University of New Mexico Computer Immune Systems Data Sets [109].

In terms of neural network approaches, Han et al. [36] employed evolutionary neural networks in anomaly detection. According to Han et al. [36], evolutionary neural networks have the advantage of shorter training times compared to the conventional neural network approaches and it can learn the structure and weights of the network simultaneously. The authors employed the Solaris Basic Security Mode (BSM) Audit logs of UNIX applications (i.e. eject, fdformat, ps, ftpd, sendmail, telnetd) from the 1999 MIT Lincoln Laboratories Intrusion Detection Data sets [63]. In the input layer, they had 10 input nodes, hence setting their system call window length to 10. In the output layer, they had 2 nodes, where the nodes represented attack and normal behaviour. The authors stated that they focused on detection of user-to-root attacks [36] but did not provide detailed information on the training data or the applications that they focused on.

On the other hand, even though a bottleneck feedforward back propagation neural network is employed in one-class document classification by Manevitz et al. [64], it is suitable for the one-class learning scheme where the training takes place only on system calls consisting of normal behaviour. In a bottleneck neural network, given an n dimensional input space, a two layer network is built where the second layer has the same number of outputs as inputs. The number of neurons in first layer is m, where $m << n$. By utilizing fewer neurons on the hidden layer, authors state that it is possible to avoid the saturation problem which is the result of learning from only positive examples. The network is trained to produce outputs identical to the input space, hence error is expressed in terms of Euclidean distance or mean square error. In terms of the one-class classification problem, Manevitz et al. [64] state that when a new pattern is presented, a neural network produces the corresponding outputs. If the presented pattern is similar to the training samples, the difference between the outputs and the inputs will be similar. Furthermore, Manevitz et al. [64] investigated

threshold determination for one-class classification efforts. Although they employed their methodology on different classes of documents provided in the Reuters database [61], it can be adopted easily to work as an anomaly detector. Thus, this thesis employs a similar detection methodology in the experiments, therefore bottleneck neural networks are discussed further in Section 4.2.5.

3.2 Previous Work on Mimicry Attacks

The 'mimicry' concept [111] has its origins in biology where an organism bears a superficial resemblance to another organism. In general, mimicry is advantageous to the mimic. Numerous types of mimicry exist in biology depending on the organisms involved and the interaction between the organisms.

The mimicry attack which is of interest to this thesis is similar to 'aggressive mimicry' in which "the mimic adopts certain of the recognition marks of its model in order to secure advantage over the model itself or over a third species that interacts with the model" [111]. In such a scheme, the mimic may be a predator trying to gain an advantage over the prey, or it can be the prey camouflaging its characteristics to avoid predation. Within the context of computer security, a mimicry attack can de defined as an attack which is modified in order to avoid detection by the detector (misuse or anomaly).

Wagner et al. [105] introduced the 'mimicry attack' concept in which original attacks were modified to evade detection. Wagner et al. [105] draw an analogy to biological mimicry where a successful mimic will be recognized as 'self' by the immune system (i.e. recognized as 'legitimate' by the detector) and will not produce alarms while performing the malicious task.

In this thesis, a mimicry attack is considered to have two components: (i) the preamble, which consists of the actions which the attacker must perform to take control of the vulnerable application and (ii) the exploit, which is the code that the attacker injects to achieve the malicious goals. By contrast, the previous research discussed in this section considers a mimicry attack to contain only the exploit section by assuming that the attacker can take control of the application without creating anomalous behaviour. Therefore, the term '(mimicry) exploit' which is used in this

section (and the rest of this thesis) refers to the term '(mimicry) attack' in the previous work [105] [96] [99] [32] [56] [34] [77].

In the paper which introduced the mimicry attack concept, Wagner et al. [105] proposed three methods to avoid detection: (i) modifying system call parameters; (ii) inserting system calls which are irrelevant to the attack being deployed while minimizing the anomaly rate; and finally (iii) generating equivalent exploits by replacing the system calls which can be identified easily by the detector. An example of the last method is substituting an exploit which spawns a UNIX shell with an exploit which creates a super-user account where the outcome results in the attacker gaining super-user privileges. Given the assumption that the system calls in the normal behaviour database is rich enough for an attacker to deploy the exploit, the objective is to determine whether an equivalent exploit exists in the normal behaviour database. To do so two sets of languages are defined: Allowed system calls A and malicious system calls M. Based upon language theory, they determine whether the intersection of A and M is empty or not. If it is not empty, that means an equivalent exploit can be constructed against the detector, which was pH. Wagner et al. employed pH with a sliding window size of 6, so the results were applicable to Stide as well. Mimicry exploits were generated for the wuftpd service against pH by modifying the detectable system call sequences manually. Normal behaviour was generated by "running wuftpd on hundreds of large file downloads over a period of two days" [105]. Although Wagner et al. were aware of the attack preamble, they assumed that the attacker could take control of the application without being detected [105]. This implies that the break-in does not generate anomalous behaviour which the anomaly detector can detect. By taking preambles into account and investigating the impacts of preambles, this thesis proposes that the attacker may not always be able to take control of the application without causing some anomalies. This is due to the fact that, during the break-in, the attacker does not have full control of the application and may not be able to control the execution of the vulnerable application.

Stide detects foreign sequences which are not in the normal database. Thus, Tan et al. [96] investigated hiding in the detector's blind spots in more detail by developing variants of a core exploit manually with the objective of increasing the minimal foreign

sequence length. They reported that if the foreign sequence length is greater than the sliding window size of Stide, an attack could evade detection. In their experiments they employed Stide on traceroute and passwd applications. For traceroute, normal behaviour was obtained by "executing traceroute to acquire diagnostic information regarding the network connectivity between the localhost and nis.nsf.net" [96]. For the passwd application, normal data was obtained by executing the passwd command without any arguments, which replaces the old password with the new one provided by the user. They provided a detection map for Stide, which shows the detection of an exploit (or lack thereof) as the sliding window size changes. When the foreign sequence length becomes greater that the sliding window length, Stide is unable to detect the exploits [96].

In a more recent work, Tan [99] employed four methods to change the behaviour of the exploit manually by: (i) hiding an exploit in the blind spot of the detector; (ii) modifying an exploit so that it resembled like a normal behaviour; (iii) hiding an exploit so it resembled a less dangerous exploit; and (iv) modifying an exploit so that it looked like a different exploit. In their experiments, Stide was employed to monitor the exploits against restore, tmpwatch and kernel/traceroute applications. Normal behaviour for Restore was obtained by "monitoring a regular user executing the restore system program to retrieve backup data from a remote backup server" [99]. Normal behaviour for tmpwatch was generated by populating a short directory tree with files under the */tmp* directory and executing the tmpwatch program to clean files more than 5 days old. Normal behaviour for the kernel attack was not obtained since the vulnerability in the kernel was used to exploit another vulnerability in traceroute. In their paper, they argue that malicious acts and anomalous behaviour are not synonymous, which is to say, sophisticated attackers can alter their actions to hide their exploit as normal behaviour or as a less serious exploit.

Using a categorization scheme, Gao et al. [32] divided anomaly detectors into three categories: black-box detectors [30] [91] which only make use of system calls, gray-box detectors [89] [27] which use – in addition to system calls – runtime observations such as program counter values and return addresses stored in the stack. White-box detectors [103] [26] however, incorporate information from the source code

as well, which makes it difficult to hide the attacks. The authors presented a systematic study, which showed the benefits and overheads of changing gray-box anomaly detector parameters such as (1) the amount of runtime information, (2) the atomic unit which the detector monitors and (3) the sliding window size. They employed the ftpd and httpd services on a Red Hat host and generated their proprietary data by developing test runs to generate system call traces along with program counter and return address records. However, they did not provide any information on how the test runs were developed. Experimental results indicated that expanding the model by using more information and increasing window size results would increase the mimicry exploit length. In other words, attackers would need more code to hide their actions. Although it is more difficult to evade white-box detectors, the authors determined that they are platform dependent and are not universally applicable [32]. In addition to the systematic study, the authors presented a methodology to forge the program counter information manually on statically linked executables so that the gray-box detector [89] [27] does not detect an anomaly in the return addresses even though system calls are made by the exploit code [32].

Kruegel et al. [56] developed a methodology against adaptive detectors (i.e. Stide variants [30] [91] [89] [27] [103] [26]). Kruegel et al. [56] implemented the automation using a static tool at the Intel x86 assembly level to redirect control flow using symbolic execution. It is assumed that at the end of a system call execution, the prevention mechanism (such as [15]) will return the execution to the correct code. This implies that attackers can execute only one system call before the program regains control, which may not be sufficient in most cases because attackers need first to elevate their privilege, then spawn rootshell. Therefore, the objective of the attacker is to deploy the attack by gaining and relinquishing the control of the program so that the application code is forced to return to the attack code after the overflow occurred. This implies that the attacker will rearrange the execution environment (including CPU registers, writable memory locations such as stack, heap and data segment) instead of executing the system calls explicitly. To this end, the changes in the execution environment are expressed as polynomial expressions. The condition, which forces a malicious pointer modification, is added to the constraints. If the

resulting linear inequality has a solution, the attacker can find a configuration which can deploy an attack. They employed their methodology on the httpd, ftpd and imap services. Since their work is a comprehensive application analyser, they did not utilize any anomaly detectors (i.e. they analysed vulnerable applications to determine if a vulnerability exists). Needless to say such methodology assumes a full access to the source code and executables of an application (i.e. every time the source code changes or updates are applied, the analysis needs to be repeated). Although such a 'white-box' methodology performs a comprehensive search for mimicry exploits, enormous expert knowledge is required to define the execution environment state as polynomial expressions. Furthermore, having an expert define the execution environment had the potential for over-simplification of the system state or introducing biases due to the expert's view of the execution environment.

Giffin et al. [34] generated mimicry exploits against Stide [30] and variants [91] [89] [103] [26] by applying automatic model checking to prove that no reachable operating system configuration corresponds to the effect of an exploit. In their work, they utilized two models: (1) the system call behaviour of an application and (2) a model of the security critical system state, which describes the malicious operating system state. An approach based upon push down automata was employed to implement a program model which can search exhaustively for any sequence of system calls allowed by the program which can lead to a malicious operating system state. If there exists a sequence in the program model which can be employed to induce a malicious operating system state, it indicates that a mimicry exploit exists. In their experiments, they employed the ftpd, restore, traceroute and passwd applications. The ftp data was obtained from Forrest et al. [109], whereas the data for the remaining applications were generated by the authors following the approach which Tan described [96]. Numerous malicious activities were defined such as spawning a UNIX shell, writing to the password file and changing the permission of the password file to world-writable. Consequently, they found malicious sequences in the normal behaviour definition of Stide for all four programs. Although the search for malicious sequences was automatic in their approach, the operating system model,

application (program) model and system call specifications as well as the attack configuration were generated manually. This implies that for different operating systems, new operating system models need to be developed. Furthermore, proof of successful detection and the existence of mimicry attacks depend upon the careful abstraction of the operating system state. In other words, if the operating system is not defined properly, it may hinder the search process. The operating system models [34] are merely summaries of the actual operating systems and they were employed due to the computational cost of searching for mimicry exploits against the actual operating system. Thus, the search for mimicry exploits was limited to the conditions defined by the operating system models.

Parampalli et al. [77] proposed a methodology for generating mimicry exploits manually against "powerful system call monitors." A "powerful system monitor" is defined as a detector which has full knowledge of the system call parameters as well as their roles in the execution of the system call. They introduced the persistent interposition attack concept in which the objective of the attacker is to modify the read and write system calls to deploy the exploit. Their methodology is similar to man-in-the-middle attacks since the objective of the exploit code is to intercept and modify the read and write system calls which the victim application makes. They employed their methodology on a server running the Apache web server. Their results on the Apache web server showed that although the persistent interposition attacks were not powerful enough to obtain a rootshell, they could evade monitors which monitor system call arguments while achieving goals such as stealing financial information or impersonating web servers. Although the methodology aimed to evade "powerful system call monitors," their work focused on the analysis of the vulnerable applications (from the perspective of persistent interposition attacks) and did not provide specifics on the detection methodologies to which the method applies.

The work by Sparks et al. [92] shares similarities with the approach proposed in this thesis. Sparks et al. [92] employed Grammatical Evolution as a 'black-box' fuzzer in which the objective is to craft a set of inputs which will cause the program to take the execution path leading to a potentially unsafe string copy. The feedback to the fuzzer takes the form of a Markov Model which encapsulates the control flow

graph hence analysis of the source code is necessary in their work . Each individual corresponds to the set of inputs to the application being tested and the individuals are evaluated based upon how close they get to executing an unsafe string copy (with respect to the Markov Model). The authors [92] tested their approach on a Windows Ftp server (namely, tftpd.exe) and their results indicated that an approach based upon evolutionary computation is quicker than a random search for finding solutions.

In terms of evading misuse detectors, Vigna et al. [102] described a methodology to generate variations of an exploit automatically against Snort and ISS RealSecure to test the quality of detection signatures. Their main motivation was to provide a methodology which could evaluate the effectiveness of the signatures against evasion attacks. Stochastic modification of code was employed to generate variants of exploits in order to render the exploit undetectable. Techniques such as packet splitting, evasion and polymorphic shellcode were proposed [102]. They employed their evasion techniques on numerous exploits such as ftpd, httpd, imap, rpc and ssl. The modified exploits were then sent to the misuse detectors monitoring network traffic. Their results [102] showed that their evasion methodology succeeded in evading the detectors on numerous exploits, although certain exploits (such as the rpc) were detected in spite of the evasion techniques used. Furthermore, in their experiments, signatures from Snort were available to the attacker but the signatures from ISS RealSecure were not (the company chose not to share them, which is not surprising since the detector is a proprietary software on the market). Their results indicated that their approach was not as effective when the signatures of the target detector were not available to the attackers. This implies that generating attacks against a 'black-box' detector constitutes a more difficult problem than generating attacks against a 'white-box' detector since the attackers have to generate attacks based upon their limited knowledge of the detector.

Chapter 4

Intrusion Detection Systems Utilized

As discussed in Section 3.1, two methodologies can be employed for detecting buffer overflow attacks: misuse detection and anomaly detection. In misuse detection, identifying characteristics of a buffer overflow attack such as repeating NoOPs are sought in the data stream which the detector monitors. Conversely, in anomaly detection, a normal behaviour model is employed to detect deviations. This thesis employs detectors from both methodologies. For misuse detection, Snort was employed whereas for anomaly detection, Stide, pH, pH with a schema mask (pHsm), the Markov Model and Neural Network-based detectors were employed. The common attributes of the detectors which make them suitable for thesis experiments are that: (1) they are publicly available (in terms of source code or the description of the methodology employed) and (2) they do not require access to the source code of the applications nor do they require modifications to the environment in which they work.

From the perspective of a 'white-hat' attacker, the objective for evading misuse detectors is to find an attack which can deploy without triggering a signature, whereas the objective for evading anomaly detectors is to modify the attack so that it conforms to the normal behaviour of the detector.

4.1 Misuse Detectors

In the misuse detection approach, the detection process involves searching for known attack signatures on network or system resources. One of the main drawbacks of such systems is that they can only detect known attacks which are included in the signature database. A common example of a signature database is an array of link lists. This structure enables a search to be performed on only the applicable test conditions, thus minimizing computational requirements.

4.1.1 Snort

Snort is one of the better-known lightweight IDSs, which focuses on performance, flexibility and simplicity. It is an open-source intrusion detection system which is now in quite widespread use [87]. It can detect various attacks and probes including instances of buffer overflows, stealth port scans, common gateway interface attacks, and service message block system probes. Hence, it is an example of an active intrusion detection system which detects possible intrusions or access violations while they are occurring. Current versions of Snort provide IP de-fragmentation and TCP assembly to further the detection of attacks, albeit at the expense of having to view the complete attack data.

4.2 Anomaly Detectors

Anomaly detection systems attempt to build models of normal user behaviour and use this as the basis for detecting suspicious activities. In this way, known and unknown (i.e. new) attacks can be detected as long as the attack behaviour deviates sufficiently from normal behaviour. When a buffer overflow attack is deployed, a vulnerable privileged program is exploited to do something which it is not supposed to do. This implies that it is possible to observe a change in program behaviour. However, should the attack be sufficiently similar to normal behaviour, it may not be detected.

The anomaly detectors discussed in this section monitor system call traces to detect the attacks. System calls are operating system routines which provide the interaction between the applications and system resources such as memory, disks and peripherals. Given that the operations which alter the system state are handled through system calls, monitoring the applications at the system call level provides a suitable granularity for detecting the attacks [52].

Although numerous alternative detectors exist for detecting buffer overflow attacks, there are various traits which make the anomaly detectors discussed in this section suitable for the experiments. First, they employ different detection methodologies while monitoring the same resource (i.e. the system calls that the application

makes). This implies that evolving attacks against the detectors discussed in this section can demonstrate that evasion attacks can be evolved against a variety of detectors. Second, they provide feedback in the form of anomaly rates and delays which can be utilized to guide the search for the evasion attacks.

4.2.1 Stide

Forrest et al. [30] employed a methodology motivated by immune systems. This characterizes the problem as distinguishing 'self' from 'non-self' (normal and abnormal behaviours respectively). An event horizon is built from a sliding window applied to the sequence of system calls made by an application during normal use. The sequences formed by the sliding window are stored in a table which establishes the normal behaviour model. During the deployment (detection) phase, if the pattern from the sliding window is not in the normal behaviour database, it is considered a mismatch.

Input to the Stide detector [109] takes the form of the system call traces of an application for which the detector is trained. Specifically, Stide builds a normal database by segmenting the training data (of system call traces) into fixed length sequences [97]. To do so, a sliding window of size N is employed over the training dataset and the resulting system call patterns are stored in the normal database. During testing, the same sliding window size is employed on the data. The resulting patterns are compared against the normal database and if there is no match, a mismatch is recorded. Given a window size of N and system call trace length M, the anomaly rate for the trace is calculated by dividing the number of mismatches by the number of sliding window patterns (i.e. $M - N + 1$). The experiments provided in this thesis employ Stide with the default training parameters, which are listed in Table 4.1.

Table 4.1: Stide configuration parameters

Parameter	**Setting**
Sliding window length	6

4.2.2 Process Homeostasis (pH)

Process Homeostasis (pH) [91] is an anomaly detector based upon Stide which employs a detection methodology similar to Stide. pH is implemented as an extension to the Linux 2.2 Kernel. Therefore, pH monitors system calls more efficiently by capturing system calls directly at the kernel level as opposed to Stide which employs Strace[1] to capture system calls. pH monitors the changes in short sequences of system calls by employing look-ahead pairs. While employing the sliding window approach, pH does not store the sliding window patterns but records tuples, which consist of the current and past system calls and the sliding window location. Somayaji [91] established that the look-ahead method is more efficient to store and could potentially converge to a normal profile more quickly than the sequence method. Additionally, tolerization and sensitization concepts were introduced. Tolerization allows pH to improve false alarm rates by leaving out minimal anomalies, which are likely to be caused by slight changes in normal behaviour. Sensitization prevents abnormal behaviour from leaking into the normal behaviour database [91].

During training, a sliding window is employed over the training set and a normal database is built which can be represented as a three dimensional matrix. The dimensions are: (1) the current system call; (2) the previous system call on the sliding window; and (3) the location of the previous system call on the sliding window. During testing, the same sliding window is employed on the test data. If a given sliding window sequence produces a look-ahead pair which is not in the normal database, a mismatch is recorded. Similarly to Stide, given a window size of N and system call trace length M, the anomaly rate for the trace is calculated by dividing the number of mismatches by the total number of look-ahead pairs.

pH responds to attacks by slowing down the process. Delay is an exponential function of the locality frame count which aims to identify the clusters of anomalies. To this end, pH maintains a count of how many of the past LF (usually 128) system calls were anomalous. Process delays can affect the execution of a program substantially when a cluster of anomalies is observed. In the experiments, pH was employed with the default training parameters, which are listed in Table 4.2.

[1]Strace can be downloaded from `http://sourceforge.net/projects/strace/`.

Table 4.2: pH configuration parameters

Parameter	**Setting**
Look-ahead pair window size	9
Locality frame window size	128
Delay Factor	1
Suspend execve after	10 anomalies
Suspend execve duration	2 days
Anomaly limit	30
Tolerize limit	12

4.2.3 Process Homeostasis with a Schema Mask (pHsm)

Inoue and Somayaji [38] discussed the differences between look-ahead pairs and sequences. In their paper, the authors also proposed an improvement to pH based upon the random schema mask concept. Their main motivation was the observation that longer windows improve detection rates hence there exists a potential to increase the difficulty of generating mimicry attacks against pH (and indirectly Stide and variants). This improved pH is called pH with a schema mask (pHsm) in this thesis.

In pHsm, a longer sliding window (usually 20 [38]) is maintained and a number of taps (generally 9 which is also the sliding window size of the original pH [91]) are taken from the sliding window. The locations of the taps are determined randomly before training and this location information constitutes the schema mask. The authors state that introducing a schema mask creates another source of generalization [38]. The schema mask method has the following advantages: (1) pH can monitor a longer time window without increasing computational requirements (2) if the attacker does not know the schema mask, he/she would have to develop mimicry attacks based upon window sizes. The configuration parameters for the detector are detailed in Table 4.3.

Table 4.3: pHsm configuration parameters

Parameter	**Setting**
Look-ahead pair window size	20
Number of taps taken from the sliding window	9
Tap locations	Determined before training
Locality frame window size	128
Delay Factor	1
Suspend execve after	10 anomalies
Suspend execve duration	2 days
Anomaly limit	30
Tolerize limit	12

4.2.4 The Markov Model-Based Detector

The Markov Model is a statistical modelling technique which is useful for building probabilistic models of event sequences evolving in time. Markov Models have been utilized within the context of speech recognition and intrusion detection systems [114] [98] [33] as in the related case of a Finite State Automata representation [89]. The Markov Model was selected as an anomaly detector in this thesis because: (1) it can build probabilistic models using exemplars from only one class (i.e. normal behaviour) and (2) it can capture temporal (i.e. sequence) information without employing a sliding window.

Although higher order Markov Models exist where the current state depends upon a number of previous states, the Markov Model anomaly detector implemented in this thesis employs a first order Markov Model. In a first order Markov Model, the next state is only dependent upon the current state, and such an assumption is widely employed in these systems to reduce the number of 'free parameters' which require estimation. Given such an architecture, the following parameters are defined: a set of 'states' I, a probability transition matrix P, which describes the condition under which a transition between states occurs, and the probability distribution of the states λ.

In order to establish values for the above model parameters, the Baum-Welsh model is assumed [7]. Let I be a countable set of states, where $i, j \in I$ represent

states. If there are N states in I, a first order Markov model can be represented as a two-dimensional N by N matrix $P = \big(p(i,j) \mid i,j \in I\big)$. Let X_t define the state at step t. From the training data, the probability of transition from state i to j, $p(i,j)$, can be calculated as follows;

$$p(i,j) = \frac{C(X_t = j, X_{t-1} = i)}{\sum_i C(X_{t-1}) = i)} \tag{4.1}$$

In Equation 4.1, $C(X_t = j, X_{t-1} = i)$ is the number of times state j follows state i in the training data. The i^{th} row of the P is the probability distribution of moving to all states from state i. This probability distribution is also called λ_i. Such a scheme was used previously to act as a generic model for TCP packets with the objective of summarizing normal data [50]. For each test sequence, the Markov Model estimates log likelihood as follows:

$$LL_{seq} = \sum_T log\Big(p(X_t, X_{t-1})\Big) \tag{4.2}$$

Given that log likelihood in Equation 4.2 is dependent upon the attack size and therefore not suitable for distinguishing between attack and normal behaviour, the detection decision is based upon a characterization of state transition behaviour. A similar concept was employed in previous Markov Model detector approaches [98] and in Stide [109]. After the Markov Model is trained, a test sequence is presented. If there exists a transition in the test sequence which was not encountered during training (hence, $p(i,j) = 0$), a mismatch flag is set. The count of the mismatch flag is maintained, and the anomaly rate is defined in the manner provided in Equation 4.3.

$$Anomaly\ Rate = \frac{number\ of\ mismatch\ flags}{total\ number\ of\ transitions} \tag{4.3}$$

Such an anomaly rate implies that, if the test sequence follows the training model (i.e. normal behaviour), it will encounter zero or low numbers of mismatch flags. Thus a low anomaly rate is assigned. The configuration parameters for the Markov Model detector are provided in Table 4.4.

4.2.5 Auto-Associative Neural Network

In the experiments detailed in this thesis, an auto-associative neural network was employed as an anomaly detector. As opposed to the other detectors discussed in

Table 4.4: Markov Model parameters

Parameter	**Setting**
Order	First order
Number of states	223 (Number of system calls)
Training Algorithm	Baum-Welsh

the previous sections which employ sequence information, the input to the auto-associative neural network takes the form of the frequency distribution of system calls. This approach bears similarities to the detector employed by Kang et al. [43], which uses a bag of words representation as the detector input. In such a representation scheme, system calls are mapped to integers. Given N system calls, a system call trace is summarized into an N dimensional vector, where each element of the vector maintains the number of times the system call is encountered in the trace. Dividing each element in the vector by the total number of system calls provides the system call frequency distribution for the trace. As such, the resulting vectors constitute the normal behaviour characteristics. Since the frequency distribution of a trace is calculated after the trace is complete, the auto-associative neural network detector can be considered as an 'off-line' detector, which provides post-mortem analysis of the system call traces after they are executed.

Given the frequency distribution vector for the test trace, the detection is therefore based upon the divergence between the frequency distribution vectors of the test trace and the 'normal behaviour'. Although it is possible simply to store the frequency distribution vectors from the normal system call traces and facilitate detection using a similarity metric, this thesis employs auto-associative neural networks to build models of the normal behaviour system call frequency models. Training a machine learning algorithm on the frequency distributions creates a generic model which encapsulates the 'acceptable' system call frequency distributions for normal behaviour. Furthermore, training a detector on numerous frequency distributions allows the detector to 'learn' various characterizations of normal behaviour, hence producing a more robust detector.

Although different neural network approaches such as evolutionary neural networks were applied to anomaly detection [36], the application of an auto-associative neural network is new to anomaly detection. The auto-associative neural network is a bottleneck network which was employed in financial forecasting [60], principal component analysis [55], document classification [64] and novelty detection [40]. The main idea behind an auto-associative neural network is to develop models from one-class data and make decisions on the test data based upon the similarities to or diversions from the model which the auto-associative neural network encapsulates.

Given q dimensional data, a multilayer perceptron with p nodes in the hidden layer ($p \ll q$) and q nodes in the output layer is trained. Figure 4.1 shows an example of an auto-associative neural network with 6 inputs/outputs and 3 neurons in the hidden layer. The neural network aims to produce an output similar to the inputs provided during training. When a test input is presented, the neural network will produce the output similar to the input if the input is similar to what was encountered during training.

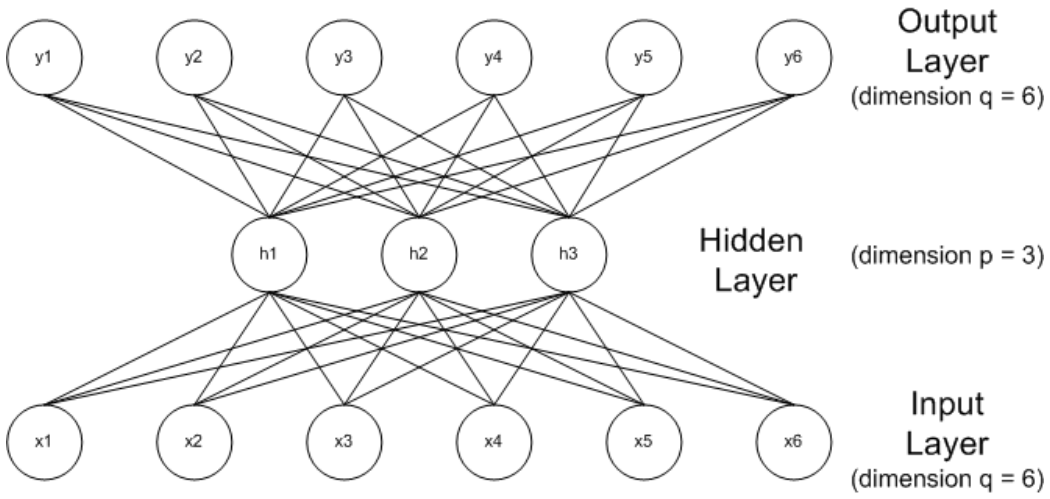

Figure 4.1: An auto-associative neural network

From the perspective of anomaly detection, if the applied input does not produce an output similar to the input, it is considered anomalous. In order to measure the degree of anomaly and produce an anomaly rate, the given input, $X = (x_1, x_2 \ldots x_n)$, and the produced output, $Y = (y_1, y_2 \ldots y_n)$, are compared using Euclidean distance. Euclidean distance in Equation 4.4 varies between 0 and 1, where larger numbers

indicate anomalous behaviour. The result of the Euclidean distance is then adjusted to vary between 0 and 100 so that the neural network produces an output similar to the anomaly rates which the other detectors such as Stide and pH produce.

$$d = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{4.4}$$

The normal system call traces detailed in Section 8.2.1 are summarized into 223 dimensional vectors where each dimension corresponds to the frequency of the system call associated with it. The hidden layer has 15 neurons and the output layer has 223 neurons. The network is trained with conjugant gradient backpropagation. During training, an error is calculated by subtracting the output Y from the target X. The neural network employs a mean square error, Equation 4.5, to test for termination.

$$mse = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2 \tag{4.5}$$

The training parameters for the Auto-assocative Neural Network are detailed in Table 4.5.

Table 4.5: Auto-associative Neural Network parameters

Parameter	**Setting**
No. of neurons in hidden layer	15
Hidden layer transfer function	Hyperbolic tangent sigmoid (tansig)
No. of neurons in output layer	223
Output layer transfer function	Linear (purelin)
Training function	Conjugant gradient backpropagation
Maximum epochs	1000
Minimum mean square error	1.00E-06

Chapter 5

Learning Algorithms Utilized

In this thesis, two Evolutionary Computing (EC) algorithms are investigated for exploit generation, namely Grammatical Evolution (GE) and Genetic Programming (GP). Grammatical Evolution [76] and Genetic Programming [53] [58] share several properties with Evolutionary Computation [24] relevant to this work.

1. **Representation:** the representation of individuals takes the form of a computer program, naturally fitting the objective of designing alternative malicious code.

2. **Quantification of the Performance:** the performance of the solutions is quantified using a fitness function in which there are no smoothness constraints on the form which such a function should take (unlike neural networks, where the cost function must typically be differentiable). As such, a fitness function provides a greater flexibility in expressing the objectives of the learning algorithm in terms meaningful to the application domain (vulnerability testing, in this case).

3. **Code Bloat:** the code bloat phenomenon in EC provides the attacker a way to hide the true intention of the attack. Introns correspond to the code segments in the program which do not contribute explicitly to the functional properties of the exploit. Introns are generally removed to produce concise solutions since they do not contribute to the functional operation of the individual. However, in the case of mimicry attacks, introns are beneficial to the attacker, if their statistical characterization matches normal behaviour as measured by the detector. During the fitness calculation, the feedback from the detector guides the evolution toward the intron code that matches the statistical characterization of normal behaviour. Consequently, this represents a scenario, in which introns

have a specific measurable contribution to the utility of the individuals; whereas the general practice is to drop the code corresponding to introns post training.

4. **Population-based:** being based upon a population of candidate solutions, the opportunity exists to build multiple exploits per run. Furthermore, with the ability to satisfy multiple (and sometimes conflicting) goals, maintaining a population of candidate solutions has the potential to provide solutions which optimize the different goals of the attacker.

5. **Multi-objective Fitness Function:** support for the multiple objectives of an attack is easy to provide without combining the objectives in a linear fashion (by assigning weights to each objective), which is a natural extension of qualifying the performance of solutions using the fitness function.

Section 5.1 introduces a generic Evolutionary Computation model to define the terminology and familiarize the reader with the concepts of EC. Grammatical Evolution and Genetic Programming are merely representations within the generic EC model. To this end, Grammatical Evolution is discussed in Section 5.2 and two versions of Genetic Programming are discussed in Sections 5.3 and 5.4.

This chapter introduces the above-mentioned algorithms without focusing on a particular problem. The problem-specific details of each algorithm, such as representation, function sets and fitness calculation are detailed further in Chapters 6, 7 and 8.

5.1 A Generic Evolutionary Computation Model

Evolutionary Computation draws inspiration from a Darwinian Model of Evolution, where a population of individuals (i.e. candidate solutions) is maintained. In the Darwinian Model of Evolution, the traits of an individual (i.e. the observable characteristics) are controlled by its genes. Similarly in EC, the traits of an individual are determined by its genes. A genotype consists of a string of genes. In EC, the genes in a genotype are generally in the form of a string of numbers (integer, binary or otherwise). Genotypes translate to phenotypes, which are the physical and observable characteristics of an individual. In other words, the genotype in EC represents

the individual (e.g. binary vs. integer representation), whereas the phenotype is its observable characteristics (e.g. the resulting program or expression).

The type and the representation of the phenotypes depends upon the genotype-phenotype mapping methodology, which is where the EC algorithms diverge. Numerous EC algorithms exist but in particular, Genetic Algorithms [70], Genetic Programming [53] [58] and Grammatical Evolution [76] are of interest.

In terms of genotype-phenotype mapping, Grammatical Evolution in Section 5.2 employs a context-free grammar to map the genotype to a phenotype in the form of a program. On the other hand, Genetic Programming algorithms detailed in Sections 5.3 and 5.4 employ function and terminal sets to map the genotype to the phenotype program. Therefore both GE and GP represent the individuals as programs in an arbitrary language, however the genotype-phenotype mapping method varies. In contrast, Genetic Algorithms (GAs) are not concerned with mapping the genotype to a phenotype since each individual can be represented simply as a string of numbers. A potential application of GAs is scheduling problems where the objective is to optimize a sequence of events. A comparison of GAs and GPs is made in Section 5.4.1, since the Genetic Programming detailed in Section 5.4 bears similarities to GAs.

In EC, the individuals in a population reproduce and create children for the next generation. Natural selection ensures that the favourable characteristics of the individuals will have more chance to propagate to the next generation. This implies that the individuals with favourable phenotypes will have a better chance to survive and reproduce, although the the stochastic nature of the selection process allows weaker individuals to survive as well, albeit with a smaller chance.

The fitness of an individual is expressed relative to the phenotype whereas the reproduction operators such as mutation and crossover are applied to the genotype, where the latter is part of a process for stochastic credit assignment. In such a scheme, fitter individuals have a better chance of transferring their genetic material to the next generation. EC algorithms implement reproduction operators in numerous ways such as crossover, mutation and swap. Of course, reproduction in EC differs from reproduction in biology since it is more targeted more toward the machine learning paradigm.

Within the context of machine learning, reproduction in EC facilitates a search for 'fitter' solutions within the search space. The search space can be defined as the set of all candidate solutions to a problem along with a notion of 'distance' between the solutions. Three reproduction operators are of interest in this work: crossover, mutation and swap. In crossover, two parents, which are selected from the population, exchange genotypes to create children. In other words, children combine the existing genetic material from both parents. Mutation, however, involves only one parent, where the child is created by altering one or more genes in the parent. Therefore, the child combines new genetic material with the genetic material from the parent. Similarly, swap selects a parent and uniformly swaps two (or more) genes to create a child. In other words, the child contains the same genetic material with the parent but with a different order. By providing different combinations of the existing genetic material (i.e. crossover and swap) and by introducing new genetic material (i.e. mutation), reproduction operators provide the means to create children. However, all three operators are stochastic, therefore there is no guarantee that the resulting children have not been encountered previously.

In order to be able to compare individuals to determine the fittest individual, a method for fitness assignment is necessary. For this purpose, EC employs a fitness function which rewards individuals as they get better at achieving the target objective(s). A fitness function is problem dependent. For example, in case of a regression problem, a typical fitness function will assign fitness based upon how close the individual is to representing the target function. In the case of generating attacks against a detector, the fitness function 'rewards' individuals which produce fewer alarms or anomalies.

Furthermore, the training of an EC algorithm involves the definition of selection and replacement operators. A selection operator defines which individuals in a population will be selected as parents to participate in reproduction and the replacement operator determines which individuals 'survive' to the next generation. In the case of a uniform selection scheme, every individual in the population has the same chance of reproducing regardless of their fitness. However, in fitness proportional selection, individuals with greater fitness have a greater probability of reproducing. In terms

of replacement, the children can replace their parents or other individuals in the population which have lower fitness.

Various evolution strategies can be employed for training EC algorithms. In a generational approach, every individual in the population participates in reproduction, therefore the entire population can change in the next generation. Alternatively, tournament selection selects a smaller set of individuals from the population for reproduction. Commonly, the children of the better performing half of the tournament replace the individuals in the worse half of the tournament. Such a replacement scheme is called 'elitist' which implies that the fittest individual is always retained in the population.

As discussed previously in this chapter, Grammatical Evolution and Genetic Programming differ mainly in their representation and genotype-phenotype mapping. In Sections 5.2, 5.3 and 5.4, the discussion of each algorithm is divided into four subsections. Representation sections deal with not only the genotype representation (linear versus. tree representation) but also the genotype-phenotype mapping scheme (context-free grammars versus function and terminal sets). Training sections define the selection and replacement operators (generational versus tournament selection) and provide a high level overview of the training process. The fitness function sections discuss the details of the fitness calculation pertinent to the Evolutionary Computation, including the support for multi-objective optimization, when applicable. The discussions of the problem specific details of the fitness calculation are left to the corresponding chapters (namely, Chapters 6, 7 and 8). The search operator sections detail the reproduction operators such as crossover, mutation and swap operators which are developed based upon the representation and the problem being solved.

5.2 Grammatical Evolution

Grammatical Evolution is an Evolutionary Computation method which can produce programs in an arbitrary language defined by a context-free grammar. As discussed previously in this chapter, the characteristics which make GE suitable for the experiments in this thesis are that the solutions take the form of computer programs and the performance of a solution is determined by a fitness function. The former

characteristic allows GE to generate buffer overflow attacks in an arbitrary language, subject to the limitations of the representation, whereas the latter characteristic provides a method for describing the various goals of buffer overflow attacks in order to facilitate the search process. Furthermore, the context-free grammar in GE ensures that the genotype to phenotype mapping process produces valid programs (with respect to the grammar), regardless of the search operators. This is a clear advantage of GE over other EC algorithms, particularly GP, in which the function and terminal sets should be developed to prevent creating invalid instructions such as a division by zero condition.

In GE, individuals are composed of strings of codons where each codon is represented as an integer. In GE, codons are analogous to the genes of a genotype. The genotype is the specification of an individual (e.g. binary vs. integer representation) whereas the phenotype is its observable characteristics (e.g. the resulting program or expression). A genotype (a string of codons) is translated into a phenotype by employing a context-free grammar, typically in Backus-Naur form. The use of the context-free grammar ensures the validity of the individuals, regardless of the how search operators alter the genotype.

5.2.1 Representation

In GE, individuals are represented essentially as sequences of integers. The grammar in Backus-Naur form (BNF) defines the phenotype (i.e. the program) upon which the fitness is calculated. Starting from the first codon (gene), GE applies the grammar to the individual until the complete program is generated.

A sample GE grammar is provided in Figures 5.1. Furthermore, Figure 5.2 provides an example of a GE individual in genotype form.

The mapping begins from the start symbol `<expr>`, which can take 4 different forms according to rule (A) in Figure 5.1. GE determines which grammar rule to utilize based upon the first codon of the genotype. For example, in Figure 5.2, the first codon is 28 and the rule which applies to `<expr>` (i.e. rule (A) in Figure 5.1) has four types. Calculating the modulus, $28 \; mod \; 4 = 0$, the first codon is mapped to the rule (A.0) in Figure 5.1 and the expression becomes `<expr> <op> <expr>`. Using the

```
(A) <expr> ::= <expr> <op> <expr>     (0)
         | ( <expr> <op> <expr> )     (1)
         | <pre-op> ( <expr> )        (2)
         | <var>                      (3)

(B) <op> ::= +      (0)
           | -      (1)
           | /      (2)
           | *      (3)

(C) <pre-op> ::= Sin        (0)
               | Cos        (1)

(D) <var> ::= X      (0)
            | Y      (1)
            | 1.0    (2)
```

Figure 5.1: A sample GE grammar

28	35	200	91	130	75	47	170	68	84	181	123	56	210

Figure 5.2: The sample individual in genotype format

same process, the first parameter of `<expr> <op> <expr>` is mapped by using the second codon. The mapping process continues until a complete program is formed or the maximum number of wrapping events is reached. Using the first 8 codons, the genotype in Figure 5.2 is mapped to the phenotype `X * Cos ( 1.0 )`. The mapping process for the sample individual is summarized below:

```
                    <expr>
28  mod 4 = 0       <expr> <op> <expr>
35  mod 4 = 3       <var>  <op> <expr>
200 mod 4 = 0         X    <op> <expr>
91  mod 4 = 3         X     *   <expr>
130 mod 4 = 2         X     *   <pre-op> ( <expr> )
75  mod 2 = 1         X     *   Cos      ( <expr> )
47  mod 4 = 3         X     *   Cos      ( <var>  )
170 mod 3 = 2         X     *   Cos      ( 1.0 )
```

If GE reaches the end of the individual before the mapping is complete, it 'wraps around' and starts from the first codon. Note that although the codons remain the same in integer value, a codon can be mapped to different values based upon the current rule which GE utilizes. If the individual does not map completely after a predetermined number of wraps, the mapping stops and the individual is assigned the minimum fitness value. Conversely, if GE forms the expression completely without using all codons, the remaining codons are ignored. The GE implementation employed in this thesis utilizes a fixed length representation, which implies that all individuals have the same number of codons.

5.2.2 Training

The main GE training function is provided in Algorithm 1. During the GE training, fitness sharing is employed to encourage diversity in the population. The radius for fitness sharing changes as the new individuals are introduced to the population, therefore the radius is recalculated every 10 generations.

Given the number of niches and the set of attributes upon which the shared fitness is calculated, the radius is determined by the minimum and maximum values,

Algorithm 5. Fitness calculation involves dividing the raw fitness by the niche count. The niche count is based upon the individual's similarity to other individuals. In other words, shared fitness decreases if the individual is similar to many other individuals. The calculation of shared fitness is detailed in Algorithm 4. The calculation of raw fitness is problem dependent and is discussed in Section 6.2.1.

In the experiments for this thesis, O'Neill's GE implementation [76], which adopts a generational approach, is employed as defined in Algorithm 1. In a generational approach, the entire population can change in every generation. A recent analysis [41] compared the steady-state approach with the generational approach and concluded that both methods succeeded in solving the test problems albeit with different rates of convergence and amount of code growth.

Employing a generational approach implies that all the individuals have the chance to participate in the search at each generation. At each generation, the population is arranged uniformly in pairs for crossover. With a certain probability, the single point crossover detailed in Algorithm 6 is applied to the pairs. The crossover operator produces two children. The children replace the corresponding parents if their fitness value is greater. This scheme is elitist in nature which implies that the best individual is always maintained in the population. The training continues until GE reaches the maximum number of generations limit. Algorithm 1 does not provide any details on fitness calculation since the fitness calculation is problem specific and will be discussed in later chapters. The reader can assume that any change in an individual instigates a fitness calculation.

The GE and GP algorithms discussed in this chapter require uniform selection and the application of search operators with given probabilities. Therefore, Algorithm 2 provides a high level definition of a random number function whereas Algorithm 3 defines the algorithm which allows the application of a search operator with a given probability. These functions are also employed in the discussion of Linear Genetic Programming, Sections 5.3 and 5.4. In the algorithms discussed in this chapter, $|DS|$ defines the number of elements in the data structure DS.

Algorithm 1: Main GE training loop

Input : Number of generations GC, probability of crossover P_{xo}, niche count q

Output: Population of individuals $population[\,]$

```
1  for gen = 1 to GC do
2      if gen mod 10 = 0 then
3          CalculateRadius(population[ ], q);
4      end
5      used = ∅ ;
6      for pairs = 1 to |population[ ]|/2 do
7          p1 = Random( x ∈ ℤ+ | (x ≤ |population[ ]|) ∧ (x ∉ used)) ;
8          used = used ∪ {p1} ;
9          p2 = Random( x ∈ ℤ+ | (x ≤ |population[ ]|) ∧ (x ∉ used)) ;
10         used = used ∪ {p2} ;
11         if TestProb(P_xo) then
12             children[ ] = ApplyXO(population[p1], population[p2]) ;
13             if children[1].fitness > population[p1].fitness then
14                 population[p1] = children[1];
15             end
16             if children[2].fitness > population[p2].fitness then
17                 population[p2] = children[2];
18             end
19         end
20     end
21 end
```

Algorithm 2: Random(...) function

Input : S, specification of a set of numbers

Output: r, which is an element of set S

```
// rand() is an arbitrary pseudo-random number generator
1  r = rand( r ∈ S);
```

Algorithm 3: TestProb(...) function

Input : A probability *prob*, on which the decision is made

Output: *outcome*, which is 1 if the test is true and 0 if false

```
1 r = Random( x ∈ ℝ | 0 ≤ x ≤ 1);
2 if r < prob then
3     outcome = 1;
4 else
5     outcome = 0;
6 end
```

5.2.3 Fitness Function

EC employs a fitness measure to quantify the optimality of individuals. Therefore, the fitness function 'rewards' individuals so that they can be evaluated. Commonly, the individuals with higher fitness values will have a better chance of survival. The definition of the fitness function depends upon the problem. For some problems the output of the individual can be compared with the desired output. The smaller the difference between the output and the desired value will result in a greater fitness value. On the other hand, in certain problems, fitness can be calculated in addition to the outcome, by the consequences of the execution of the individuals. For example, in the mimicry attack generation problem, the fitness of an attack depends not only upon the success of the attack but also upon the stealth of the attack.

Given that the fitness function definition is problem specific, the details of the fitness functions are covered in Section 6.2.1. This section focuses on the fitness sharing concept which GE employs in the experiments in Chapter 6.

Fitness Sharing

Although EC is a population-based search algorithm, schema theory indicates that as the fitter individuals reproduce, the population diversity decreases, resulting in the population converging on a small region of the search space [59]. In order to encourage the population to provide multiple unique solutions, the fitness sharing concept [19] [68] is borrowed from Genetic Algorithms. In this case, the fitness of an individual

is discounted in proportion to the similarity with others in the population. Shared fitness for an individual i is calculated based upon the raw fitness of the individual divided by the niche count m_i.

$$f_{shared} = \frac{f_{raw}}{m_i} \tag{5.1}$$

In Equation 5.1, the niche count m_i increases as the similarity of an individual to other individuals increases. This implies that the shared fitness decreases as the individual becomes more similar to the other individuals in the population. The niche count is calculated over the population of n individuals as detailed in Equation 5.2.

$$m_i = \sum_{j=1}^{n} sh(d_{i,j}) \tag{5.2}$$

In Equation5.2, $d_{i,j}$ denotes the Euclidean distance between individual i and j, which is provided in Equation 5.3. In the example provided in this section, such Euclidean distance is calculated based upon two attributes of the individuals, denoted as X and Y. However, in general, it can be calculated on one or more attributes. The individual attributes employed in the thesis experiments are discussed further in Section 6.2.1.

$$d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{5.3}$$

If distance d is smaller than the determined radius σ, sharing function $sh(d)$ returns a value between 0 and 1, which increases as the distance decreases. Hence the sharing function can be expressed as:

$$sh(d) = \begin{cases} 1 - \frac{d}{\sigma} & \text{if } d < \sigma \\ 0 & \text{if } otherwise \end{cases} \tag{5.4}$$

Equation 5.4 implies that, for individuals with similar attributes, the niche count will increase causing the shared fitness to decrease. σ is estimated from the population by determining the current extremes. This takes the form of the minimum and maximum values of the attributes X and Y. Given a number of niches q, σ can be calculated as defined in Equation 5.5 [19].

$$\sigma = \frac{\sqrt{(max(X) - min(X))^2 + (max(Y) - min(Y))^2}}{2\sqrt{q}} \tag{5.5}$$

As the population changes during training, the boundaries formed by the attributes X and Y will change as well. Since determining the boundaries requires a pass of the entire population, σ is calculated every few generations (5 in the thesis experiments).

An overview of the fitness calculation function is provided in Algorithm 4. Furthermore, the overview of the function, which calculates the radius, is detailed in Algorithm 5.

Algorithm 4: CalculateFitness(...) function

Input : Index of the individual *ind*, Raw fitness of the individual f_{raw}, the attributes of individuals on which the shared fitness is calculated ,X and Y.

Output: Shared fitness f_{shared}

```
1  m_ind = 0 ;                                   // Niche count of the individual
2  for i = 1 to |population[ ]| do
3      if i ≠ ind then
4          x_diff = population[i].X − population[ind].X;
5          y_diff = population[i].Y − population[ind].Y;
6          d = sqrt((x_diff)^2 + (y_diff)^2) ;
7          if d < σ then
8              m_ind = m_ind + (1 − d/σ);
9          end
10     end
11 end
12 f_shared = f_raw/m_ind
```

5.2.4 Search Operators

A crossover operator allows new individuals to be created from the existing individuals in the population. Typically, the crossover operator involves selecting two individuals from the population (i.e. the parents) and producing two new individuals (i.e. the children) by combining material from each parent.

Algorithm 5: CalculateRadius(...) function

Input : Attributes of individuals on which the shared fitness is calculated X and Y, number of niches q, a population of individuals population[ ].

Assumptions: Assumes that $max(X)$ and $min(X)$ functions exists to determine the largest and smallest values in array X.

Output: The current radius σ

1 $\sigma = \frac{\sqrt{(max(population[\ ].X) - min(population[\ ].X))^2 + max(population[\ ].Y) - min(population[\ ].Y))^2}}{2\sqrt{q}}$

GE employs single point crossover where a crossover point is selected uniformly. The crossover operator splits the parents into two fragments and combines the first fragment from the first (second) parent and the second fragment from the second (first) parent to create the first (second) children. Single point crossover is detailed further in Algorithm 6.

Algorithm 6: ApplyXO(...) function

Input : A pair of individuals $parent1$ and $parent2$, from which the children are created

Output: Two children $children[\,]$

```
// Note that the individuals are fixed length
```

1 $len = |parents[1].gene[\,]|$;
2 $xo\_point =$ `Random(` $x \in \mathbb{Z}^+ \mid x \leq len$`)` ;
3 **for** $loc = 1$ **to** len **do**
4 **if** $loc < xo\_point$ **then**
5 $children[1].gene[loc] = parent1.gene[loc]$;
6 $children[2].gene[loc] = parent2.gene[loc]$;
7 **end**
8 **else**
9 $children[1].gene[loc] = parent2.gene[loc]$;
10 $children[2].gene[loc] = parent1.gene[loc]$;
11 **end**
12 **end**

5.3 Linear Genetic Programming

In a similar fashion to Grammatical Evolution, Linear Genetic Programming (GP) produces programs in an arbitrary language. However GP differs from GE in genotype to phenotype mapping by employing function and terminal sets. Linear GP shares common traits with both the Genetic Algorithms (GAs) [69] and the traditional tree structured GP approaches [53], hence both methodologies are discussed to put Linear GP [6] into perspective.

Genetic Programming differs from Genetic Algorithms in representation. In GAs the individuals are represented as a string of numbers whereas traditional GP employs a tree data structure. In GAs, the genotype may or may not require mapping to a phenotype (i.e. the fitness function can simply use the sequence of numbers for evaluation), whereas GP employs a function set and a terminal set to translate the tree representation into a program in an arbitrary language. The terminal set defines the variables and constants of a program whereas the function set defines the functions which utilize the terminal set.

Similar to Genetic Algorithms, Linear Genetic Programming employs a string of numbers as genotype. Furthermore, Linear GP employs an instruction set, which consists of the function and terminal sets, to map the genotype to programs in an arbitrary language. Therefore, linear GP is similar to the GAs in terms of the linear representation of individuals and similar to traditional tree-structured GP in terms of the mapping scheme.

Two training configurations are provided for linear GP in Algorithms 7 and 12. The first configuration, which is detailed in this section, is employed in evolving buffer overflow attacks in Chapter 7. The second configuration detailed in Section 5.4 utilizes Pareto ranking and is employed in evolving mimicry attacks against anomaly detectors in Chapter 8.

5.3.1 Representation

The specific representation utilized in the Linear GP employed in this thesis defines instructions as 4 bytes where 2 bytes are allocated for the function identifier (i.e. the opcode) and one byte is allocated for each terminal of the function (i.e. the operands).

This implies that all instructions have the same number of bytes. Therefore, the first two bytes of the instruction identify the function to be used whereas the last two bytes identifies which terminal(s) the function uses.

A sample instruction set for GP is provided in Figure 5.3. The sample instruction set consists of assembly instructions which can take registers or constants as parameters, which are provided in Figure 5.3 as well. Although linear GP employs 4 bytes to define an instruction, the example provided in Figures 5.3 and 5.4 employs 1 byte to define an individual for the sake of simplicity.

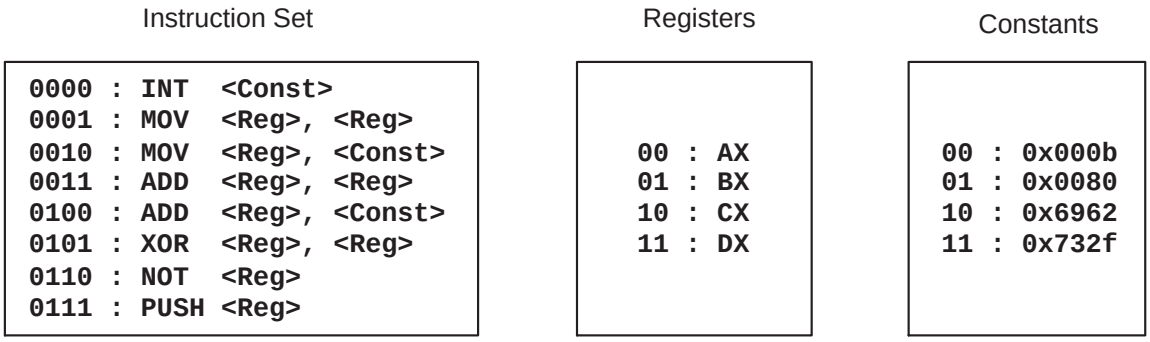

Figure 5.3: Sample GP instruction set and parameters

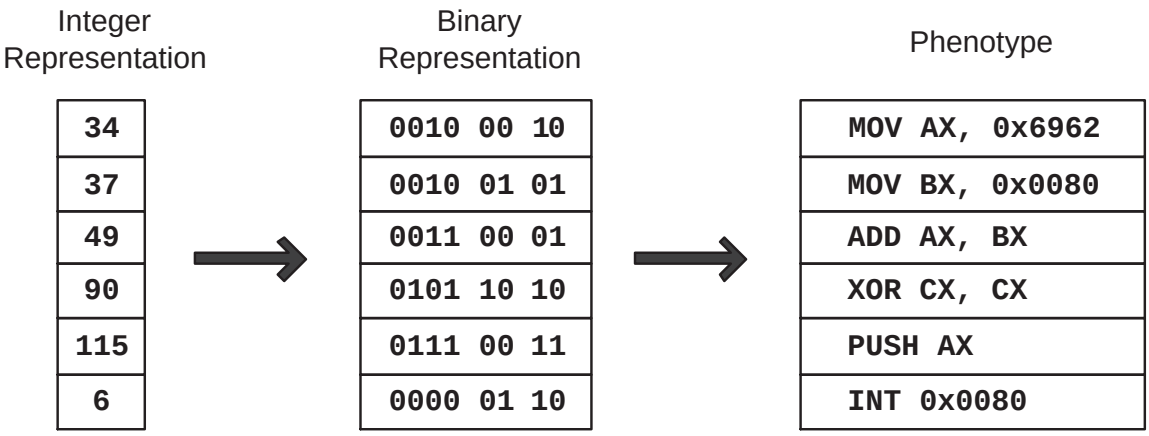

Figure 5.4: An example of a genotype-phenotype mapping for a linear GP individual

Figure 5.4 presents a linear GP individual the genotype of which is represented in integer format. Based upon the mapping defined by the instruction set and parameters, the genotype is mapped to a phenotype. To do so, each instruction is converted

to a binary representation. The first 4 bits of each gene define the instructions, whereas the last four bits are allocated for the definition of the instruction parameters (2 bits for each parameter, up to two parameters). For example, the integer 34 is equivalent to 00100010 in binary. The first four bits, 0010, map to the instruction `MOV <Reg>, <Const>`. In the list of registers in Figure 5.3, the 00 maps to the register `AX` and the 10 maps to the constant `0x6962`. If the instruction has one parameter (e.g. in case of integer value 115, which maps to `PUSH AX`), the last two bits (i.e. 11) are ignored. During the initialization of the population and the application of the mutation operator, as new instructions are introduced to the individuals, the validity of the individuals is checked to ensure that the instruction and parameter fields produce a valid instruction.

In addition, two forms of instruction count are considered, namely the fixed length and the variable length. In the simpler LGP model discussed in this section, each individual is limited to a fixed instruction count, or page, implying that an equal number of instructions is exchanged during crossover [37]. Conversely, in the case of the second LGP model, Section 5.4, instruction counts of the children are allowed to to vary relative to their parents, where the multi-objective model is able to provide explicit feedback as to the utility of this property.

The main reason for replacing GE with linear GP in the experiments is the effectiveness of the search operators in a GE representation. Namely, the search operators in GE were not particularly efficient at manipulating register references during the initial experiments in evolving buffer overflow attacks against misuse detectors [48]. In particular, the alterations made to the earlier codons in GE affect how the later codons are mapped. This implies that even a single codon change in GE can change the context of the individual substantially. The linear representation provides a more direct method for evolving buffer overflow attacks successfully, where GP can modify the operands more efficiently.

5.3.2 Training

In the training scheme detailed in Algorithm 7, GP employs the tournament selection provided in Algorithm 8, where four individuals are selected uniformly from the

population and sorted according to their fitness. The fitness calculation is problem specific and is carried out by a runtime environment which is discussed in Section 7.2.2. Crossover in Algorithm 9, single instruction mutation in Algorithm 10 and swap in Algorithm 11 aim to facilitate the search and are discussed in Section 5.3.4. The better performing half of the tournament undergoes mutation, crossover and swap operators and the resulting children replace the lower half of the tournament.

As opposed to a generational approach, this is a steady state approach where only a small subset of individuals produce children at each iteration. The replacement scheme is elitist in which the better performing individuals are always maintained in the population. The training continues until the GP reaches the maximum number of iterations limit. For the sake of simplicity the fitness calculation Algorithm 7 does not cover the fitness calculation, which is detailed in Section 7.2.2.

5.3.3 Fitness Function

In the thesis experiments on evolving buffer overflow attacks against misuse detectors in Chapter 7, the fitness of an individual is calculated by the runtime execution environment. The runtime execution environment executes the individual symbolically and returns a fitness value between 0 and 10, where larger values indicate fitter individuals, based upon the success of the individual. Given that the fitness calculation is problem specific, it is not covered in this section. The fitness calculation using a runtime execution environment will be discussed in Section 7.2.2.

5.3.4 Search Operators

Crossover operators combine the existing genetic material in the population to produce children with a new ordering, whereas the mutation operators introduce new genetic material to the population. The GP training scheme in Algorithm 7 employs page-based crossover, single-instruction mutation and instruction swap operators, which are discussed in this section.

Algorithm 7: Main GP training loop

Input : Number of iterations $NumIterations$, probability of crossover P_{xo}, probability of mutation P_{mut}, probability of swap P_{swap}

Output: Population of individuals $population[\,]$

```
1  for iter = 1 to NumIterations do
2      t[ ] = TournamentSelection(population[ ]) ;
3      if TestProb(P_xo) then
4          new_inds[ ] = ApplyPageBasedXO( population[t[1]], population[t[2]]);
5      end
6      if TestProb(P_mut) then
7          new_inds[ ] = ApplyMutation(new_inds[1], new_inds[2]);
8      end
9      if TestProb(P_swap) then
10         new_inds[ ] = ApplySwap(new_inds[1], new_inds[2]);
11     end
       // Children replace the lower half of the tournament
12     population[t[3]] = new_inds[1] ;
13     population[t[4]] = new_inds[2] ;
14 end
```

Algorithm 8: TournamentSelection(...) function

Input : Population of individuals *population*[ ]

Output: The tournament, *tournament*[ ], which contains population index of four individuals, which are sorted according to their fitness

1 $C = \emptyset$;
2 **for** $i = 1$ **to** 4 **do**
3 $\quad C = C \cup \{\texttt{Random(}\ x \in \mathbb{Z}^+ \mid x \leq |population[\ ]| \wedge x \notin C\texttt{)}\}$;
4 **end**
5 **for** $i = 1$ **to** 4 **do**
6 $\quad c_i = x \in C \mid \forall c_j \in C \wedge population[c_i].fitness \geq population[c_j].fitness$;
7 $\quad$ tournament[i] $= c_i$;
8 $\quad C = C - \{c_i\}$;
9 **end**

Page-Based Crossover

In Page-Based Linear GP, individuals consist of pages where each page contains an equal number of instructions. Therefore, page-based crossover in Algorithm 9 selects a page uniformly from each parent and exchanges their contents. In this crossover scheme, the length of the children remains the same. This crossover scheme is employed to evolve buffer overflow attacks against misuse detectors in Chapter 7.

Single-Instruction Mutation

The single-instruction mutation is applied to a single individual. In this mutation scheme, an instruction is selected with uniform probability and replaced with a different instruction from the instruction set, again chosen with uniform probability. The single-instruction mutation scheme, which is detailed in Algorithm 10, is employed mainly in evolving buffer overflow attacks against misuse detectors in Chapter 7 but later is replaced with instruction-wise mutation in evolving mimicry attacks against anomaly detectors in Chapter 8. The single-instruction mutation mutates only one instruction per application, whereas the instruction-wise mutation, which is discussed in Section 5.4.4, mutates multiple instructions per application, since each instruction

Algorithm 9: ApplyPageBasedXO(...) function

Input : A pair of individuals from which the children are created $parents[\,]$, Page size $PgSize$

Assumptions: $\rightleftharpoons$ operator exists which swaps two genes

Output: Resulting children $children[\,]$

```
1 children[ ] = parents[ ] ;
2 xo_point1 = Random( x ∈ ℤ+ | x < |parents[1]| ∧ x mod PgSize = 0) ;
3 xo_point2 = Random( x ∈ ℤ+ | x < |parents[2]| ∧ x mod PgSize = 0) ;
4 for loc = 1 to PgSize do
      // Exchange the pages
5     children[1].gene[xo_point1 + loc] ⇌ children[2].gene[xo_point2 + loc];
6 end
```

of the individual is tested for mutation. The utilization of instruction-wise mutation is due mainly to the increased search space size when evolving mimicry attacks against anomaly detectors, which necessitates a more effective mutation operator. The discussion of search space sizes is made in Sections 7.2.4 and 8.2.4.

Algorithm 10: ApplyMutation(...), single-instruction mutation function

Input : Set of opcodes $FunctionSet$, set of operands $TerminalSet$, a pair of individuals from which the children are created $parents[\,]$

Output: Resulting children $children[\,]$

```
1 children[ ] = parents[ ];
2 IS = {i = (f, p1, p2) | f ∈ FunctionSet ∧ p1, p2 ∈ TerminalSet} ;
3 for ind = 1 to |children[ ]| do
4     loc = Random( x ∈ ℤ+ | x ≤ |children[ind].gene[ ]|) ;
5     children[ind].gene[loc] = Random(i ∈ IS) ;
6 end
```

Instruction Swap

The swap operator selects two instructions from the same individual with uniform probability and interchanges their respective positions, thus providing the basis for investigating different permutations of the same instruction. The instruction swap operator, detailed in Algorithm 11, is employed both in evolving buffer overflow attacks against misuse detectors in Chapter 7 and in evolving mimicry attacks against anomaly detectors in Chapter 8.

Algorithm 11: ApplySwap(...) function

Input : A set of parents, upon which the swap is applied $parents[\,]$

Assumptions: $\rightleftharpoons$ operator exists which swaps two genes

Output: Resulting children, $children[\,]$

1 $children[\,] = parents[\,]$;
2 **for** $ind = 1$ **to** $|children[\,]|$ **do**
3 $instr1 =$ `Random(` $x \in \mathbb{Z}^+ \mid x \leq |children[ind].gene[\,]|$`)` ;
4 $instr2 =$ `Random(` $x \in \mathbb{Z}^+ \mid x \leq |children[ind].gene[\,]| \wedge x \neq instr1$`)` ;
`// Exchange the instructions`
5 children[ind].gene[instr1] $\rightleftharpoons$ children[ind].gene[instr2] ;
6 **end**

5.4 Linear Genetic Programming with Pareto Ranking

The linear GP framework discussed in this section differs from the GP framework discussed in Section 5.3 in a number of ways.

Representation: The previous linear GP framework has a fixed length representation, whereas this GP framework supports variable length representation. Specifically, in evolving buffer overflow attacks at the assembly level, the individual length is usually fairly small (tens of instructions as opposed to hundreds). On the other hand, evolving mimicry attacks against anomaly detectors involve individuals which can be very concise or very long. Assuming a variable length representation provides GP with the freedom to establish which instruction count is the most appropriate by making use of a Pareto multi-objective fitness evaluation. Thus, unlike the fixed length model in which the range of instruction counts was established *a priori*, considerable variation might exist in the instruction counts of the individuals.

Support for Multi-Objective Optimization: The fitness calculation in the previous GP framework in Section 5.3 is not particularly multi-objective. That is to say, although there are steps which need to be completed for the attack to succeed, the ultimate goal of the exploits is to deploy a system call (i.e. execve). However, in evolving mimicry attacks against anomaly detectors, the individuals aim to optimize multiple characteristics of a mimicry attack such as the anomaly rates, delays and attack lengths. Therefore in the experiments which involve evolving mimicry attacks against anomaly detectors (Chapter 8), the linear GP detailed in Section 5.3 is extended to support a Pareto rank-based multi-objective fitness function.

Search Operators: Given a variable length representation, new search operators are introduced, namely cut and splice crossover and the instruction-wise mutation. The cut and splice crossover operator allows the production of children which can be shorter or longer than the parents. Furthermore, the instruction-wise mutation provides a more effective means to introduce new genetic material to the population.

5.4.1 Representation

Sharing the same representation scheme with Linear Genetic Programming in Section 5.3, the representation defines instructions as 4 bytes where 2 bytes are allocated for the opcode and two operands are each allocated one byte. In other words, the first two bytes of the instruction identifies the function to be used whereas the last two bytes identify which terminal(s) the function uses. The individuals are variable length, which implies that the children can be longer or shorter than their parents. However, the genotype to phenotype mapping example provided in Section 5.3.1 still applies to the GP framework detailed in this section.

Given that a mimicry attack can be expressed as a sequence of system calls, the Genetic Programming detailed in this section shares similarities with GAs in the sense that the individuals can be represented as a sequence of integers. In order to identify the similarities between the GP detailed in this section and GAs, a distinction between GAs and GPs is necessary. The main difference lies in the mapping of the genotype to phenotype in which GP aims to map the string of numbers (i.e. genotypes) to a program, whereas GA retains the string of number representation. Although the mimicry attacks which GP generates consist of instructions *and* parameters (i.e. system calls can be evolved with parameters), the fitness calculation, which is carried out mainly by the target anomaly detector, is based upon the ordering of the instructions. Therefore, although the mimicry attack generation methodology is GP, it can be implemented as a GA without the loss of generality.

5.4.2 Training

The training loop is detailed in Algorithm 12, where Pareto ranking is employed for multi-objective optimization. At each iteration, the Pareto Ranking function in Algorithm 14 calculates the rank of each individual by using the Pareto dominance concept (Algorithm 17), which is discussed in Section 5.4.3.

At each iteration, two parents are selected uniformly from the population. Two children are created by applying cut and splice crossover, instruction-wise mutation and swap, Algorithms 18, 19 and 11, respectively. It is important to note that,

compared with single instruction mutation, instruction-wise mutation tests each instruction for mutation, hence providing greater change in an individual.

Using a replacement scheme similar to Kumar et al. [57], the children are appended to the population (of size N) after the search operators, hence producing $N + 2$ individuals. This scheme bears similarities to both generational and steady state approaches. Similar to a steady state approach, both parents and the children can take part in the next iteration. Furthermore, similar to the generational approach, the children are compared against the entire population, although the replacement does not alter the entire population. The extended population with $N+2$ individuals is sorted according to the Pareto ranks using Algorithm 14 and the lowest ranked two individuals are discarded from the population (Algorithm 15) hence restoring the population size to N. This replacement scheme is elitist in nature since it does not cause the loss of non-dominated solutions. In other words, at each training step, the population either moves toward the Pareto front or remains the same. For the sake of simplicity, fitness calculation in Algorithm 12 does not cover the fitness calculation, which is detailed in Section 8.2.3.

Kumar et al. [57] employed rank frequency distribution functions to test for convergence and early termination. The frequency distribution summarizes the number of individuals for each rank. If the frequency distribution of the ranks remains unchanged over a pre-defined number of iterations (five in the thesis experiments), the population is considered converged and the training is terminated. Similarly, the GP employed in the thesis experiments maintains a frequency density function of the ranks by utilizing Algorithm 16.

Unlike other well known Pareto multi-objective models [14] [115] [18] [28], the approach of Kumar et al. [57] does not make use of niche-based diversity metrics. This is a considerable advantage under the GP context as most niching metaphors are based upon the concept of a distance calculation (i.e., in terms of the original domain input attributes). This might be a problem when evolving payloads for the buffer overflow attacks because domain attributes and metrics for comparing such payloads (i.e. a phenotypic comparison) such as 'edit distance' are notoriously difficult to provide effective definitions for. Indeed, the niche-based diversity metric of GE in

Section 5.2.3 utilized such a scheme and assumed that it was sufficient to make a multi-model assumption relative to the fitness landscape as opposed to the genotypic representation space. However, in cases of evolving exploit code, such appropriate measures are not available.

5.4.3 Fitness Function

Various methods exist for supporting the multi-objective fitness calculation. In the most straightforward method, the objectives are combined into one fitness value by assigning weight values to each objective. Although this approach is simple and easy to implement, the resulting fitness value is sensitive to changes in the weight values. Other methodologies exist such as niching and fitness sharing [19] [68], which aim to maintain a diversity in the population by penalizing individuals which produce similar outcomes. The experiments on optimizing buffer overflow characteristics in Chapter 6 employ both of these methods.

Goldberg [35] introduced the Pareto Ranking concept with the objective of producing a scalar fitness from a vector of objectives without combining them in an ad-hoc fashion. The Pareto optimality can be defined as follows: in a minimization problem with the objective vectors F_A and F_B for individuals A and B, A is said to Pareto dominate B if and only if:

$$F_A \prec F_B \Leftrightarrow (\forall_m)(F_A(m) \leq F_B(m)) \wedge (\exists_m)((F_A(m) < F_B(m))) \tag{5.6}$$

Equation 5.6 implies that A Pareto dominates B, if and only if A is at least as good as B on all objectives and it is better than B on at least one objective. The Pareto dominance function is defined in Algorithm 17 [67]. Non-dominated individuals form the Pareto front where each individual on the front is optimal with respect to one of the objectives. The Pareto ranking has two advantages. First, it allows the GP to maintain solutions which can optimize different objectives (i.e. in case of a buffer overflow attack, minimizing the anomaly rate and minimizing the delay can be considered as different objectives). Second, it allows GP to maintain some diversity by decreasing the rank of the repeated individuals.

The GP framework discussed in this section employs the Pareto ranking algorithm [57], which was originally proposed by Fonseca et al. [28]. In Pareto ranking,

Algorithm 12: Main GP training loop with Pareto ranking

Input : Number of iterations $NumIterations$, probability of crossover P_{xo}, probability of mutation P_{mut}, probability of swap P_{swap}, number of iterations before the population is considered converged, $TerminateCount$

Output: Population of individuals *population*[ ]

```
1  terminate = 0;
2  PopSize_orig = |population[ ]|;
3  ParetoRank(population[ ]) ;
4  for iter = 1 to NumIterations ∧(terminate < TerminateCount) do
5      parents[1] = SelectParent(population[ ]);
6      parents[2] = SelectParent(population[ ] - {parents[1]});
7      if TestProb(P_xo) then
8          children[ ] = ApplyXO(parents[ ]);
9      end
10     children[ ] = ApplyInstrwiseMutation(children[ ], P_mut);
11     if TestProb(P_swap) then
12         children[ ] = ApplySwap(children[ ]);
13     end
14     population[PopSize_orig + 1] = children[1] ;
15     population[PopSize_orig + 2] = children[2] ;
16     ParetoRank(population[ ]) ;
17     Replace(population[ ]);
18     new_rank_fdf = CalculateRankFDF(population[ ]) ;
19     if new_rank_fdf = rank_fdf then
20         terminate++ ;
21     else
22         terminate = 0 ;
23     end
24     rank_fdf = new_rank_fdf ;
25 end
```

Algorithm 13: SelectParent(...) function

Input : Population of individuals *population*[ ]

Output: An individual selected in inverse proportion to its Pareto rank

```
1  sum = 0;
2  for ind = 1 to |population[ ]| do
3      sum = sum + 1/population[ind].rank ;
4  end
5  r = Random( x ∈ ℝ+ | x ≤ sum) ;
6  sum2 = 0 ;
7  for i = 1 to |population[ ]| do
8      sum2 = sum2 + 1/population[ind].rank ;
9      if r ≤ sum2 then
10         return population[ind] ;
11     end
12 end
```

Algorithm 14, the rank of the individuals is initialized with 1. For each individual, the algorithm determines the number of individuals which the given individual dominates. The rank of an individual is incremented by the number of individuals that it dominates. This implies that the non-dominated individuals will have a rank of 1. In case of a tie between two individuals, the rank of the individual defined by the inner loop in Algorithm 14 is incremented by one. Therefore lower rank values correspond to better performing individuals. Non-dominated individuals with rank 1 form the Pareto-front of the population in which the solutions are non-dominated and optimize at least one of the objectives.

5.4.4 Search Operators

GP with Pareto ranking employs cut and splice crossover, instruction-wise mutation and swap as the search operators. The cut and splice operator in Algorithm 18 selects different crossover points on the parents, therefore the amount of exchange may not be symmetric. Compared with the single instruction mutation operator, the

Algorithm 14: ParetoRank(...) function

Input : Population of individuals *population*[ ]

Output: Ranks of individuals in *population*[ ] are updated

```
1  T = ∅ ;
2  for ind = 1 to |population[ ]| do
3      population[ind].rank = 1 ;
4  end
5  for i = 1 to |population[ ]| do
6      for j = 1 to |population[ ]| do
7          r = ParetoDom(population[i], population[j]);
8          if r = -1 then
9              if i ≠ j ∧ (i, j) ∉ T then
10                 T = T ∪ {(i, j)};
11                 population[j].rank++ ;
12             end
13         else
14             population[i].rank = population[i].rank + r ;
15         end
16     end
17 end
```

Algorithm 15: Replace(...) function

Input : Population of individuals $population[\,]$

Assumptions: $\rightleftharpoons$ operator exists which swaps two genes

Output: Removes the worst ranked two individuals and returns $population[\,]$

```
1  PopSize_orig = |population[ ]| − 2 ;
   // Move the worst ranked two individuals to the end for deletion
2  for i = 2 to 1 do
3      w_r = 1 ;
4      w_j = −1 ;
5      for j = 1 to PopSize_orig + i do
6          if population[j].rank ≥ w_r then
7              w_r = population[j].rank ;
8              w_j = j ;
9          end
10     end
11     population[w_j] ⇌ population[PopSize_orig + i] ;
12 end
13 population[PopSize_orig + 2] = ∅;
14 population[PopSize_orig + 1] = ∅;
```

Algorithm 16: CalculateRankFDF(...) function

Input : Population of individuals $population[\,]$

Output: $fdf[\,]$, which is a frequency function of the ranks

```
1  for i = 1 to |population[ ]| do
2      fdf[i] = 0 ;
3  end
4  for i = 1 to |population[ ]| do
5      ind_rank = population[i].rank ;
6      fdf[ind_rank] = 1/ |population[ ]| ;
7  end
```

Algorithm 17: ParetoDom(...) function

Input : Two individuals, $indA$ and $indB$ with feature vectors $indA.F[\,]$ and $indB.F[\,]$, where smaller values are more optimal

Output: Returns 1 if $indA$ Pareto dominates $indB$, -1 if $indA$ and $indB$ are indifferent and 0 if otherwise

```
1  for i = 1 to |indA.F[ ]| do
2      if indA.F[i] ≤ indB.F[i] then
3          flag = flag + indA.F[i] < indB.F[i] ;
4      else
5          return 0 ;
6      end
7  end
8  if flag > 0 then
9      return 1 ;
10 else
11     return -1 ;
12 end
```

instruction-wise mutation operator in Algorithm 19 introduces more changes into the population. Such a mutation scheme provides a more effective way to introduce new genetic material to the population, which focuses on a problem with a more extensive search space (Section 8.2.4) compared with the problem which the GP framework in the previous work focuses (Section 7.2.4). The same swap operator which was discussed in Section 5.3.4 is utilized. The greedy search operator in Algorithm 20 is a 'hill climbing' operator which aims to improve the best individual by identifying the best single instruction change and is employed in a limited number of experiments [45].

Cut and Splice Crossover

In cut and splice crossover, two different crossover points are selected uniformly from each parent. This splits the parents into two fragments each. The first (second) child combines the first fragment from the first (second) parent and the second fragment from the second (first) parent. Since the amount of exchange may not be equal, the children can have different lengths from their parents. The thesis experiments which involve evolving mimicry attacks against anomaly detectors in Chapter 8 employ the cut and splice crossover operator detailed in Algorithm 18.

Instruction-wise Mutation

The instruction-wise mutation operator tests each instruction independently for the application of the mutation operator. Following a positive test, the instruction is replaced with another instruction which is selected stochastically from the instruction set. Compared with the single instruction mutation operator, it introduces more new genetic material to the population. Consequently, its probability is generally much lower than the probability of a single instruction crossover.

In the case of the instruction-wise mutation operator, a linear annealing schedule is employed such that at the last tournament, the mutation probability is zero, decaying linearly as the tournament count increases. The basic motivation is to enable the crossover operator to investigate different contexts of population material as the tournaments advance.

Algorithm 18: ApplyXO(...) function

Input : A pair of individuals from which the children are created *parents*[ ]

Output: Resulting children *children*[ ]

```
// Note that the individuals are variable length
```

1 $children[\,] = \emptyset$;
2 xo_point1 = `Random(` $x \in \mathbb{Z}^+ \mid x \leq |parents[1].gene[\,]|$`)` ;
3 xo_point2 = `Random(` $x \in \mathbb{Z}^+ \mid x \leq |parents[2].gene[\,]|$`)` ;
4 c1_available = 1;
5 c2_available = 1;
6 **for** $loc = 1$ **to** xo_point1 **do**
7 children[1].gene[c1_available] = parents[1].gene[loc];
8 c1_available++ ;
9 **end**
10 **for** $loc = 1$ **to** xo_point2 **do**
11 children[2].gene[c2_available] = parents[2].gene[loc];
12 c2_available++ ;
13 **end**
14 **for** $loc = xo\_point1 + 1$ **to** $|parents[1].gene[\,]|$ **do**
15 children[2].gene[c2_available] = parents[1].gene[loc];
16 c2_available++ ;
17 **end**
18 **for** $loc = xo\_point2 + 1$ **to** $|parents[2].gene[\,]|$ **do**
19 children[1].gene[c1_available] = parents[2].gene[loc];
20 c1_available++ ;
21 **end**

The instruction-wise mutation operator detailed in Algorithm 19 is employed in evolving mimicry attacks against anomaly detectors in Chapter 8.

Algorithm 19: ApplyInstrwiseMutation(...), instruction-wise mutation function

Input : Set of opcodes $FunctionSet$, set of operands $TerminalSet$, a pair of individuals from which the children are created $parents[\,]$, probability of mutation P_{mut}

Output: Resulting children $children[\,]$

```
1 children[ ] = parents[ ] ;
2 IS = {i = (f, p1, p2) | f ∈ FunctionSet ∧ p1, p2 ∈ TerminalSet} ;
3 for ind = 1 to |children[ ]| do
4     for loc = 1 to |children[ind].gene[ ]| do
5         if TestProb(P_mut) then
6             children[ind].gene[loc] = Random( i ∈ IS) ;
7         end
8     end
9 end
```

Instruction Swap

The instruction swap operator detailed in Section 5.3.4 is also employed in this training configuration.

Greedy Search

The greedy search operator selects the current best performing individual and searches for the best single instruction change, which is the single instruction change which produces the best fitness value. This implies that all possible instructions at each instruction location on the individual are evaluated. Given the computational cost of accessing such an operator, it is only applied every 1000 tournaments, on a limited number of experiments [45].

Algorithm 20: GreedySearch(...), Greedy search function

Input : An individual to be mutated, *individual*, set of opcodes $FunctionSet$ and set of operands $TerminalSet$

Output: Returns the mutated individual, *individual*

```
1  IS = {i = (f, p1, p2) | f ∈ FunctionSet ∧ p1, p2 ∈ TerminalSet} ;
2  temp = individual;
3  max_fit = individual.fitness;
   // Test all gene locations
4  for i = 1 to |individual.gene[ ]| do
       // Test all opcode / operand combinations
5      for j = 1 to |IS| do
6          temp.gene[i] = j ∈ IS ;
7          if temp.fitness > max_fit then
8              max_i = i;
9              max_j = j;
10             max_fit = temp.fitness;
11         end
12     end
13 end
   // Apply the single gene mutation that produces the best fitness
14 individual.gene[max_i] = max_j;
```

Chapter 6

Optimizing Buffer Overflow Characteristics

As discussed in Chapter 2, since a block of NoOP instructions presents a detectable pattern, the first objective of the attacker is to optimize the length of the NoOP sled and the accuracy of the approximated return address. From an attacker's perspective, a shorter NoOP sled is desired but this implies that the approximated return address should be more accurate. The methodology described in this chapter aims to automate the identification of suitable malicious buffer overflow characteristics using Grammatical Evolution. The methodology proposed in this chapter focuses on evading misuse detectors namely, the Snort detector, by altering characteristics such as the NoOP sled length and the desired return addresses. Furthermore, it represents the first step towards improving the payload of the stack overflow attack, namely the shellcode.

6.1 Background and Motivation

In this thesis, the principal objective of evolving buffer overflow attacks is to assess the detectors against buffer overflow vulnerabilities. Recent work on vulnerability testing has indicated that intrusion detection systems can detect a particular instance of an attack, but are unable to 'generalize' to the class of overflow attacks [13] [22] [23] [66] [88] [96] [102] [105]. The main objective of employing GE is to automate the generation of successful malicious buffers. This set of experiments can be considered as part of a wider framework in which the stack buffer overflow attacks are generated automatically for the purpose of discovering the blind spots and weaknesses in intrusion detection systems. Thus, in the wider framework, evolving the shellcode at the assembly and shellcode levels using Genetic Programming is covered in Chapters 7 and 8, respectively.

In this chapter, Grammatical Evolution is employed to discover the NoOP sled

and return address characteristics of the malicious buffer with the objective of identifying a wide range of successful buffer overflows. The population represents a set of candidate exploits with different NoOP and return address characteristics. The approach discussed in this chapter requires the memory address of the stack and feedback regarding the success of the attack, both of which can be obtained by a debugger or an executable code analyser. Evolution is guided by the definition of a suitably informative fitness function which determines the 'quality' of the malicious buffer. Attack diversity is maintained by modifying the fitness function to incorporate fitness sharing, which discounts fitness based upon the degree of similarity between individuals. The fitness function also represents the principal mechanism for incorporating *a priori* knowledge. In this case, minimizing the NoOP sled length is known to improve the chances of avoiding detection. Incorporating this bias into the fitness function and testing the resulting exploits against a misuse detection system, Snort, indicated that the resulting attacks were more effective at avoiding detection.

6.2 Methodology

The methodology aims to evolve programs which can craft buffer overflows to automate vulnerability testing against misuse-based detectors. To do so, Grammatical Evolution is employed to discover the characteristics of a successful buffer overflow. Moreover, fitness sharing is utilized to encourage the evolution of different malicious buffers. For the purposes of this work, a simple and generic vulnerable application was developed, which performs a data copy without checking the internal buffer size. The malicious buffers which GE generated were deployed against the Snort detector to assess the detection rate (Section 6.2.3).

6.2.1 Grammatical Evolution

The details of Grammatical Evolution are discussed in Section 5.2. A simple C grammar was developed for generating programs which assemble the malicious buffer exploits, Figure 6.1. The resulting C program is an individual which approximates the desired return address and assembles the malicious buffer exploit. Each individual of the population represents a buffer overflow attack. The grammar allows an individual

to determine the offset, the size of the NoOP sled, and the number of desired return addresses; hence GE alters these parameters to generate malicious buffers with different characteristics. The malicious buffer contains a NoOP sled, a 46-byte shellcode which spawns a UNIX shell, and back-to-back desired return addresses. The first set of experiments with a basic GE (detailed in Section 6.3) showed that the population converges to one type of solution. In the second and the third sets of experiments in this chapter, niching based upon fitness sharing [19] [68] was used to encourage population diversity, that is, multiple types of attack. Thus, a fit individual can obtain a low 'shared' fitness, if many individuals find a similar solution. Therefore, in this context, fitness sharing is implemented to pressure individuals into utilizing different NoOP sled sizes and return addresses. The generic parameters employed in training of the individuals are determined empirically and summarized in Table 6.1.

```
code : exp
exp : detn detb deto alloc offsetc prel1 loop1 loop2 prel3 loop3 post3
digit : 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
number : digit + digit * 10 + digit * 100 + digit * 1000
detn : nsize = number ;
detb : bsize = nsize + number ;
deto : offset = number ;
alloc : buffer = malloc ( bsize );
offsetc: esp = sp();ret = esp - offset;
prel1 : ptr = buffer; addr_ptr = (long *) ptr;
loop1 : for ( i = 0 ; i < bsize ; i = i + 4 ) { exp1 };
loop2 : for ( i = 0 ; i < nsize ; i = i + 1 ) { exp2 };
prel3 : ptr = buffer + nsize;
loop3 : for ( i = 0 ; i < strlen (shellcode) ; i = i + 1 ) { exp3 };
post3 : buffer[ bsize - 1] = 0;
exp1 : *(addr_ptr++) = ret;
exp2 : buffer[ i ] = '\\x90';
exp3 : *(ptr++) = shellcode[ i ];
```

Figure 6.1: A simple C grammar for generating programs which assemble the malicious buffer

Table 6.1: Grammatical Evolution training parameters

Parameter	**Setting**
Number of individuals	200
Number of generations	500
Probability of mutation	0
Probability of crossover	0.9
Replacement Strategy	Children replace the parents if their fitness is better
Number of niches	5
Training Time	Approximately 7 hours
Number of runs	10

Fitness Function

The fitness function is used to express several characteristics which are related to achieving the overall objective. That is to say, basing the fitness function upon a binary criterion – such as whether the individual gains super-user status successfully or not – would not provide a sufficiently informative function space for the efficient evolution of exploits. In this work, six characteristics of a malicious buffer are utilized.

1. **Existence of the shellcode ($\mu_{shellcode}$):** a binary flag declaring whether the (root shell access) shellcode is inserted into the malicious buffer successfully. Thus, even if the buffer overflow is successful, without the shellcode, the attack cannot succeed.

2. **Success of the attack ($\mu_{success}$):** the reaction of the application, i.e. a binary flag indicating whether the root shell was obtained.

3. **NoOP sled score:** based upon the ratio of NoOP instructions to the overall size of the NoOP sled prior the shellcode. If the execution jumps into the NoOP sled, any non-NoOP instruction in the NoOP sled can have undesirable effects on the succeeding shellcode. The NoOP sled score is formulated as:
$$score_{NoOP} = 1 - \frac{number\ of\ non - NoOP\ instructions}{NoOP\ sled\ length} \tag{6.1}$$

4. **Back-to-back desired return address score:** similar to the NoOP sled score; it is based upon the ratio of correct desired return addresses to the total

number of 4-byte return addresses following the shellcode. If the stack pointer is overwritten with a faulty desired return address, the execution will not jump to the shellcode. The score can be calculated with:

$$score_{retError} = 1 - \frac{number\ of\ faulty\ return\ addresses}{number\ of\ return\ addresses} \tag{6.2}$$

5. **Desired return address accuracy:** this is the difference between the desired return address and the actual address of the variable. A small difference indicates that the approximation is accurate. Accuracy is formulated as:

$$score_{dist} = \frac{1}{|address_{actual} - address_{desired}| + 1} \tag{6.3}$$

6. **Score calculated on the NoOP sled size:** a large block of NoOP instructions can be detected easily by a misuse detector, thus minimizing the size of the NoOP sled is considered to improve the chances of not being detected. This is implemented in the third set of experiments. The score on the NoOP sled length is expressed as:

$$score_{NoOPError} = \frac{1}{1 + NoOP\ sled\ length} \tag{6.4}$$

The last four characteristics are incorporated into the fitness function with their respective weights. In the experiments discussed in this chapter, the weights are all equal and detailed in Table 6.2.

Table 6.2: Weights of the four characteristics of a malicious buffer

Weight	**Value**
Error on NOOP Sled (W_{NE})	20
Error on desired return addresses (W_{RE})	20
Desired return address accuracy (W_{DA})	20
NOOP sled size score (W_{NS})	20

The fitness function provides the basis for directing the search for solutions. In this work, the view is taken that a hierarchy of objectives exists. Thus, if the malicious buffer does not contain the shellcode (i.e. $\mu_{shellcode} = 0$), the individual is assigned a minimum fitness. If the attack is successful (i.e. $\mu_{success} = 1$), the individual is

assigned a fitness between 100 and 120 based upon the size of its NoOP sled. The perfect individual should be successful with a small NoOP sled, or no NoOP sled at all. If the attack is not successful, it is assigned a fitness based upon the error rate of the NoOP sled, desired return addresses and the accuracy of the approximation. The overall fitness function incorporating these properties (with NoOP sled minimization) has the following form:

$$\begin{aligned} fitness = \mu_{shellcode} \times \Big(& \mu_{success} \times (100 + W_{NS} \times score_{NoOP}) \\ & + (1 - \mu_{success}) \times \Big(W_{NE} \times score_{NoOPError} \\ & + W_{RE} \times score_{retError} + W_{DA} \times score_{dist} \Big) \Big) \end{aligned} \tag{6.5}$$

Fitness Sharing

As discussed in Section 5.2.3, fitness sharing [19] [68] from Genetic Algorithms is employed to encourage the population to provide multiple unique solutions. The fitness of an individual is discounted in proportion to its similarity to others in the population. Consequently, the definition of the similarity metric will affect the variations observed in the attributes of the solutions. In this case, such a scheme is introduced to encourage solutions with different NoOP sled lengths and the number of desired return addresses. That is to say, given a successful attack, fitness sharing encourages the identification of additional evasion attacks with different NoOP sled lengths and the number of desired return addresses.

As the population evolves, the boundaries formed by the NoOP sled lengths and the number of desired return addresses will also change. Since determining the boundaries requires a pass of the entire population, σ is calculated every 10 generations in the experiments discussed in this chapter. Fitness sharing is discussed in Section 5.2.3.

6.2.2 The Vulnerable Application

Similar to the example given by Erickson [25], a basic vulnerable application is developed, but this time using four 500-byte arrays. Erickson [25] employed only one 500-byte array whereas in this case, the preliminary experiments with the application

Erickson provided indicated that the NoOP sled is frequently too small to raise any alarms (detailed in Section 6.3). The vulnerable program shown in Figure 6.2 has setuid (set user ID upon execution) bit-enabled and runs with root privileges. It copies the first command line argument to the fourth array without checking the size. This means a successful attack should deploy a malicious buffer which is long enough to overwrite the return address after exceeding 2000 bytes.

```
int main(int argc, char *argv[])
{
   char buffer1[500];
   char buffer2[500];
   char buffer3[500];
   char buffer4[500];
   printf("Vulnerable : Variable at Addr : 0x%x\n", buffer4);
   strcpy(buffer4, argv[1]);
   return 0;
}
```

Figure 6.2: The vulnerable application which was developed for the experiments

6.2.3 The Detector

After the initial experiments, the aforementioned bias toward a minimal NoOP sled was utilized to improve the chances of avoiding detection. To validate the enhancement, the attacks were deployed against a misuse detection system, Snort [87], to observe the detection (or lack of detection) of the malicious buffers. Snort is one of the prevalent lightweight IDSs which attempt to balance (detection) performance, flexibility and simplicity. It represents a widely used open-source intrusion detection system which is able to detect various attacks and probes including instances of buffer overflows, denial of service attacks and stealth port scans [87]. Snort 2.3.2 (build 12) was installed and patched with the latest signatures (March 9, 2005) from the Snort web site [107]. Since the main interest of the experiments in this chapter is in the detection of shellcode attacks, all signatures were disabled except shellcode signatures. There are 21 shellcode signatures, which detect different encodings of NoOP instructions as well as other well-known instructions such as setting the user or group

ID to root. Other than the signature reduction, Snort was employed with its default parameters.

For the IDS to detect an attack, the malicious buffer which the attack in question deploys should be manifested in the event stream which the IDS monitors. Since Snort is a network-based IDS, this means that the shellcode should appear in network traffic. To make the shellcode apparent in the Snort event stream (i.e. the network traffic), the vulnerable application is altered to print the contents of a variable. In this scenario, the attacker connects to the target host via telnet and dispatches the malicious buffer. It is assumed that the attacker has no way of suppressing the variable dump, which triggers the Snort signatures. Given the use of encrypted protocols such as SSH, it is important to note that the shellcode may not always appear in network traffic. However, the objective in employing Snort is not to observe the detection of the shellcode by a network-based IDS *per se*. Instead, the objective here is to determine whether the attacks can evade a misuse detection system (especially since the NoOP sled length is being minimized). Given that Snort is one of the widely used misuse detection systems, it was natural to deploy it for this purpose. After each malicious buffer is deployed, the Snort log files are checked to determine how many alarms were raised. From the attacker's point of view, between two successful attacks, the one which raises fewer alarms is favoured. A similar evaluation methodology was employed to test the detection capabilities of IDSs on service vulnerability attacks in Vigna et al. [102] against Snort and ISS RealSecure.

6.2.4 Discussion of the Search Space Size

The GE grammar detailed in Figure 6.1 determines the boundaries of the NoOP length, the length of the malicious buffer and the offset which is employed to calculate the approximated return addresses. Although each individual is a C program, the outcome of the individuals can be expressed in terms of the above-mentioned three characteristics. This implies that the search space size depends upon the number of values which the NoOP length, the length of the malicious buffer and the offset can take.

Let n be the number of values which the NoOP length can take, m be the number

of values which the malicious buffer length can take and r be the number of values which the offset can take. Therefore, the search space size is determined by the number of variations in characteristics (n, m, r) which describe the malicious buffer. Given that each characteristic is determined independent of the others, the total number of variations in (n, m, r) is calculated as $m \times n \times r$.

The integer definition in grammar in Figure 6.1 allows n, m and r to vary between 0 and 9999, therefore each can have 10000 different values. Consequently, the size of the search space (i.e. the total number of candidate solutions), given the grammar in Figure 6.1 is $10^4 \times 10^4 \times 10^4 = 10^{12}$.

6.3 Results

In the initial set of experiments, fitness sharing was not utilized. The second set of experiments utilizes fitness sharing, whereas the third set of experiments incorporates the bias for encouraging smaller NoOP sleds.

The results are expressed in terms of the fitness of the individuals (attacks) and the number of alerts which Snort generated when they were executed. Moreover, identifying whether a subset of attack properties is more correlated with evolving successful buffer overflow attacks than others is of interest. To do so, three characteristics of a malicious buffer were observed: the NoOP sled size, the number of desired return addresses, and an assessment of the buffer overflow. For the latter, four types of buffer overflow are considered.

- **Invalid Buffer:** the malicious buffer does not contain the shellcode, hence it has zero chance of success.
- **Valid Buffer:** the buffer has the NoOP sled, shellcode and desired return addresses present.
- **Viable:** in addition to being valid, the buffer deploys successfully, obtaining a root shell.
- **Undetectable:** in addition to being viable, Snort raises no alarms during its execution.

Table 6.3 details the assessment of buffer overflows for different experiments. In all three experiments, the C grammar which builds the programs which assemble malicious buffers ensures that the majority of the population is at least valid. Although niching reduces the number of viable buffers, it also encourages diversity in the population, which will be discussed later in this section.

Table 6.3: Malicious buffer types and counts for three experiments

	No Niching	**Niching**	**Niching and NoOP min**
Invalid	2	6	2
Valid	0	118	111
Viable	146	54	57
Undetectable	52	22	30

Figures 6.3, 6.4 and 6.5 summarize the population from the three experiments with the NoOP sled size and the number of desired return addresses plotted with fitness. As mentioned above, the vulnerable variable is approximately 2000 bytes away from the return address (i.e. the EIP value stored on the stack). In Figure 6.3, experiments with the basic GE provides attacks which resulted in a range of return addresses (between 5000 and 10000). The introduction of the sharing function in Figure 6.4 increased the diversity of all three parameters: fitness, NoOP size, and return address. This indicates that the attacks learn to overwrite the EIP with an approximated return address. In the third set of experiments, Figure 6.5, two attacks stand out from the rest of the population with a fitness value of 110. These attacks can deploy successfully while using a single NoOP instruction, making them very difficult to detect using signatures targeting the NoOP code. Moreover, this is achieved without compromising the success of the attack itself.

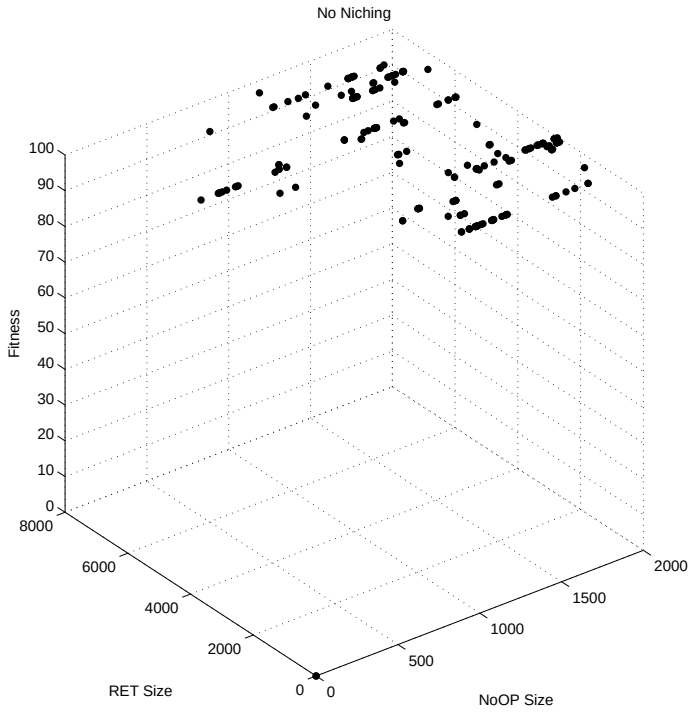

Figure 6.3: Fitness, NoOP sled size and the desired return address size of the population in the last generation in the experiments without niching

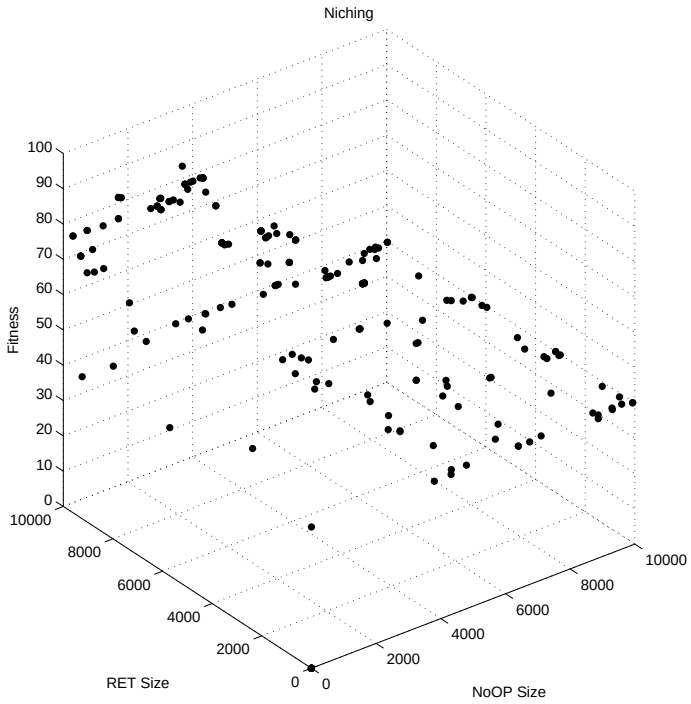

Figure 6.4: Fitness, NoOP sled size and the desired return address size of the population in the last generation in the experiments with niching

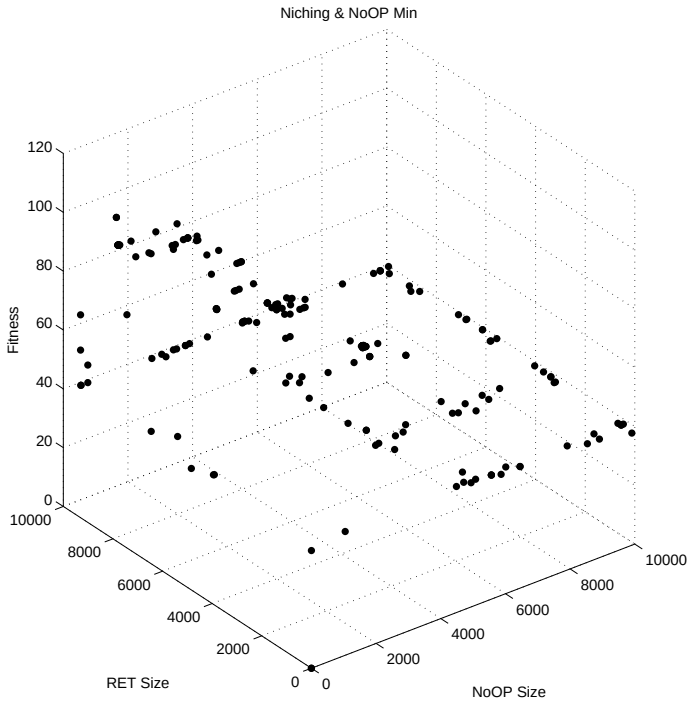

Figure 6.5: Fitness, NoOP sled size and the desired return address size of the population in the last generation in the experiments with niching and NoOP minimization

Figures 6.6, 6.7 and 6.8 show the NoOP sled size and the accuracy of the desired return address for the three experiments. In all three experiments, the NoOP sled size has a linear relation with the accuracy of the desired return address. In other words, as the accuracy improves (i.e. distance gets smaller), the NoOP sled size becomes smaller. Moreover in the case of successful attacks (i.e. attacks with fitness values of 100 or above), it was observed that the NoOP sled size was always kept below 2000 bytes.

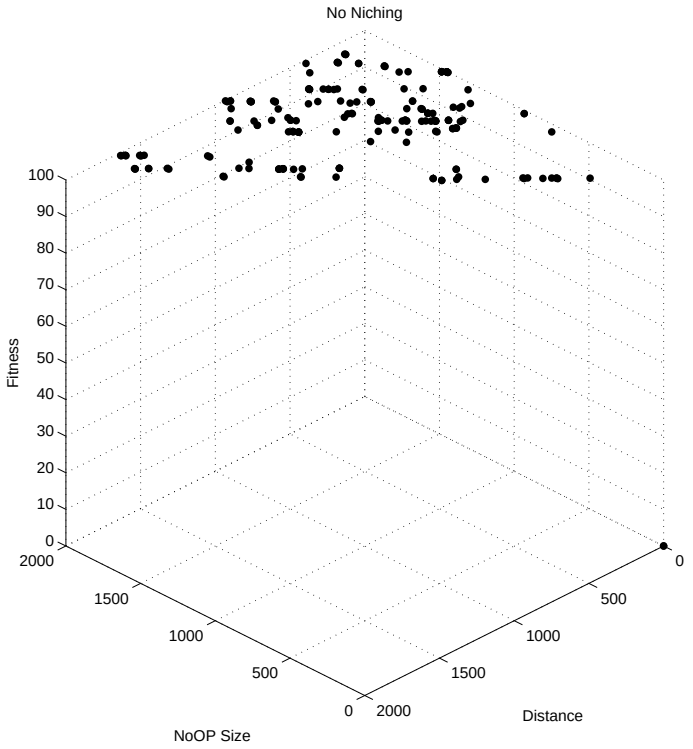

Figure 6.6: Fitness, NoOP sled size and the accuracy of the desired return address of the population in the last generation in the experiments without niching

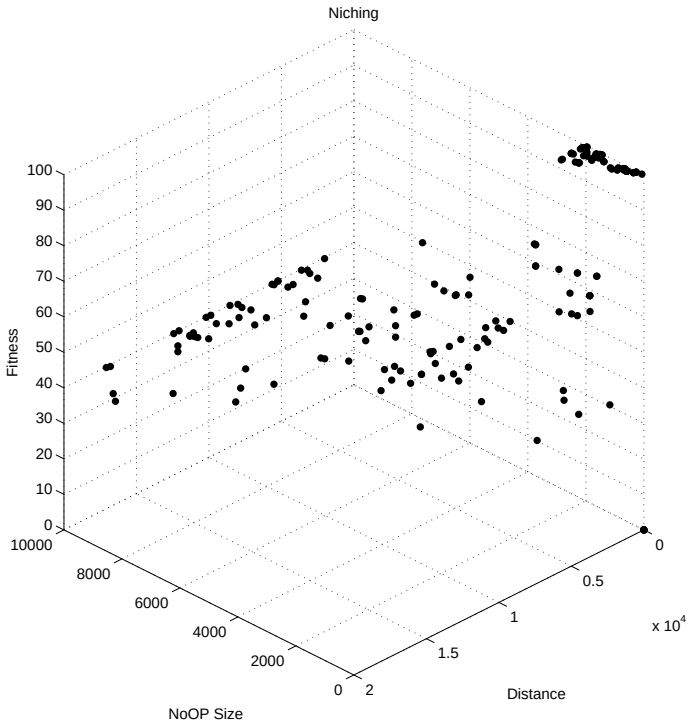

Figure 6.7: Fitness, NoOP sled size and the accuracy of the desired return address of the population in the last generation in the experiments with niching

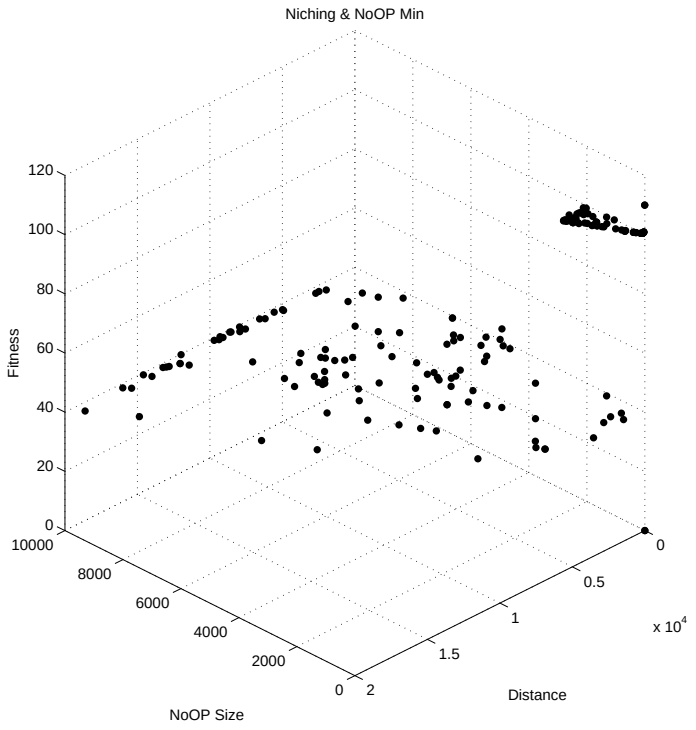

Figure 6.8: Fitness, NoOP sled size and the accuracy of the desired return address of the population in the last generation in the experiments with niching and NoOP minimization

Figure 6.9 details the mean fitness of the population over 500 generations. In all three experiments, populations converged to a solution after approximately 100 generations. In the niching experiments, the mean fitness of the population is lower than the mean fitness of the experiments without niching because attacks which generate valid buffers while maintaining diversity have a shared fitness comparable to the shared fitness of the attacks which generate viable buffers with similar parameters.

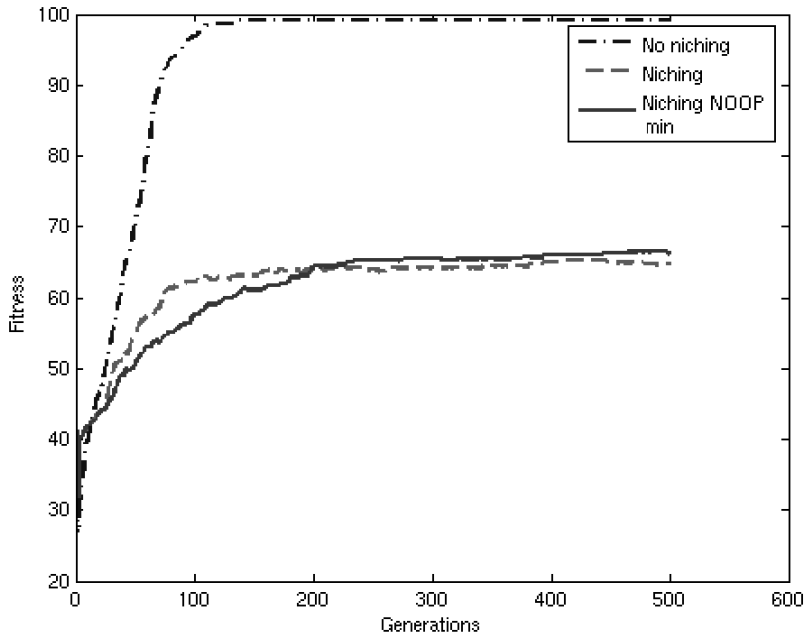

Figure 6.9: Mean raw fitness of the population over 500 generations

Figure 6.10 shows the change of the NoOP sled length over generations. Figure 6.10 demonstrates that the population converges after 100 generations. In the experiments without NoOP minimization, after a few hundred generations, the mean NoOP sled length stops decreasing; whereas in the NoOP minimization experiments the fitness function continues to minimize NoOP sled length even if the buffer overflow deploys successfully.

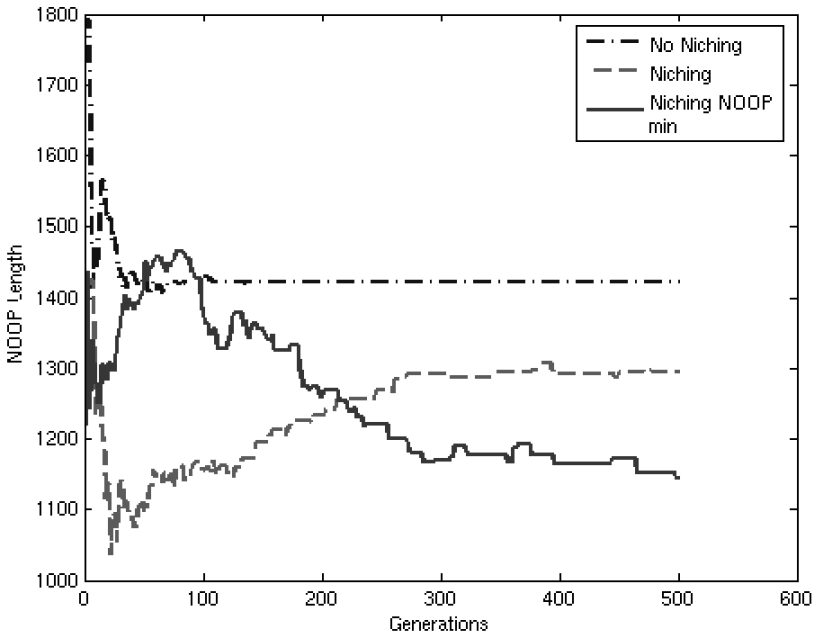

Figure 6.10: Mean NoOP size for viable and undetected attacks over 500 generations

In Figures 6.11, 6.12 and 6.13, buffer overflows are plotted with NoOP sled sizes, the number of alerts and fitness. Since the population without niching in Figure 6.11 converged with less diversity, the number of alerts was 0, 1 or 2. In the case of experiments which employed niching (Figures 6.12 and 6.13), the alert count ranged between 0 and 10 (greater diversity in NoOP sled length). Signature analysis showed that the Snort NoOP signature (Figure 6.14), which monitors the existence of large blocks of `0x90`, triggered all alerts.

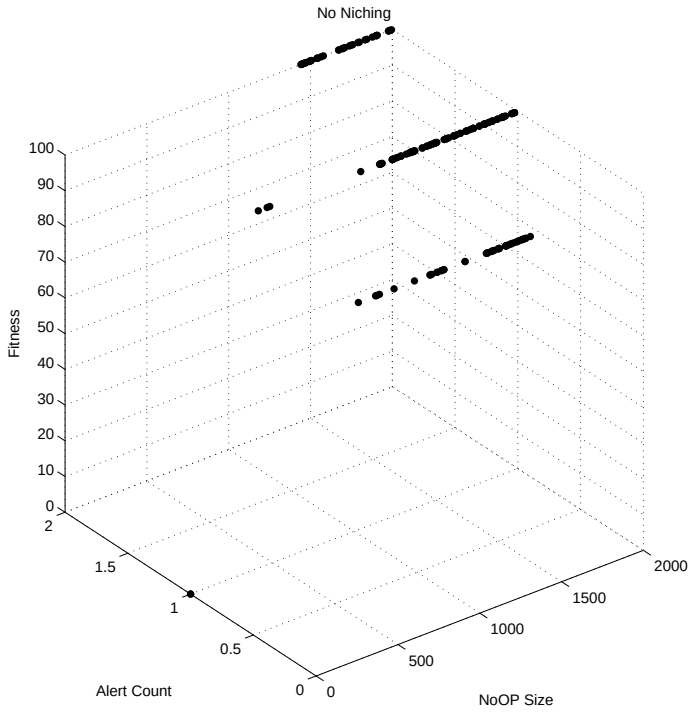

Figure 6.11: Fitness, NoOP size and alert counts of the population in the last generation in the experiments without niching

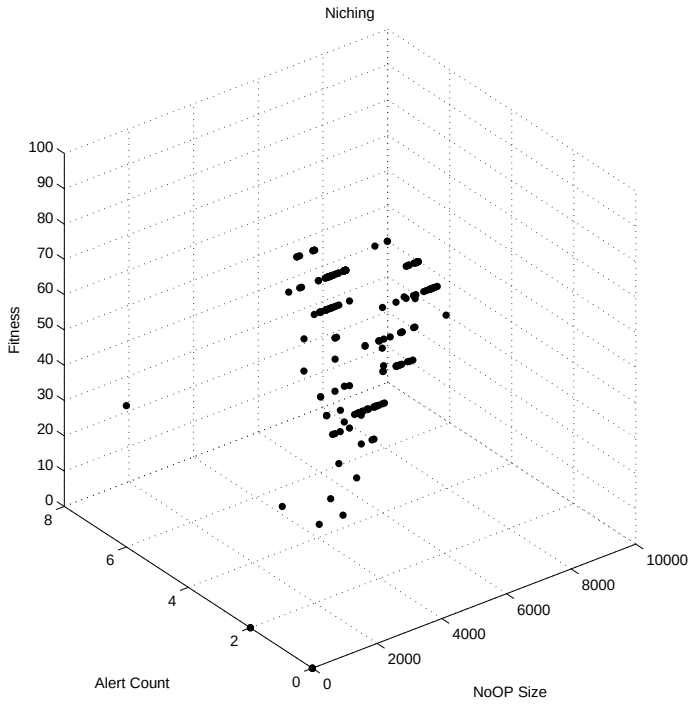

Figure 6.12: Fitness, NoOP size and alert counts of the population in the last generation in the experiments with niching

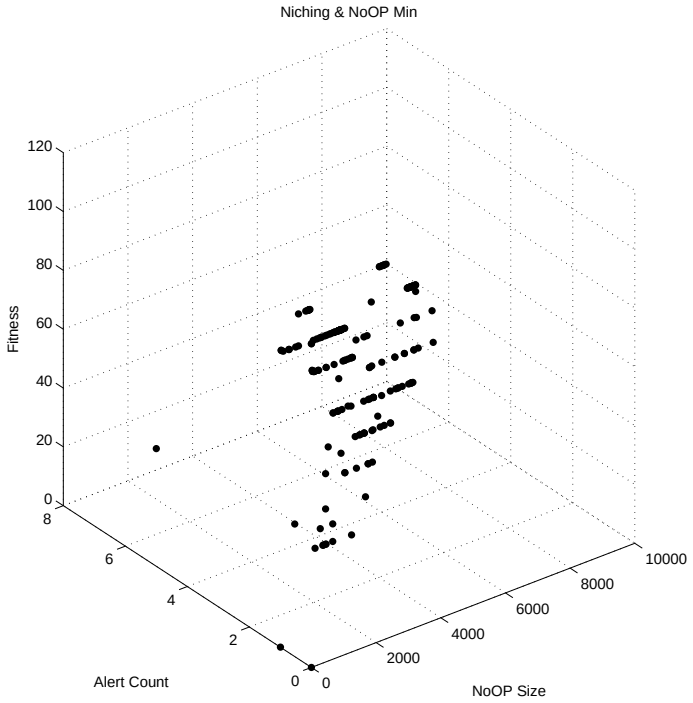

Figure 6.13: Fitness, NoOP size and alert counts of the population in the last generation in the experiments with niching and NoOP minimization

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
(msg:"SHELLCODE x86 NOOP"; content:"|90 90 90 90 90 90 90
90 90 90 90 90 90 90|"; depth:128; reference:arachnids, 181;
classtype:shellcode-detect; sid:648; rev:7;)
```

Figure 6.14: The Snort signature for detecting x86 NoOP sleds

Figure 6.15 details the average alert count for viable attacks. Table 6.3 shows that basic GE managed to produce the most undetectable attacks. However, it is also apparent that in terms of the average alert count of the population, niching with NoOP minimization produces the least number of alerts. Moreover niching with NoOP minimization results in two attacks with only one NoOP instruction each, effectively undetectable.

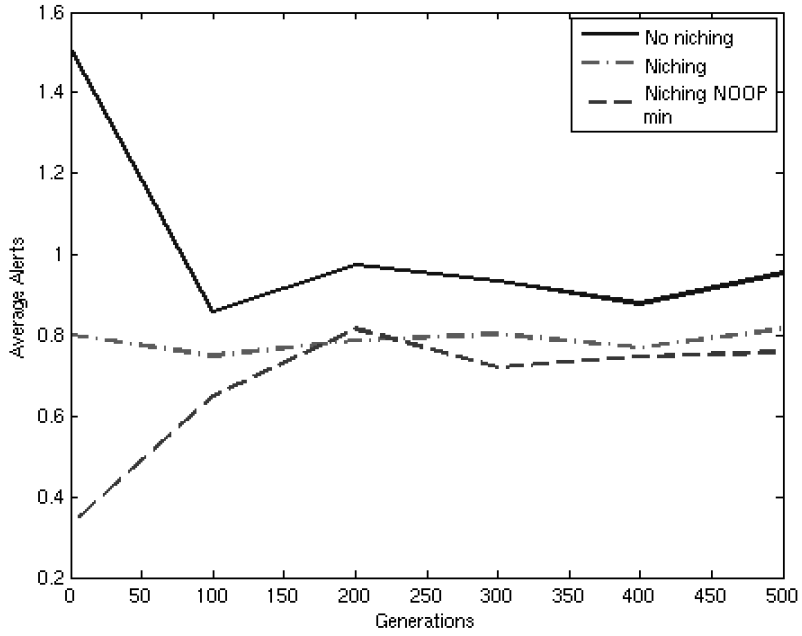

Figure 6.15: Average alert count for viable and undetectable attacks for the three sets of experiments

6.4 Discussion of Results

Grammatical Evolution was employed to evolve C programs to perform three tasks:

1. approximating the address of the vulnerable variable;
2. determining the length of the NoOP sled and the number of desired return addresses;
3. assembling the malicious buffer in the light of the characteristics established by the first two tasks (as indicated before, between two attacks which deploy successfully, the one which raises fewer alarms is preferred).

Hence, every 100 generations, the population was tested against Snort to determine the detection (or lack of detection) of the attacks. Since the only signature triggered during the deployment of attacks was the NoOP detection signature, Figure 6.14, results indicate that Snort employs signatures based solely upon the NoOP sled size to detect the buffer overflow attacks generated in the three experiments.

Three sets of experiments were performed, namely, (i) basic GE, (ii) GE with niching, which encourages a population to maintain diversity, and (iii) GE with niching and NoOP minimization since longer NoOP sleds are detected easily. Although all three experiments produced comparable results, basic GE produces the best mean fitness and the most viable attacks. On the other hand niching produces programs which can craft a malicious buffer with different NoOP sled sizes and the number of desired return addresses. Furthermore, NoOP minimization produces smaller mean NoOP sled lengths and fewer alerts per population, which is desirable from an attacker's point of view. Results also show that in order for an attack to be successful, the return address (EIP) should be overwritten with an accurate desired return address which directs the execution to a point in the NoOP sled or the first instruction of the shellcode. NoOP sled length decreases as the accuracy of the desired return address increases.

In summary, the results indicate that GE can automate the optimization of NoOP sled length and return address accuracy. As well, GE can bypass the Snort signature which monitors the existence of long sequences of NoOP instructions.

Chapter 7

Evolving Exploits at Assembly Level

Chapter 6 investigated the optimization of the NoOP and return address components of buffer overflow attacks in order to evade misuse detectors such as Snort. Another important aspect of buffer overflow attacks is the payload (i.e. the shellcode) design, where the attacker aims to evade detection by modifying the payload so that, while the payload performs the attacker's goals, it does not trigger any signatures (of a misuse detector) nor deviate from normal behaviour (as defined in the database of the anomaly detector). The methodology proposed in this chapter investigates the design of the payload at the assembly language level using Linear Genetic Programming to evade misuse detectors. The principal objective is to obfuscate the attack in such a way that the misuse detectors (namely, Snort) cannot identify the true intent of the code. Altering the payload at the system call level in order to evade anomaly detectors is discussed in Chapter 8.

7.1 Background and Motivation

As discussed in Chapters 2 and 6, misuse detectors usually contain signatures which describe the distinguishing characteristics of the shellcode such as the existence of `0x90` bytes, and system call parameters such as `'/bin/sh'`. Although the encryption of the shellcode [22] renders it undetectable, Payer et al. [79] established that it is possible to identify the characteristics of the decryption engine which need to accompany the encrypted shellcode.

The methodology discussed in this chapter focuses on evolving the payload at the assembly language level using Linear Genetic Programming. Employing a Linear Genetic Programming-based methodology has the following advantages:

- The code bloat property of Genetic Programming provides a mechanism for

hiding the true intent of the code. Furthermore, GP solutions have been observed to have functionality distributed across the length of the individual. Given that misuse detection is sensitive to changes, padding attack code with non-operational but syntactically correct instructions and changing the ordering of attack instructions helps to hide the payload from misuse detectors which investigate only the sequence of bytes.

- Given the fitness function which describes the goals of the core attack, GP can discover different ways to attain the goals, hence mimicking the core attack. This makes it difficult for the misuse detector to detect the variant attack.

Previous work [102] has demonstrated the use of random modifications to a 'core' attack with the objective of highlighting weaknesses in standard signature-based detectors. Updates to a detector based upon an immune system using a Genetic Algorithm (GA) as the 'attacker' have also been proposed [23]. In this case, however, no attack as such is built (the problem simplifies to function optimization). A GA has been proposed as well for acting as an attack agent in an artificial 'server-user-hacker' environment. In this case the GA is used to construct hacker behaviour from a predefined set of attack scripts [9]. As a consequence, there is no attempt to obfuscate the true intent of the intended behaviour, or to discover alternative methodologies for achieving the same objective. Finally, previous work [48] and Chapter 6 have demonstrated that Evolutionary Computation (GE in particular) can be used to establish the composition of buffer overflow attacks comprising a predefined exploit, NoOP sled and return address. The result has been a set of attacks capable of defeating Snort, a widely used signature-based intrusion detection system.

7.2 Methodology

The principal objective of the methodology is to evolve the shellcode at the assembly language level. To do so, Linear Genetic Programming is employed to generate shellcodes, which spawn a UNIX shell. A fitness function defines the goals for deploying an 'execve' system call. In order to prevent the evolved attacks from crashing the host where the training takes place, a runtime environment is developed which allows

the safe execution of the shellcode and evaluates the fitness of the attack without altering the state of the actual system. The resulting attacks were deployed against the Snort intrusion detection system to evaluate their detection rate.

7.2.1 Fitness Function

Categorically, the attack which is evolved is an 'execve' attack. Execve is a system call which executes a program where the program takes the form of an argument (UNIX shell `'/bin/sh'` in this case). In standard C format, UNIX defines 'execve' as, `int execve(const char *path, char *const argv[], char *const envp[])`.

The first parameter of the execve is the command name (i.e. the program to be executed); the second parameter contains pointers to strings which will be given to the program as arguments; the third parameter contains pointers to environmental variables which are also stored as strings. A minimalist call to the 'execve' function using a C program, which spawns a UNIX shell, is shown in Figure 7.1.

```
int main()
{
   char *command = "/bin/sh";
   char *args[2];
   args[0] = command;
   args[1] = 0;
   execve(command, args, 0);
}
```

Figure 7.1: Calling the execve system call from a C program

The minimal requirements for executing an 'execve' system call for spawning a UNIX shell from a shellcode is detailed as follows:

1. register EAX contains `0x0B` i.e., the system call number for 'execve';
2. register EBX points to `'/bin/sh0'` on the stack;
3. register ECX points to the argument array in the stack;
4. register EDX contains NULL;
5. interrupt `0x80` is executed;

In order to spawn a UNIX shell prompt, 'execve' requires that the command pointer should be in the EBX register, the pointer to 'args' should be in the ECX register and the pointer to the third argument (which, in this case, is the NULL pointer) should be in the EDX register. Moreover, the program name `'/bin/sh'` should be pushed to stack. To achieve these goals, 11 assembly instructions are needed. After 10 instructions are executed (the 11^{th} is the interrupt which transfers control to the execve system call), registers EAX, EBX, ECX and EDX should be configured correctly and the stack should contain the program name to be executed (i.e. `'/bin/sh'`). Given the state of the stack and registers after the 10^{th} instruction, if the values are not set correctly, greedy replacement is used to determine how many instructions are needed to correct it. For example, if `'/bin/sh'` has not been pushed to the stack, 3 instructions are sufficient to achieve this goal. Another important point is that if the task has been half-accomplished, viable instructions should be determined. Using this principle, the fitness function summarized in Figure 7.2 returns a maximum fitness of 10 if all conditions are satisfied, otherwise it subtracts the number of instructions needed to correct the program, relative to the minimal set of sub-goals. The basic fitness function therefore takes the form of a hierarchical fitness function in which sub-goals (b) to (e) need to be completed before the interrupt. However, depending upon the composition of the language used to evolve the attacks, there are multiple programs producing the required (buffer overflow) attack behaviour.

7.2.2 Runtime Environment and Fitness Evaluation

In order to obtain the behavioural fitness requirements defined in Figure 7.2, individuals representing an exploit are executed. Although it is possible to run the attack on a 'real' environment, this approach suffers from the disadvantage that an attack can potentially crash the environment, terminating the training process as a whole. Therefore, execution of the program is accomplished using a virtual runtime environment, which simulates the execution of assembly programs on the 32-bit Intel Architecture [39].

(a) Fitness = 10;

(b) IF stack does not contain '/bin/sh0', THEN subtract number of instructions necessary to do so from Fitness (1 to 3);

(c) IF register EBX does not point to string from (b), THEN Fitness − = 1;

(d) IF register ECX does not point to argument array in stack, THEN subtract number of instructions necessary to do so from Fitness (1 to 3);

(e) IF register EDX ! = NULL, THEN Fitness − = 1;

(f) IF an INT is not executed, THEN Fitness − = 1;

Figure 7.2: Basic fitness function for establishing correct behaviour for the 'execve' exploit

Utilizing symbolic execution for the analysis of malicious behaviour is quite established in the literature. For example, Crandall et al. [16] employed symbolic execution to analyse and detect worm behaviour at the assembly language level. Similarly, Wagner et al. [104] formulated the detection of buffer overflows as an integer range analysis problem to identify overflows before the code is executed. Yang et al. [113] utilized the symbolic execution of various file system functions to analyse malicious disk images. More similar to the approach taken in this chapter, Christodorescu et al. [13] employed a symbolic execution of computer viruses and worms at assembly level to detect the obfuscated variants of the original malicious code. Similar to the characterization of success in Figure 7.2, they defined a "malicious code automaton" which describes the objectives of the original virus.

Although limited in functionality, the runtime environment employed in this chapter is developed with sufficient functionality to execute an execve system call properly, Figure 7.3. Limitations take the form of explicitly prohibiting accesses to, or modification of memory or the heap. The runtime environment, Figure 7.3, contains simulated data structures such as:

1. General-purpose IA32 registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP) which can be used in 32 or 16-bit modes. 8-bit mode is available on EAX, EBX, ECX, and EDX;

2. A special purpose register for testing flag values;

3. An addressable stack.

Registers and the stack are also simulated, which is to say that the execution of a program does not modify the actual stack and registers of the host machine. The execution of each instruction is defined as a C function which modifies the simulated data structures of the virtual machine. Instructions are implemented using the IA32 instruction definitions from the IA32 Developer's Manuals [39]. Special attention was given to flag modification in order to determine which execution branch to take (e.g. the case of a control transfer instruction such as a JMP). When the interrupt instruction is called, a snapshot of the virtual machine state is taken, from which the fitness is calculated.

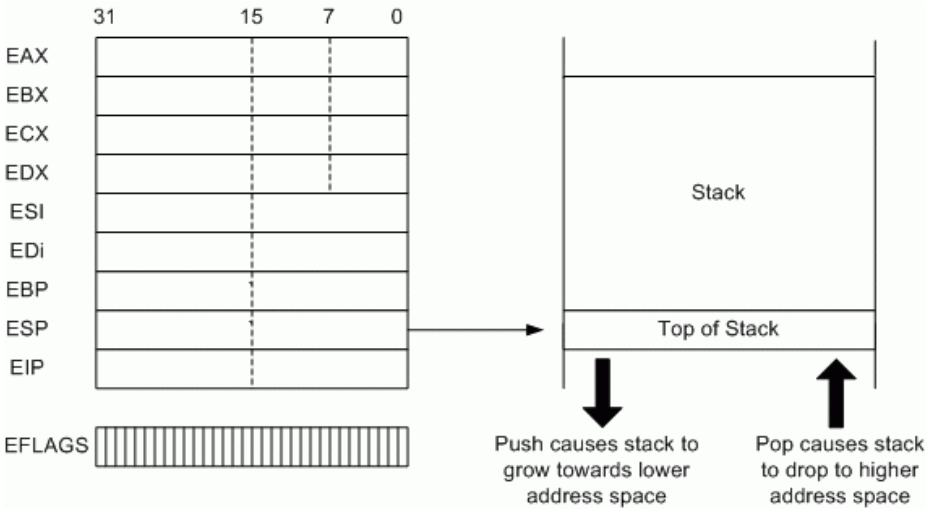

Figure 7.3: Virtual runtime environment for fitness evaluation

7.2.3 Linear GP

Individuals are represented using linear GP in which instructions are composed from a 2-byte opcode and two operands (each 1-byte). In other words, all instructions have the same number of bytes. Table 7.1 defines the instruction set architecture and Table 7.2 defines the parameter types. IMM32 denotes the 32 bit immediate values (immediate is the term used for constants). REG32 denotes the 32 bit general purpose registers, EREG32 denotes the extended 32 bit registers and LREG8 denotes the low 8 bit registers.

The training parameters, which are determined empirically, are detailed in Table 7.3. Individuals are defined using a fixed length format, thus all the individuals are initialized with the same number of instructions. Selection takes the form of a steady state tournament over 4 individuals. The children from the best performing half of the tournament overwrite the individuals corresponding to the worst half of the tournament, taking their place in the population. Search operators take three forms: page-based crossover, instruction mutation, and instruction swap, Table 7.3. Therefore, the crossover operator is constrained to exchanging an equal number of instructions (a page) between two individuals. The mutation operator selects a single instruction with uniform probability and replaces it with a different instruction from the instruction set, Table 7.1. The swap operator selects two instructions from the same individual with equal probability and interchanges their respective positions.

Table 7.1: Linear GP instruction set

Instruction	**Instruction Type**	**Parameter 1**	**Parameter 2**
INT	Control Transfer	0x80	N/A
CDQ	Data Transfer	N/A	N/A
PUSH		IMM32	N/A
PUSH		REG32	N/A
MOV		EREG32	EREG32
MOV		LREG8	0x0B
ADD	Binary Arithmetic	REG32	REG32
SUB		REG32	REG32
INC		REG32	N/A
DEC		REG32	N/A
MUL		REG32	N/A
DIV		REG32	N/A
AND	Logic	REG32	REG32
OR		REG32	REG32
XOR		REG32	REG32
NOT		REG32	N/A

Table 7.2: Parameter types

Parameter Type	**Options**
IMM32	0x68732f2f (‘hs//’), 0x6e69622f (‘nib/’)
REG32	EAX, EBX, ECX, EDX
EREG32	REG32 + ESP
LREG8	AL, BL, CL, DL

Table 7.3: GP parameters

Parameter	**Setting**
Crossover	Page-based crossover with 0.9 probability
Mutation	Uniform instruction-wide mutation with 0.5 probability
Swap	Instruction swap within an individual with 0.5 probability
Selection	Tournament of 4 individuals
Stop Criteria	At the end of 50,000 tournaments.
Population	500 individuals with 10 pages and 3 instructions per page.
Replacement	Children replace the worst half of the tournament.
Training Time	Approximately 6 hours.
Number of runs	20

7.2.4 Discussion of the Search Space Size

In order to determine the search space size of the instruction set given in Table 7.1, a number of variations for each instruction should be determined. For example, PUSH REG32 can be mapped to four different instructions (namely, PUSH EAX, PUSH EBX, PUSH ECX and PUSH EDX) whereas PUSH IMM32 can be mapped to two (namely, PUSH 0x68732f2f and PUSH 0x68732f2f). Given an instruction with two parameters where each parameter can take 5 different values (e.g. the MOV instruction), there exist 25 different variations for the instruction. An enumeration of all possible instruction mappings in Table 7.4 reveals that the instruction set given in Table 7.1 allows for 137 different variations.

Table 7.4: Number of instructions which the instruction set in Table 7.1 allows

Instruction Types	**Parameter 1 Count**	**Parameter 2 Count**	**Variation Count**
INT 0x80	1		1
CDQ			1
PUSH IMM32	2		2
PUSH REG32	4		4
MOV EREG32, EREG32	5	5	25
MOV LREG, 0x0B	4	1	4
ADD REG32, REG32	4	4	16
SUB REG32, REG32	4	4	16
INC REG32	4		4
DEC REG32	4		4
MUL REG32	4		4
DIV REG32	4		4
AND REG32, REG32	4	4	16
OR REG32, REG32	4	4	16
XOR REG32, REG32	4	4	16
NOT REG32	4		4
Total number of instruction variations allowed			**137**

Let n be a number of values that an instruction can take and l be the length of the program. Given that each l instructions can take n different values, the number of candidate programs, which defines the search space, can be calculated with n^l. Since

the length of the individuals is fixed at 30, and each instruction can take 137 different values, the search space size (i.e. the total number of candidate solutions) is 137^{30}, which is approximately 10^{64}.

7.3 Results

In the preliminary experiments, GE was utilized to evolve attacks at the assembly language level. However, GE proved very inefficient at manipulating register references using the standard GE search operators. Therefore, the purpose of utilizing linearly-structured GP is to avoid this problem. In the following, three sets of experiments are described in which the instruction set is incrementally expanded, thus increasing the search space, but providing for greater freedom in the resulting program content (thus a wider range of behavioural properties). This case results in code which has the capacity to intermix attack and obfuscation.

In all cases, the fitness function takes the form of Figure 7.2, augmented with an additional term to measure the likelihood of an attack being executed. Specifically, since all individuals have a fixed length of 30 instructions and it takes 11 instructions to describe the attack, there are up to 19 instructions denoting introns with respect to the malicious code. Within this context, introns, which are discussed under code bloat in Chapter 5, are effective NoOP instructions which do not contribute directly to achieving the goals of the attack. If the approximated return address was not accurate enough to jump to the first instruction, jumping to an effective NoOP (intron) region would allow an attack to deploy successfully. Therefore, from an attacker's perspective, it is more advantageous to place the attack instructions toward the end of the shellcode. Due to inaccurate approximation of the return address, if execution of a successful attack fails by jumping past a relevant instruction, the location of the instruction is called the failure point. The probability of execution is defined as the failure point divided by the number of all possible points; or a denominator of 19 in this case. An attack with the failure point closer to the end is more desirable.

7.3.1 Minimal Instruction Set

The first experiment employs an instruction set composed from the minimum subset of instructions necessary to build the malicious exploit alone (the first 6 instructions of Table 7.1 plus the XOR instruction). This represents a minimal search space in which the relevant instruction sequence, instruction arguments, and intron behaviour are all expressed in terms of the 6 instructions known to describe the minimalist exploit.

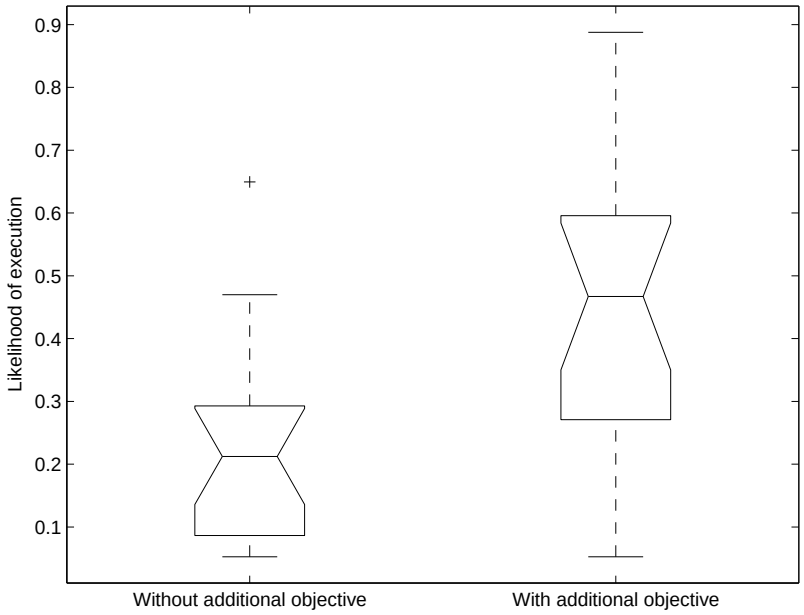

Figure 7.4: Likelihood box plot of executing an attack with and without the additional fitness objective

Figures 7.4 and 7.5 summarize the probability of executing a working exploit and a number of unique individuals under the basic fitness function (baseline) and basic fitness function with the additional objective of maximizing the probability of executing a valid exploit. The vertical lines of the box plots in Figures 7.4 and 7.5 show the third quartile, median and first quartile. Whiskers extend from each end of the box to the adjacent values in the data which are within 1.5 times the inter-quartile range from each end of the box. Outliers, which are data points with values beyond the ends of the whiskers, are displayed with the plus sign.

From Figure 7.4, it is apparent that including the additional objective doubles

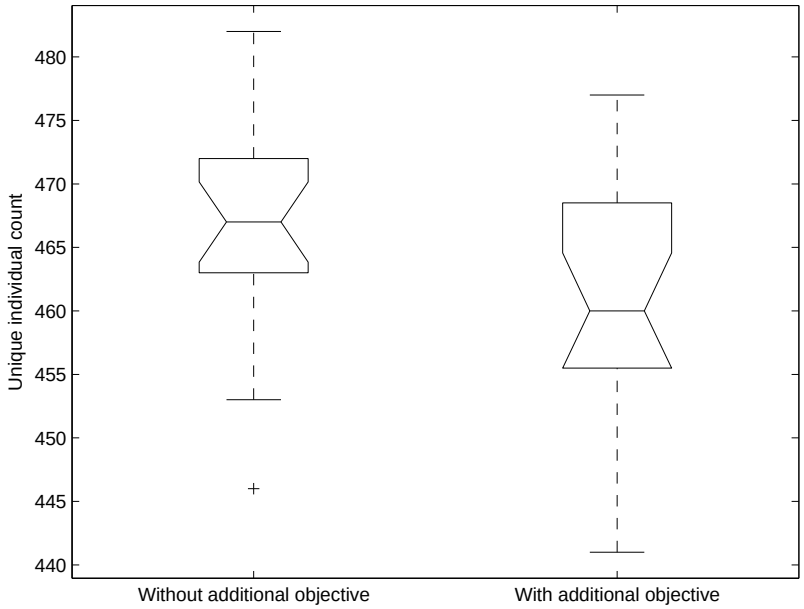

Figure 7.5: Population diversity box plot with and without the additional fitness objective

the likelihood of executing the exploit. A unique individual differs from all others in the population by at least one or more instructions. Figure 7.5 indicates that the population diversity is maintained throughout the generations, without incorporating techniques such as fitness sharing (Section 5.2.3) to encourage diversity.

7.3.2 Extended Instruction Sets

Two additional experiments were conducted, each augmenting the base instruction set as follows:

- the basic 6 instructions plus 6 arithmetic instructions and XOR, Table 7.1;
- the basic 6 instructions, plus 6 arithmetic, plus four logical instructions, Table 7.1.

The results were detailed in terms of the mean fitness, the number of hits (i.e. the number of programs, which has fitness above or equal to 10), and the mean probability of execution for the basic instruction set and the two increments, Figures 7.6, 7.7 and 7.8, respectively. All three figures indicate an improved characteristic

when arithmetic instructions are included in the instruction set. Thus, arithmetic instructions were either helpful in attaining the objectives in different ways or they were good introns. Although the introduction of logic instructions impairs the results compared with the inclusion of arithmetic instructions, the results are still better than the basic instruction set. Hence, extending the search space by adding new instructions does not have a substantially negative impact on deploying successful attacks.

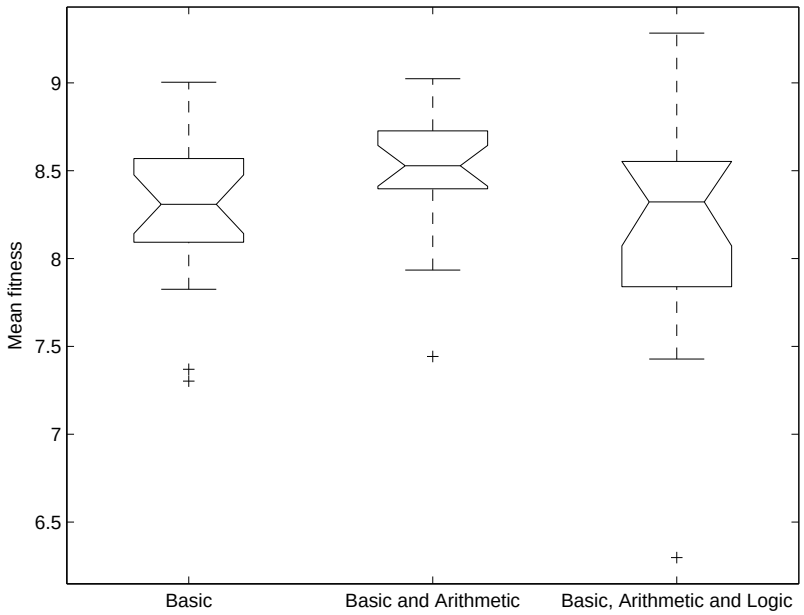

Figure 7.6: Box plot of mean fitness averaged over 20 runs

Table 7.5 provides a comparison between an evolved attack and the core attack from which the fitness function was developed. Exploit code is shown in bold whereas the remaining instructions of the program act like introns. It is apparent that the evolved attack discovered different ways to attain sub-goals (b), (d) and (e) in Figure 7.2. The evolved attack executes successfully and spawns a UNIX shell. Furthermore, the introns, which can also be considered as the obfuscation code segments, are distributed over the entire exploit.

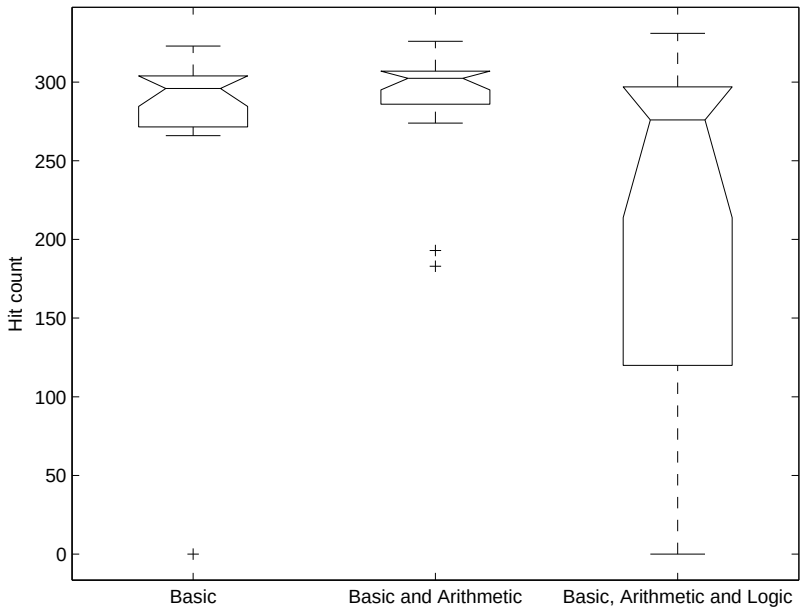

Figure 7.7: Box plot of hit count averaged over 20 runs

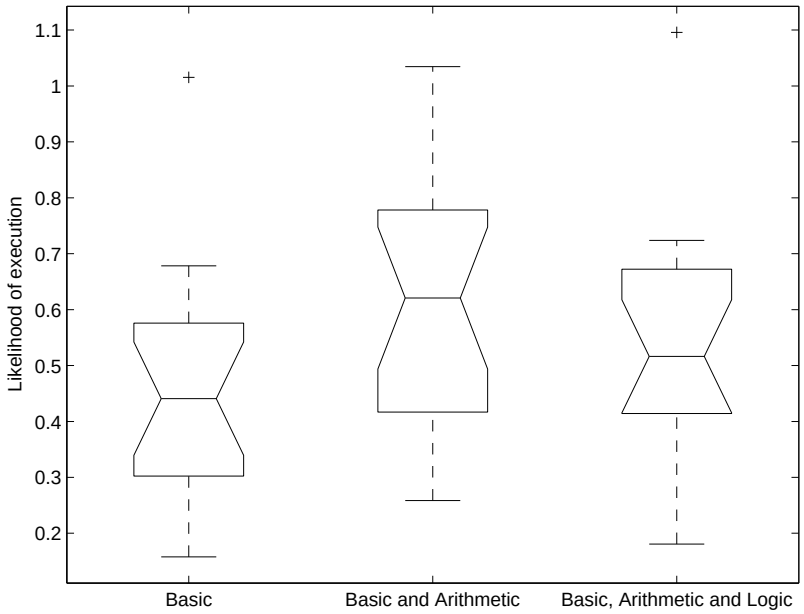

Figure 7.8: Box plot of mean likelihood of exploit execution averaged over 20 runs

Table 7.5: Evolved attack compared with the core attack from which the fitness function is developed

Evolved Program	**Core Attack**	**Sub-goals Attained**
PUSH 0x68732f2f		
MUL EAX		
PUSH EBX		
MUL EDX		
CDQ		
CDQ		
SUB EAX, EAX	XOR EAX, EAX	(e)
MUL EDX	CDQ	(e)
PUSH EDX		
MOV CL, 0x0b		
PUSH EDX		
DEC ECX		
DEC ECX		
MOV EBX, ESP		
PUSH 0x6e69622f		
PUSH EDX	PUSH EAX	(b)
PUSH 0x68732f2f	Same	(b)
PUSH 0x6e69622f	Same	(b)
MOV EBX, ESP	Same	(c)
MOV ECX, EDX	PUSH EAX (step 1)	(d)
CDQ		
MUL EDX		
PUSH ECX	PUSH EAX (step 2)	(d)
PUSH EBX	Same	(d)
MOV ECX, ESP	Same	(d)
MOV AL, 0x0b	Same	(f)
INT 0x80	Same	(f)
PUSH EDX		
PUSH 0x6e69622f		
MOV DL, 0x0b		

7.4 Discussion of Results

Linear GP was employed as a 'white-hat' attacker with the objective of altering the core attack to make it undetectable by signature-based intrusion detection systems. Results show that the code bloat property of GP provides a suitable means for hiding the actual attack by mixing exploit instructions with introns which have no effect toward achieving the goals of the attack. Furthermore, evolved attacks discover different ways of attaining sub-goals associated with building buffer overflow attacks, hence mimicking the core attack with different instructions. Consequently, it becomes harder for a signature-based detector to detect the resulting attack variant. The experiments detailed in this chapter focused on formulating a suitable fitness function and defining instruction sets. Results showed that employing an additional 'likelihood' objective increased the chances of deploying successful attacks by moving the point of failure toward the end of the shellcode. Expanding the instruction set provided additional intron behaviour and supported different avenues for achieving the sub-goals associated with the core attack. In order to observe the detection of the evolved attacks, successful attacks were tested against Snort, which is a widely-used network-based intrusion detection system. Successful attacks were transmitted over a network, where Snort was deployed, monitoring the network traffic. All of the 2110 successful attacks avoided detection by Snort.

In summary, results indicate that linear GP is a suitable method for evolving payload at the assembly level. The resulting evasion attacks achieved the goals by finding alternative assembly instructions which produced the same outcome. Furthermore, the obfuscation of the shellcode made the attacks difficult to detect by a misuse detector.

Chapter 8

Evolving Exploits at System Call Level

Misuse detectors such as Snort employ detection signatures to detect buffer overflow attacks. The main idea behind this detection method is to create signatures which describe crucial elements of the buffer overflow attack such as the NoOP sled and the existence of certain shellcode parameters such as `'/bin/sh'`. In turn, the attackers alter their attacks so that the aforementioned crucial elements in the resulting evasion attacks differ from the signatures which aim to detect them. Consequently, detector developers then aim to design new signatures which can detect variants of an attack. Within this context, this thesis employed GE to optimize buffer overflow characteristics in Chapter 6 and linear GP in Chapter 7 to evolve the shellcode at the assembly language level. By contrast, this chapter focuses on evading anomaly detectors using linear GP to evolve the shellcode at the system call level.

Anomaly detectors monitor applications in order to detect any unusual behaviour. The main idea behind anomaly detection is that any deviation from normal behaviour is an anomaly and hence should be investigated further to determine whether it is an attack. Commonly, anomaly detectors monitor the system calls [30] [89] [27] [26] [103] [72] [110] [91] [38] which an application makes since most of the crucial interactions between the operating system kernel and the application are facilitated through system calls. The most common shellcode design involves calling execve to spawn a UNIX shell. Therefore, the attackers [105] [99] [96] [32] [56] [34] design their payload so that, instead of calling execve shellcode, which creates a deviation in application behaviour, they utilize a set of system calls which are likely to be utilized by the target application during normal operation. The search for such an attack is also known as a mimicry attack search and is the focus of this chapter. The objective of the 'white-hat' attacker, in this case, is to test the anomaly detectors for any blind spots or detection weaknesses. Furthermore, another important benefit of investigating such

'white-hat' attackers is that they can aid in designing detectors which are resilient to such mimicry attacks. Detector parameterization is another benefit of the proposed approach. Given that anomaly detection is sensitive to detector configurations [97] [46], the parameters of the anomaly detectors should be configured carefully to provide an effective defense (i.e. high detection rates with low false alarm rates). To this end, the proposed 'white-hat' attacker can be deployed against various configurations of the same detector to identify the configuration which is most resistant to mimicry attacks.

8.1 Background and Motivation

The previous work relevant to mimicry attack generation is discussed in Section 3.2, whereas the relevant work on anomaly detection is discussed in Section 3.1.

The common trait in mimicry attack generation [105] [99] [96] [32] [56] [34] is that the detectors were used as 'white-box' systems. This implies that knowledge which is necessary to design such a system is expensive and the exhaustive search requires a limited semantic coverage in order to minimize computational cost. In particular, such research has for the most part concentrated on the Stide open source host-based anomaly detector [30] or its improved versions [89] [27] [26] [103]. The design of exploits then boils down to locating sequences of system calls which match the contents of an anomaly detector's normal behaviour database while reaching the behavioural objectives of the original 'core' exploit successfully. A minimalist configuration of the anomaly detector is utilized under the general observation that it is easier to make a strong detector if the alphabet of permitted instructions is small. Thus, any weakness detected under these conditions will be magnified when more realistic configurations of the detector's normal behavioural database are employed (a larger alphabet of normal behaviour will result in even more opportunities for defeating the detector). Such a problem was shown to be of a sufficiently focused form for exhaustive search algorithms to solve the problem in seconds with solutions returning the equivalent of a zero anomaly rate [105] [99] [96] [32] [56] [34]. This thesis aims to establish that it is possible to generate successful attacks without internal knowledge of the detector. Therefore, the feedback from the detector is limited to

the anomaly rate alone where this is readily provided to the user as part of normal operation. Hence, no use is made of the internal data structures or algorithms specific to a particular detector. The design of mimicry exploits now takes the form of a search process in which the anomaly rate from the detector is used as the only guide to the effectiveness of the exploit. In addition, such an approach has the potential for extending the applicability of vulnerability testing to commodity detectors for which access to source or binary recompilation and signature or behaviour databases is often prohibited.

Furthermore, this thesis makes the observation that there are two stages to a buffer overflow attack [46] [47]. The first stage is called the preamble, which are the actions which the attacker needs to take to gain control of the application. For example, in the ftpd attack [5] the attacker gains control of the application by utilizing a command which causes the corruption of the memory space of the application. This allows the attacker to direct the application to the attacker's shellcode. After the execution is directed to the attacker's code, the attacker has full control. This stage is called the exploit in which the attacker can alter the system calls which the payload executes in order to evade detection. Although Wagner et al. [105] were aware of the preambles, the previous work [105] [99] [96] [32] [56] [34] focused on the design of the exploit alone and the results were reported on the anomaly rate of the exploit alone. The results discussed in this chapter demonstrate that although such an emphasis results in exploits with a zero percent anomaly rate, the anomaly rate of the attack (i.e. preamble and exploit) produces anomaly rates above zero percent.

8.2 Methodology

In this case, the objective is to develop an automated process for building 'white-hat' attackers within a mimicry attack context. By 'mimicry' the availability of the 'core' attack is assumed where this establishes a series of behavioural objectives associated with the exploit. Therefore, the goal of the automated 'white-hat' attacker will be to establish as many specific attacks corresponding to the exploit associated with the 'core' attack. Candidate mimicry attacks will take the form of system call sequences which can avoid detection or at least minimize the anomaly rate at the corresponding

detector. By 'white-hat' it is implied that the underlying objective is to use the attacks to improve the design of the corresponding detectors via vulnerability/penetration testing.

As discussed in Chapters 6 and 7, previous research has established the suitability of the genetic programming (GP) machine learning paradigm as an appropriate process for automating specific processes associated with the design of buffer overflow attacks [48] [44]. In this chapter, the approach is extended to a general framework for mimicry attack generation, based upon the evolution of system call sequences rather than the generic parameters of an attack. The general motivations for using GP within this context are discussed below.

1. **Goal-Based Objectives.** All machine learning algorithms require a method for expressing the suitability of the current solution. Typically, this takes the form of a distance metric (e.g. sum square error) evaluated over a set of training exemplars. Such an approach implies that it is possible to define the behaviour relative to a set of exemplars describing input and desired output behaviours. This mode of operation has lead to the widespread use of machine learning methods within the context of intrusion detectors. However, in the case of mimicry attack generation, the goal is to discover programs consisting of system calls which mimic the behaviour of a predefined 'core' (buffer overflow) attack. Learning is therefore directed by information supplied following an interaction with an environment. The environment in this case takes two forms, the anomaly rate returned by the detector for the candidate attack (suggested by GP in this case) and the degree to which the generic goals associated with achieving the 'core' attack are reached. Such a mode of operation precludes the vast majority of machine learning paradigms as they make explicit assumptions regarding the relationship between representation (the components from which a solution is built) and how the objective is specified [6]. Typical examples include the smoothness constraint associated with neural networks or kernel methods. Conversely, GP has no limitation on the formulation of objectives.

2. **Representation.** Given that the objective is to design mimicry attacks, it follows that the representation utilized by the machine learning methodology

must take the form of a program (i.e. a sequence of system calls) which correspond explicitly to the attack. This requirement precludes the use of any other machine learning technique. For example, kernel and neural network models are based upon an abstract connectionist representation, broadly applicable to data driven classification, regression, and clustering problem domains. Decision tree methods provide solutions which take the form of a set of partitioning rules. In essence all these methods utilize a representation motivated by a bias to a data driven model of learning.

3. **Intron Code.** Solutions from GP take the form of a program in a predefined language. However, not all instructions comprising a solution contribute necessarily to the underlying operation of the individual, where this artifact is synonymous with the 'introns' of biological genomes. Such introns result from the stochastic action of the search operators (crossover and mutation) and might account for as much as 80% of the instructions in an individual. Typically, instructions corresponding to intron behaviour are removed post learning, as they make no contribution to the (functional) operation of the individual. In the context of this work, however, intron code aids the obfuscation of the real intent of the code. Thus, although not contributing to the design of a valid exploit, code corresponding to intron behaviour will aid the minimization of detector anomaly rates.

Anomaly detectors which are utilized in the experiments detailed in this chapter are discussed in Section 4.2. As discussed in Section 4.2, the traits which make these detectors suitable are (1) they employ different detection methodologies while monitoring system calls and (2) they provide a suitable detection feedback, in the form of anomaly rates and delays, which can be utilized to guide the evolution of the attacks. Vulnerable applications which are employed in the experiments described in this chapter are detailed in Section 8.2.1. Section 8.2.2 details the GP framework utilized for automating mimicry attack generation under a core buffer overflow attack. The fitness calculation of the attacks is discussed in Section 8.2.3.

8.2.1 Vulnerable Applications

In the experiments detailed in this chapter, four Linux applications were employed, namely traceroute, restore, ftpd, and samba, which have known and documented vulnerabilities. These are also the vulnerable applications used in the mimicry attack literature [105] [96] [99] [77].

All four vulnerabilities can be found on older Linux distributions, which were released before 2004. Given that new vulnerabilities are continuously being discovered, these comparably older attacks are chosen mainly because the ground truth of these vulnerabilities are well known. In other words, the descriptions of the vulnerabilities were thoroughly discussed and numerous exploits were published.

As buffer overflow attacks get increasingly difficult to exploit (due to non-executable stacks and stack overflow protection of the compilers such as gcc) the attackers change their tactics and take advantage of the input validation attacks against server-side vulnerabilities [95]. However, the proposed approach relates to a wider scope of attacks, where the objective of the attacker is to inject a malicious code, which can achieve attacker's goals while remaining undetected. Although the vulnerable applications discussed in this thesis provide a foundation for developing an artificial arms race between attackers and detectors, the proposed approach can also be generalized to other types of attacks where the goal is to evolve the code to be injected according to the specifications of the attacker, as defined by the fitness function and the instruction set.

Discussion of 'Normal Behaviour'

For each application, normal use cases, which represent the scenarios of legitimate use, were developed. The normal use cases developed in this research are similar to the use cases discussed in previous work [105] [96] [99] [77]. Moreover, this thesis employs multiple use cases for each application to provide a more realistic normal behaviour database for the detectors.

Compared to the normal use cases developed for this work, previous work [105] [96] [99] [77] employed fewer normal use cases to train the anomaly detectors, under the general observation that it is easier to make a strong detector if the alphabet of

permitted instructions is small. Thus, any weakness detected under these conditions will be exasperated when more realistic configurations of the detector's normal behavioural database are employed (a larger alphabet of normal behaviour will result in even more opportunities for defeating the detector).

Utilizing multiple normal use case scenarios aims to investigate how the anomaly rates change when different training sets are employed as detailed in Section 8.4 and Appendix A. Moreover, utilizing multiple use cases does not aim to create an exhaustive list of all possible normal use cases. Compared to the superset of all possible system call sequences that an application can make, multiple use cases still represent a small subset.

Traceroute and restore vulnerabilities can be exploited locally whereas ftpd and samba vulnerabilities can be exploited remotely as well. Deploying an attack locally or remotely can affect the success of the mimicry attack, mainly for two reasons. First, the 'normal behaviour' of an application providing service over the network is likely to be more extensive than an application running locally since it contains additional functionality for network communication. Second, deploying an attack over the network may cause the preamble to be more anomalous, since access to the application is limited further (i.e. over the network). Therefore, the attacker may not have sufficient control to prevent the application from generating anomalous behaviour during the break-in. An in-depth discussion of vulnerable application characteristics is provided in Chapter 10.

Traceroute

Traceroute is a network diagnosis tool which is used to determine the routing path between a source and a destination by sending a set of control packets to the destination with increasing time-to-live values. A typical use of traceroute involves providing the destination IP, whereas the application returns information on the route taken between source and destination.

Red Hat Linux 6.2 is shipped with Traceroute version 1.4a5 which is susceptible to a local buffer overflow exploit which provides a local user with super-user access [2]. The attack takes advantage of a vulnerability in malloc chunk, and then uses a

debugger to determine the correct return address for taking control of the program. In order to analyse traceroute behaviour under normal conditions, five use cases were developed, Table 8.1; whereas in previous research [96] only one normal use case was used for training, namely use case 1 from Table 8.1.

Table 8.1: Traceroute normal use cases

Use Case	**System Calls**
1. Target a remote server	736
2. Target a local server	260
3. Target a non existent host	153
4. Target localhost	142
5. Help screen	24

Restore

Restore is a component of UNIX backup functionality which restores the file system image taken by the dump command. Files or directories can be restored from full or incremental backups.

Restore version 0.4b15 on Red Hat Linux 6.2 is vulnerable to an environment variable attack where the attacker modifies the path of an executable and runs restore. This results in executing an arbitrary command with super-user privileges which leads to a root compromise. In the published attack [3], the attacker spawns a root shell. Table 8.2 summarizes five normal use cases which were developed for restore. As in the previous work [99], in this case, a regular user executing the restore system program to retrieve backup data from a remote backup server has been monitored. However, the above-mentioned monitoring process has been repeated as well for different sizes of files and backup types.

Samba

The samba suite provides printer and file sharing for Windows clients and can run on most UNIX variants. Samba sets up printer and network shares which appear as Windows disks and printers under a Windows operating system.

Table 8.2: Restore normal use cases

Use Case	**System Calls**
1. Restore a small file system dump from a full backup.	2256
2. Restore a small file system dump from an incremental backup.	1027
3. Restore a large file system dump from a full backup.	167207
4. Restore a large file system dump from an incremental backup.	68185
5. Help screen	53

Red Hat Linux 9.0 is shipped with samba suite version 2.2.7a, which has a vulnerability [4] which can be exploited over the network to gain super-user privileges. The buffer overflow occurs when the samba service tries to copy user supplied data into a static buffer without checking. The published attack binds a root shell to a network port. Although Parampalli et al. [77] utilized samba in their experiments, their aim was to investigate whether the attack could be altered in a way to allow the samba service to run after the attack was deployed. Hence, they did not discuss how normal behaviour was generated for the anomaly detector. Table 8.3 summarizes the six normal use cases which are developed for samba. The use cases in Table 8.3 contain behaviour such as mounting/unmounting a samba share, performing file I/O such as file edit or copy and password change.

Ftpd

Red Hat Linux 6.2 is shipped with Washington University Ftp Server (wuftpd) version 2.6.0(1), which provides FTP access to remote users. wu-ftpd 2.6.0(1) is susceptible to an input validation attack in which the attacker can corrupt process memory by sending malformed commands and overwriting the return address to execute the attacker's shellcode. Although the attack [5] is an input validation attack, the deployment is similar to a buffer overflow attack, which is to say, a shellcode is injected into the execution. Table 8.4 summarizes the ten normal use cases which were developed for

Table 8.3: Samba normal use cases

Use Case	**System Calls**
1. Mount a samba share successfully	1156
2. Invalid password while mounting samba share	680
3. Unmount a samba share successfully	186
4. Find and edit a remote file. (Using commands: ls - cd - ls - pico)	254
5. Find and copy a 38MB remote file to a local directory (Using commands: ls - cd - cp)	65648
6. Change samba password remotely	1527

ftpd. Use cases 7 through 10 represent the legitimate errors which a user can make during a normal FTP session. On the other hand, in the previous research [105] wu-ftpd was run on only large file downloads over a period of two days.

Table 8.4: ftpd normal use cases

Use Case	**System Calls**
1. Upload 10K data	2249
2. Upload 60M data	32912
3. Upload 650M data	334252
4. Download 10K data	2252
5. Download 60M data	32908
6. Download 650M data	334244
7. Three failed login attempts	2236
8. Help screen	2017
9. Attempt to access non-existent files and directories	2213
10. Type non-existent commands.	2017

8.2.2 Linear Genetic Programming

The process for designing mimicry attacks is automated using Genetic Programming (GP). The GP paradigm differs from most machine learning methodologies in that a 'population' of candidate solutions is maintained concurrently throughout the search

process [6]. Each candidate solution, or individual, takes the form of a program. Programs are represented, in the case of this application, as a sequence of system calls (as opposed to program characteristics in Chapter 6 or assembly instructions in Chapter 7). As such, the sequences are evolved from a user-predefined set of permitted system calls. Although parameters for the system calls are specified, there is no need to support the specification of internal the state i.e., register values.

The search process progresses through the iterative application of a selection operator, the evaluation of performance associated with the subset of individuals targeted by the selection operator, and the application of search operators. The selection operator selects two individuals from the population in which the selection probability is proportional to the rank of the individual. Search operators are then applied to the individuals, resulting in two children. After the search operators, the children are appended to the population and the population is Pareto ranked. The worst two individuals (i.e. the individuals with the lowest two ranks) are discarded from the population, hence restoring the population size.

The training parameters, which are determined empirically, are detailed in Table 8.5. Individuals are defined using a variable length format, thus population initialization creates individuals with varying program lengths. Search operators take three forms: cut and splice crossover, instruction-wise mutation and instruction swap. Note that all search operators are applied stochastically relative to a predefined probability of application, Table 8.5. The specific details of each operator are detailed in Section 5.4.4 and are described briefly below.

Cut and Splice Crossover. The crossover operator provides a scheme for investigating instruction sequences which exist currently in the population, but in different contexts. The cut and splice crossover operator selects, with uniform probability, separate crossover points on each parent. Therefore, the children can have different lengths from their parents.

Swap. The basic motivation of the swap operator is to provide the opportunity for investigating the significance of different instruction orders within the same individual (the case of a correct instruction mix, but in the wrong order). The

swap operator is applied to a single individual, selecting two instructions with uniform probability and interchanging their position.

Instruction-wise mutation. The mutation operator provides a way to introduce new sequences to the individual. Mutation is applied instruction-wise, that is to say, each instruction is tested independently for modification. If the test returns true then the instruction is replaced with an alternative instruction from a predefined list of instructions. Moreover, the probability of applying the mutation operator decays linearly with the tournament count, thus lowering the likelihood of introducing instructions which are not currently in the population as the population evolves. Effectively, this places more emphasis on the crossover operator as the evolution progresses, thus reinforcing the reuse of system call sequences which were demonstrated earlier to minimize detection.

Table 8.5: Genetic Programming Parameters

Parameter	**Setting**
Crossover	0.9 probability
Mutation	0.01 probability, linearly decreasing to 0 over the tournament limit
Swap	Instruction swap within an individual with 0.5 probability
Selection	Tournament of 4 individuals
Stop Criteria	100,000 tournaments or until the convergence criteria is met
Convergence Criteria	If the Pareto ranks remain unchanged over 10 tournaments
Population	500 individuals with instruction selection probability proportional to the percentage of the instruction in normal use cases
Program Length	Initialized over 240 system calls, maximum 1000 system calls
Replacement	Children replace the lowest ranked two individuals
Training Time	Approximately 2 days
Number of runs	50

The principle GP design decisions are now limited to defining the instruction set (representation) and appropriate goal (fitness function) and a method for achieving multiple objectives. Tables 8.6, 8.7, 8.8 and 8.9 detail the occurrences of the top 20 system calls as executed by the traceroute, ftpd, restore and samba applications respectively.

In the case of traceroute (Table 8.6) it is apparent that the top 20 system calls can cover over 90% of the executed instructions. On the other hand in the cases of ftpd and restore (Table 8.7 and 8.8) the most frequent 2 to 4 instructions cover 99% of the executed instructions. As a final remark, Tables 8.6, 8.7, 8.8 and 8.9 detailing the frequently executed system calls for traceroute, ftpd, restore and samba applications demonstrate that all four applications execute memory allocation and I/O system calls frequently where these are also appropriate for the obfuscation of mimicry attacks.

The top 20 system calls which the application executes in Tables 8.6, 8.7, 8.8 and 8.9 represent the GP instruction set for the application. It is important to emphasize that utilizing the knowledge of frequently executed system calls does not break the 'black-box' assumption. This information can be obtained by analysing the system calls on a local copy of the application and therefore does not require any access to the internal knowledge of the detector.

8.2.3 Fitness Calculation and Pareto Ranking

Pareto Ranking is a method for combining multiple objectives which are possibly conflicting under the concept of dominance [57]. Specifically in the case of a problem in which objectives are being minimized, solution A dominates solution B, if and only if A is as good as B in all objectives and A is better than B in at least one objective.

An individual which is not dominated by any other individual is called a non-dominated individual. Pareto ranking succeeds in reducing the multi-objective vector into a scalar fitness value (i.e. the rank) without combining features or assigning *a priori* weights. The Pareto ranking algorithm, which is detailed in Section 5.4, is employed in the experiments detailed in this chapter where the rank of the individual is equal to the number of individuals which it dominates [57].

Table 8.6: GP instruction set for the traceroute application

System Call	**Occurrence**	**Percentage**
gettimeofday	220	16.73%
write	142	10.80%
mmap	113	8.59%
select	99	7.53%
sendto	99	7.53%
close	93	7.07%
open	86	6.54%
read	75	5.70%
fstat	73	5.55%
munmap	49	3.73%
mprotect	34	2.59%
socket	29	2.21%
recvfrom	28	2.13%
brk	27	2.05%
fcntl	26	1.98%
connect	20	1.52%
ioctl	15	1.14%
uname	14	1.06%
getpid	12	0.91%
time	10	0.76%

Table 8.7: GP instruction set for the ftpd application

System Call	**Occurrence**	**Percentage**
read	182325	24.45%
rt_sigaction	181876	24.39%
alarm	181649	24.36%
write	181596	24.35%
close	11443	1.53%
open	1045	0.14%
time	1037	0.14%
mmap	948	0.13%
fstat	716	0.10%
munmap	506	0.07%
chdir	364	0.05%
fcntl	271	0.04%
getcwd	267	0.04%
socket	256	0.03%
connect	256	0.03%
fchdir	240	0.03%
mprotect	230	0.03%
lstat	227	0.03%
send	195	0.03%
brk	187	0.03%

Table 8.8: GP instruction set for the restore application

System Call	**Occurrence**	**Percentage**
write	211704	88.70%
read	26378	11.05%
lseek	132	0.06%
mmap	91	0.04%
open	67	0.03%
close	56	0.02%
fstat	51	0.02%
mprotect	30	0.01%
munmap	29	0.01%
brk	24	0.01%
fcntl	23	0.01%
rt_sigprocmask	23	0.01%
utime	15	0.01%
unlink	8	0.00%
chmod	8	0.00%
chown	8	0.00%
ioctl	8	0.00%
stat	8	0.00%
rt_sigaction	8	0.00%
_llseek	8	0.00%

Table 8.9: GP instruction set for the samba application

System Call	**Occurrence**	**Percentage**
read	28329	41.03%
_llseek	9612	13.92%
select	9398	13.61%
gettimeofday	9395	13.61%
send	9389	13.60%
fcntl64	1133	1.64%
stat	440	0.64%
open	218	0.32%
close	205	0.30%
munmap	153	0.22%
mmap	126	0.18%
mmap2	126	0.18%
getegid32	99	0.14%
geteuid32	97	0.14%
time	91	0.13%
getsockopt	72	0.10%
mprotect	46	0.07%
umask	44	0.06%
setresgid32	44	0.06%
write	32	0.05%

In terms of mimicry attack characteristics, the following objectives are established.

1. **Attack Success.** The original attack contains a standard shellcode which uses the execve system call to spawn a UNIX shell upon successful execution. Execve is a system call which executes the program given as the first argument. Since execve is not a frequently used system call for traceroute, restore, samba and ftpd, it is expected that the original attack could be detected easily. To this end, a different strategy is employed for defining the exploit such that the need to spawn a UNIX shell is eliminated [45]. Typically, most programs perform I/O operations – in particular to open, write to / read from and close files. Tables 8.6, 8.7, 8.8 and 8.9 demonstrate that UNIX applications frequently use open / write / close system calls. Therefore the goal of the attack is altered to involve the following three steps which mimic the goals of the original shell code attack (i.e. to gain super-user privileges):

   (a) open the UNIX password file ('/etc/passwd');

   (b) write a line, which provides the attacker a super-user account which can login without a password;

   (c) close the file.

   The objective of the search process conducted by GP is to discover a sequence of system calls (and appropriate arguments) which perform the above three steps in the correct order (i.e. the attack cannot write to a file which it has not opened), while minimizing the anomaly rate from the detector. A behavioural success function rewarding the above behaviour awards a total of 5 'points' for establishing the behavioural steps for the 'core' attack in Figure 8.1.

2. **Anomaly Rate.** The anomaly rate represents the principal metric for qualifying the likely intent of a system call sequence; a would-be attacker naturally wishes to minimize the anomaly rate of the detector. Again, no 'internal knowledge' is necessary as detectors provide alarm rates as part of their normal mode of operation. Moreover, as indicated in Section 8.1, the attacker also needs to minimize the anomaly rate of the exploit. Provided that the preamble is not

(a) Success = 0

(b) IF the sequence contains open ('/etc/passwd') THEN Success += 1

(c) IF the sequence contains write ('toor::0:0:root:/root:/bin/bash') THEN Success += 1

(d) IF the sequence contains close ('/etc/passwd') THEN Success += 1

(e) IF open precedes write THEN Success += 1

(f) IF write precedes close THEN Success += 1

Figure 8.1: Fitness function for establishing the objectives of modifying the UNIX password file

highly anomalous, relatively low attack anomaly rates can be accomplished by utilizing very concise exploits. However, this is not always the case, thus the preamble is added to facilitate the identification of the most appropriate content after the exploits are evolved.

3. **Delay.** In addition to reporting anomaly rates, pH and pHsm respond to anomalies by enforcing delays. Delay is an exponential function of the locality frame count, which imposes longer delays when the anomalies are clustered together. Consequently, even though the attack minimizes the anomaly rate, it can still be detected if the remaining anomalies are clustered together. Therefore, the attacker aims to minimize the delays associated with the attacks by preventing the clusters of anomalies from occurring in the exploit.

4. **Attack Length.** Attack length is not an immediate concern for the attacker; longer attacks potentially provide more obfuscation. However, attack length appears as an objective to encourage GP to perform a wider search for solutions (i.e. short solutions will be included as well as longer under the Pareto methodology). The longer attacks (i.e. with 1000 system calls) are tested against the vulnerable applications to ensure that the length of the shellcode does not have any side effects, which can prevent the attack from being deployed successfully.

8.2.4 Discussion of the Search Space Size

The mimicry attack generation problem can be considered as a search for a sequence which provides favourable detection characteristics such as low anomaly rates and delays. As seen in Table 8.5, the maximum sequence length is defined to be 1000 and as defined by the instruction sets (Tables 8.6, 8.7, 8.8 and 8.9), each element of the sequence (i.e. a system call) can take 20 values.

Given the maximum length of the sequence l, and the number of values each element in the sequence can take n, the total number of candidate solutions can be calculated with $n \times n \times \ldots \times n = n^l$. Therefore, the search space size (i.e. the total number of candidate solutions), which is defined by the experiment parameters and the instruction set, is 20^{1000}, which is approximately 10^{1301} for all four vulnerable application scenarios and for all five anomaly detectors.

Obviously, if the attacker chooses to utilize all the system calls available in the mimicry attack search, depending upon the operating system type, the number of values which each instruction can take increases to approximately 200 system calls. Therefore, the search space size would expand to 200^{1000}, which is approximately 10^{2301}. This is likely to be the case for a 'white-box' approach and the search space of 'white-box' approaches is discussed in Section 9.3.6.

8.3 Results

In order to establish the effectiveness of the mimicry attack generation methodology, the anomaly rates of the original attacks need to be determined. To do so, original attacks [2] [3] [4] [5] are downloaded from the SecurityFocus website[1] and deployed against the detector configurations detailed in Section 4.2 on the four vulnerable applications detailed in Section 8.2.1. The anomaly rate of the preambles and the original exploits are detailed in Tables 8.10 and 8.11, respectively. Furthermore, Table 8.12 details the anomaly rates of the original attacks (i.e. preamble + exploit).

As discussed in Section 4.2.3, pH with a schema mask (pHsm) employs schema

[1] The URL for the SecurityFocus website, as of November 2008, is `http://www.securityfocus.com`.

masks, which results in two differences from the original pH. First, it has a longer sliding window and second, it has an additional parameter (i.e. the schema mask) which the attacker needs to 'guess.' Therefore, two deployment scenarios are employed. In the first scenario, the attacker does not know the schema mask employed in the target detector whereas in the second scenario the attacker possesses this information. By utilizing two pHsm configurations, the goal is to identify the main characteristic which makes pHsm more resistant against mimicry attacks (i.e. the longer sliding window length versus the 'unknown' schema mask).

In a GP framework multiple solutions are maintained; hence multiple mimicry attacks are generated. Table 8.14 details the anomaly rates of the mimicry attacks which produced the lowest anomaly rate for each detector-application pair and Table 8.13 details the corresponding exploit anomaly rates. It is apparent that mimicry attacks produce smaller anomaly rates than the original attacks. In the case of traceroute, mimicry attacks manage to reduce the anomaly rate from 60% or above to 10% or below. Furthermore, results indicate that pHsm is effective in the case of samba, restore and traceroute where the anomaly rate of the mimicry attack against pHsm is greater than the mimicry attack against pH (Table 8.14).

Furthermore, it is crucial to emphasize that the framework is employed on all detector configurations with the same settings. Previous results [45] [49] show that it is possible to enhance results by using different training settings for different configurations. For example, although the anomaly rate Stide reports for the traceroute mimicry attack is 10.96%, it can be improved further by using greedy search operators [45].

As discussed in Section 8.1, attacks consist of two components, the exploit and the preamble. Table 8.11 shows that the exploits which are employed by the original attacks are detectable with fairly high anomaly rates, whereas the mimicry exploits which GP generates have substantially lower anomaly rates (Table 8.13). Preambles have a considerable impact on the detection of an attack, hence, a further analysis of preambles is provided in Section 8.5.

A prominent result is that none of the attacks deployed completely undetected. In previous mimicry attack research [105] [99] [96] [32] [56] [34], the attack was considered

to evade detection completely if the exploit component raised no alarms. However, even though the exploit raised no alarms (e.g. the ftpd attack against pHsm in Table 8.13), the actions taken by the attacker to take control of the application (i.e. the preamble) raised alarms (Table 8.10), hence producing non-zero anomaly rates (Table 8.14).

Table 8.10: Anomaly rate of the preamble component of the attacks (both original and mimicry)

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	6.98%	36.49%	77.78%	8.54%	22.04%
restore	77.82%	81.01%	93.67%	35.08%	13.29%
samba	3.57%	9.97%	12.07%	6.78%	6.34%
ftpd	19.04%	21.94%	14.30%	6.11%	6.88%

Table 8.11: Anomaly rate of the original exploits

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	71.48%	73.91%	83.06%	47.89%	70.21%
restore	88.13%	90.70%	98.30%	48.84%	15.53%
samba	60.04%	60.51%	99.60%	25.53%	21.15%
ftpd	47.52%	47.85%	57.29%	13.65%	18.86%

Table 8.12: Anomaly rate of the original attacks

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	61.26%	66.27%	81.79%	38.78%	31.19%
restore	84.69%	87.49%	96.77%	44.26%	14.00%
samba	10.16%	16.02%	99.95%	9.03%	5.73%
ftpd	22.78%	25.54%	20.27%	7.15%	6.91%

Table 8.13: Anomaly rate of the best mimicry exploits

	Stide	**pH**	**pHsm**	**pHsm (mask unknown)**	**Markov Model**	**Neural Networks**
traceroute	16.67%	11.71%	0.00%	27.60%	0.10%	2.47%
restore	0.40%	0.10%	0.20%	0.31%	0.10%	2.90%
samba	0.50%	0.10%	0.00%	29.23%	0.10%	16.68%
ftpd	57.14%	0.10%	0.00%	35.55%	0.10%	3.46%

Table 8.14: Anomaly rate of the best mimicry attacks

	Stide	**pH**	**pHsm**	**pHsm (mask unknown)**	**Markov Model**	**Neural Networks**
traceroute	10.96%	18.29%	2.71%	29.28%	0.20%	1.63%
restore	46.25%	48.57%	54.52%	57.92%	21.05%	5.60%
samba	3.00%	8.11%	7.36%	15.84%	5.45%	5.77%
ftpd	19.30%	16.11%	10.62%	20.19%	4.47%	1.26%

Locality Frame Counts and Delays

Although Stide keeps track of the locality frame count, pH employs the locality frame count to delay the processes. The locality frame count keeps track of the mismatches over a given time period (by default, the previous 128 system calls). Therefore, a cluster of mismatches produces high locality frame counts whereas the same number of mismatches distributed over the attack produces smaller locality frame count values. pH responds to attacks by slowing down the process based upon the observed locality frame count. The delay associated with the current system call can be expressed with Equation 8.1 [91].

$$delay\_factor \times 0.01 \times 2^{LFC} \tag{8.1}$$

Higher $delay\_factor$ values produce longer process delays and the LFC signifies how many of the past 128 system calls were anomalous. Even a slight increase in locality frame count is sufficient to stop an attack since its effects are exponential and the value remains high until the locality frame moves to a segment with few anomalies. Utilizing the Equation 8.1, Figure 8.2 shows the relationship between the locality frame count values and the resulting delays in seconds. If the observed locality frame count increases to 120, the resulting delay will be close to 10^{35} seconds, which can be expressed in centuries. Sustaining high locality frame counts throughout the attack will enforce high delays multiple times hence increasing the overall delay above 10^{35} seconds. Therefore, once the locality frame count rises above a certain value, pH effectively 'freezes' the attack, hence preventing the successful execution of the exploit. This implies that although the attack achieves a 0% anomaly rate on the exploit component, it can still be detected and stopped by focusing on the preamble alone.

In the experiments, delays are reported as opposed to the locality frame counts. Although the detectors, which monitor system call sequences, can report locality frame counts, locality frame count is not reported as a part of their standard operation. Given the 'black-box' assumption, the feedback to the GP is limited to the outputs from the detector. This means that only the anomaly rate is provided by Stide, Markov Model and Neural Network detectors and anomaly rate. Additionally, the delays are provided by pH and pHsm.

Delays associated with the preambles, original exploits and original attacks are

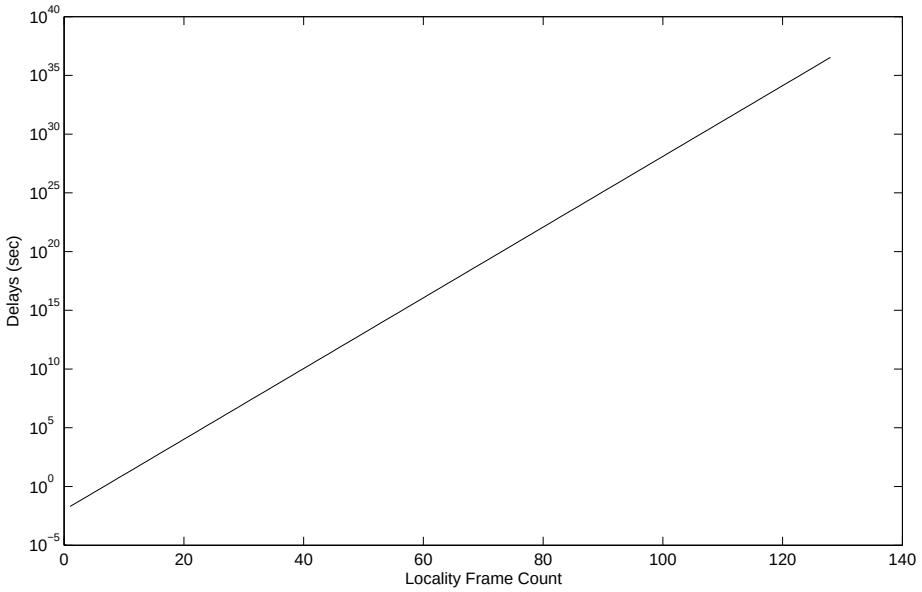

Figure 8.2: Locality frame counts and the associated delays

detailed in Tables 8.15, 8.16 and 8.17, respectively. Furthermore, Tables 8.18 and 8.19 detail the delays associated with mimicry exploits and mimicry attacks respectively. Given that Stide, Markov Model and Neural Network detectors do not utilize process delays, the associated delays in Tables 8.15, 8.16, 8.17, 8.18 and 8.19 are zero.

Although mimicry attacks have shorter delays (i.e. a traceroute mimicry attack against pHsm where the attacker knows the schema mask can deploy in seconds), any delay over 10^{15} is expressed in billions of centuries. Delays in Tables 8.17 and 8.19 demonstrate that even though the attacker manages to achieve zero (or near zero) anomaly rates for the exploits, anomalies from the preamble can generate clusters and prevent the attack from deploying.

The lengths of the best mimicry exploits are detailed in Table 8.20. It is apparent that the exploits generally expand to approximately 1000 system calls, which is the maximum exploit length in the experiments. This suggests that the exploits employ code bloat property of GP to hide the true intent of the exploit.

Table 8.15: Delay associated with the preamble component of the attacks (both original and mimicry)

	pH	**pHsm**
traceroute	0.74	0.63
restore	1.90E+38	1.01E+39
samba	7.95E+27	1.27E+40
ftpd	5.26E+30	8.03E+17

Table 8.16: Delay associated with the original exploits

	pH	**pHsm**
traceroute	4.39E+35	8.51E+35
restore	1.66E+39	3.93E+39
samba	2.97E+30	8.96E+38
ftpd	3.78E+22	4.89E+25

Table 8.17: Delay associated with the original attacks

	pH	**pHsm**
traceroute	4.39E+35	8.51E+35
restore	1.85E+39	4.96E+39
samba	3.11E+30	1.41E+40
ftpd	5.26E+30	4.89E+25

Table 8.18: Delay associated with the best mimicry exploits

	pH	**pHsm**	**pHsm (mask unknown)**
traceroute	1.11	0	1.50E+14
restore	9.94	9.87	11.1
samba	9.94	0	7.37E+12
ftpd	9.94	0	9.86E+17

Table 8.19: Delay associated with the best mimicry attacks

	pH	**pHsm**	**pHsm (mask unknown)**
traceroute	1.91	0.63	1.50E+14
restore	1.90E+38	1.01E+39	1.01E+39
samba	7.95E+27	1.27E+40	1.27E+40
ftpd	5.26E+30	8.03E+17	1.79E+18

Table 8.20: Best mimicry exploit lengths generated against five anomaly detectors in terms of system calls

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	34	118	1000	957	1000
restore	1000	1000	999	1000	1000
samba	1000	1000	1000	983	1000
ftpd	11	1000	994	1000	1000

8.3.1 Traceroute Box Plots

In addition to providing anomaly rates for the best mimicry attacks for each detector-application pair, this thesis also investigates the anomaly rates for all individuals in the population using box-plot analysis [11]. The box plot defines the third quartile, median and first quartile. Whiskers extend from each end of the box to the adjacent values in the data which are within 1.5 times the inter-quartile range from the ends of the box. Outliers are data with values beyond the ends of the whiskers and are displayed with a plus sign. The box plots detailed below summarize the characteristics of 25000 attacks (generated over 50 runs in which each run produced a population of 500 attacks).

Figure 8.3 details the exploit anomaly rate whereas Figure 8.4 details the attack anomaly rate of the population. The box plots indicate that the exploit and attack anomaly rate of the attacks against pHsm is greater than the exploit and attack anomaly rate of the attacks against the original pH. This supports the argument that pHsm is more difficult to evade than the original pH. Furthermore, anomaly rates reported by the Markov Model and the Neural Network are lower than the

anomaly rates returned by the remaining detectors. Table 8.21 details the preamble characteristics of the original traceroute attack.

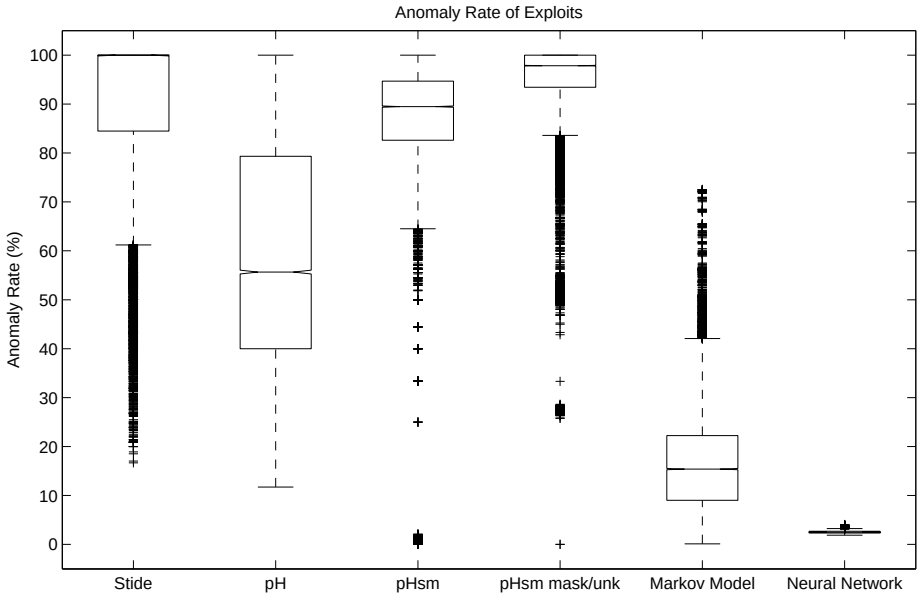

Figure 8.3: Box plot of the mimicry exploit anomaly rate for traceroute

Figures 8.5 and 8.6 detail the delays associated with the exploits and corresponding attacks, respectively. It is important to note that only pH and pHsm enforce delays on anomalous traces, therefore the remaining detectors are considered to have no imposed delays on the attacks. The results indicate that when the attacker does not know the schema mask which pHsm employs, the delays associated with the attacks are higher. Exploit lengths in Figure 8.7 indicate that the populations maintain attacks with varying lengths, except in the case of a Neural Network detector where the majority of the attacks have the maximum number of system calls (i.e. 1000).

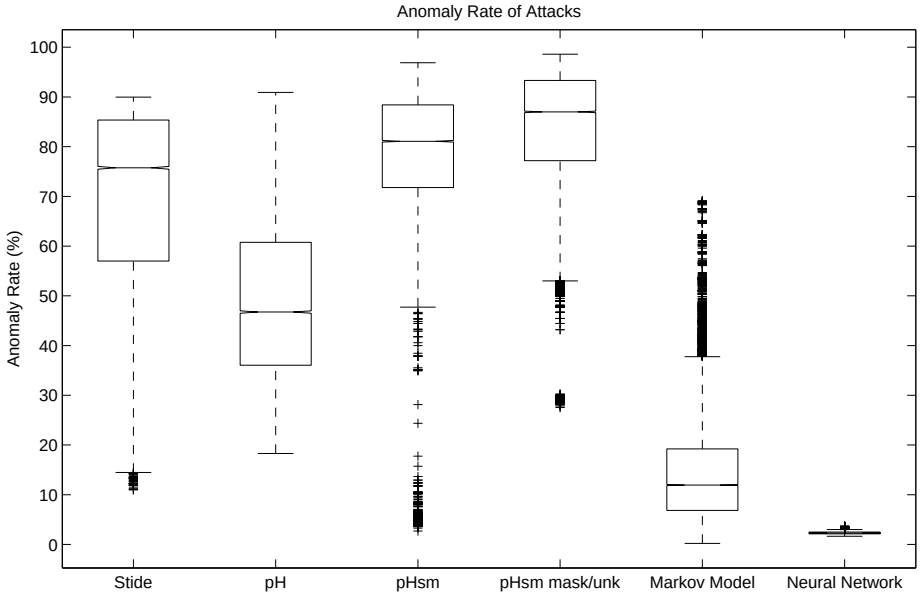

Figure 8.4: Box plot of the mimicry attack anomaly rate for traceroute

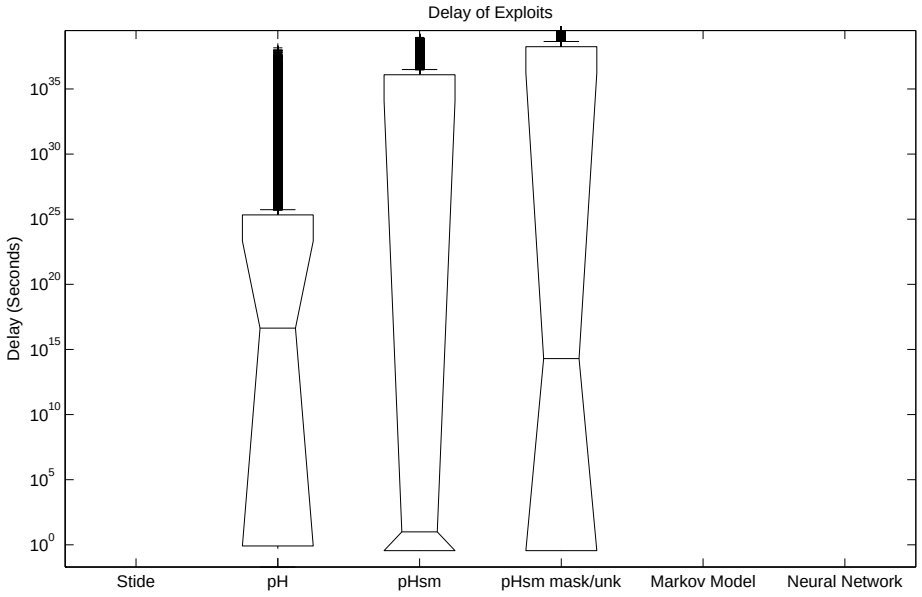

Figure 8.5: Box plot of the mimicry exploit delay for traceroute

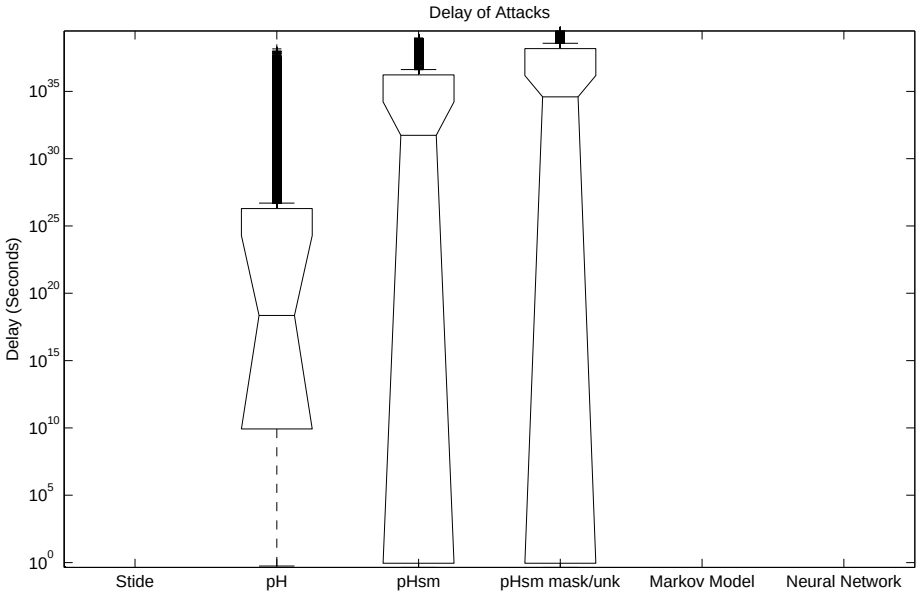

Figure 8.6: Box plot of the mimicry attack delay for traceroute

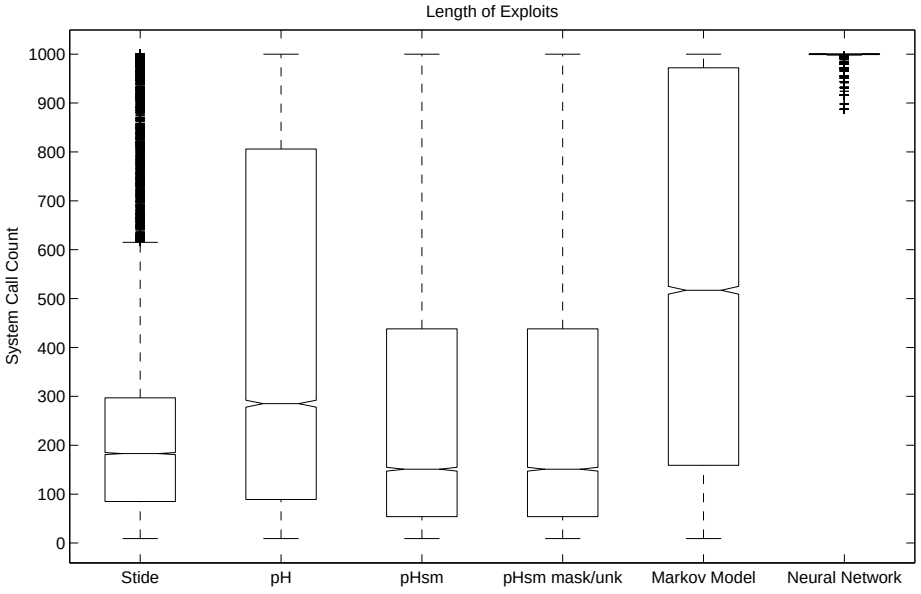

Figure 8.7: Box plot of the mimicry exploit length for traceroute

Table 8.21: Attributes of the original traceroute attack preamble

Attribute	**Value**
Preamble length	53
Anomaly rate (Stide)	6.98%
Anomaly rate (pH)	36.49%
Anomaly rate (pHsm)	77.78%
Anomaly rate (Markov Model)	8.54%
Anomaly rate (Neural Network)	22.04%

8.3.2 Restore Box Plots

Figure 8.8 shows the box plot of exploit anomaly rates for restore whereas Figure 8.9 details the anomaly rates for the corresponding attacks. As with the traceroute box plots, the anomaly rates which pHsm reports are higher than the anomaly rates which the original pH reports. Furthermore, the anomaly rate of the exploits and the corresponding attacks which Stide reports are higher than pH, Markov Model and Neural Network detectors. The exploit length in Table 8.12 suggests that the exploits against pH, Markov Model and Neural Network detectors were longer than the exploits against Stide, therefore longer exploit sizes seem to help in minimizing the anomaly rate of the attack in the case of mimicry attacks against restore.

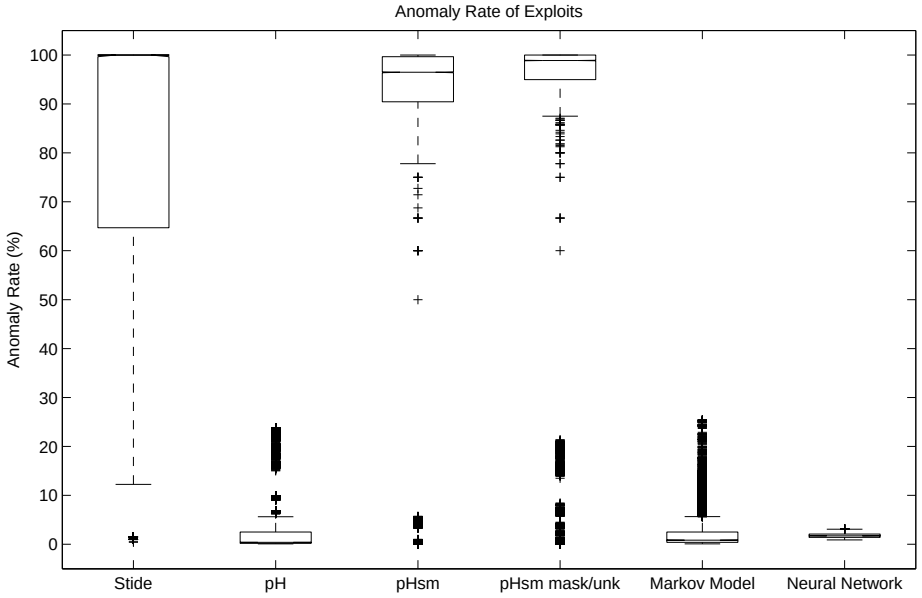

Figure 8.8: Box plot of the mimicry exploit anomaly rate for restore

Figure 8.10 details the delays associated with exploits whereas Figure 8.11 shows the delays associated with the corresponding attacks. The delays indicate that pHsm imposes more delays on the attacks compared to pH. Furthermore, the lengths of exploits are shown in Figure 8.12 which indicates that except for the attacks against pH and the Neural Network detector, the attack length varies over the permitted attack length (i.e. up to 1000).

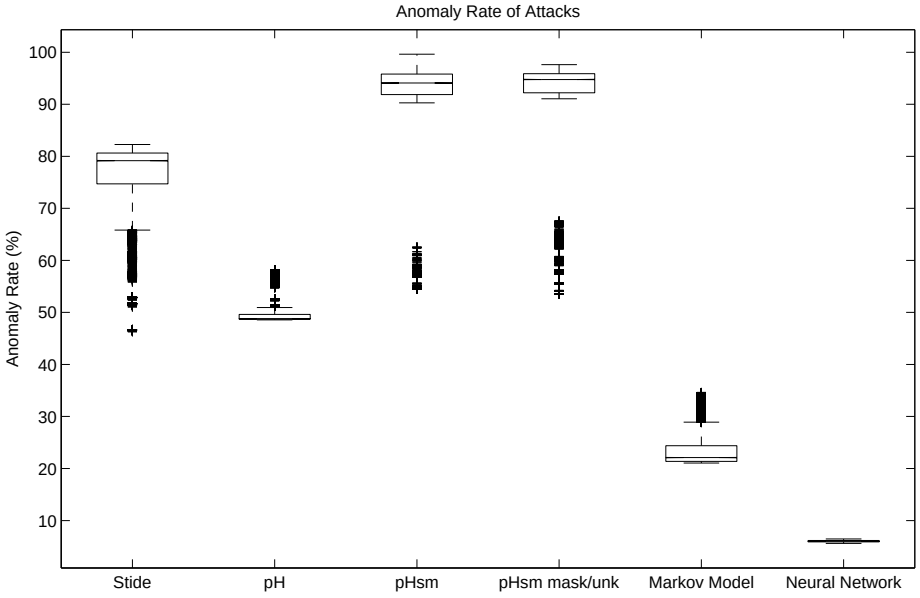

Figure 8.9: Box plot of the mimicry attack anomaly rate for restore

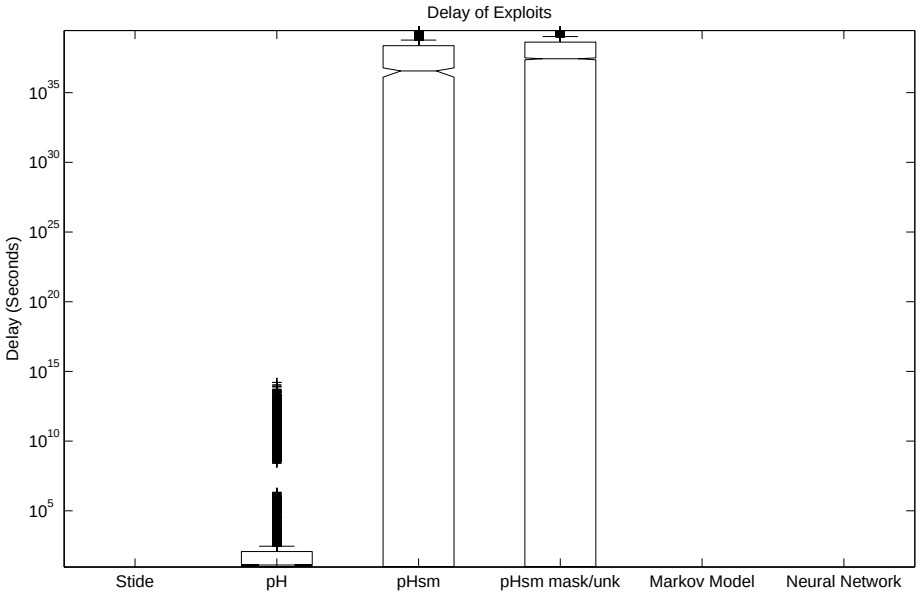

Figure 8.10: Box plot of the mimicry exploit delay for restore

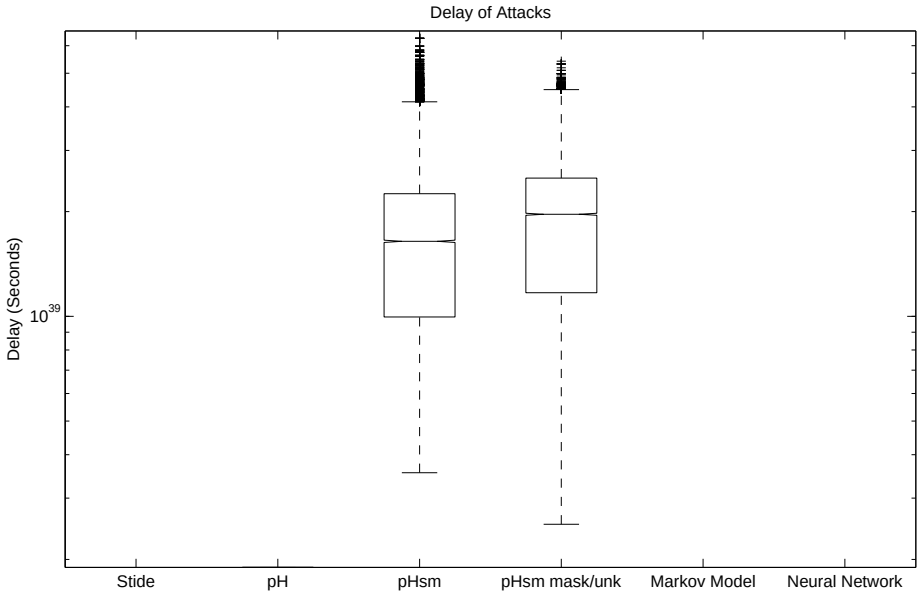

Figure 8.11: Box plot of the mimicry attack delay for restore

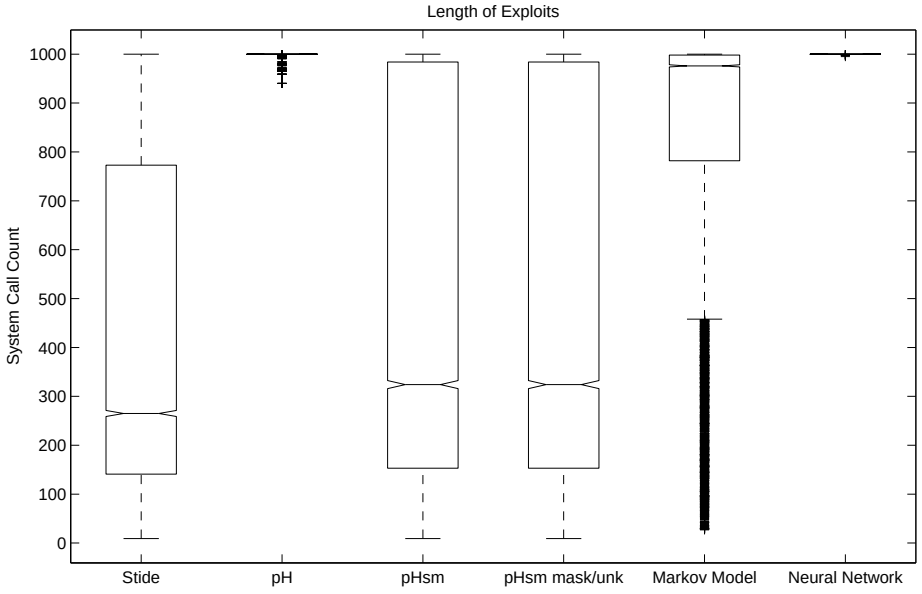

Figure 8.12: Box plot of the mimicry exploit length for restore

The characteristics of the original restore attack preamble are detailed in Table 8.22. Compared to the traceroute preamble, Table 8.21, the restore preamble is longer and more anomalous, which in turn affects the anomaly rate of the attack. This also supports the findings from Figures 8.8 and 8.12 which suggest that longer attacks produce lower anomaly rates.

Table 8.22: Attributes of the original restore attack preamble

Attribute	**Value**
Preamble length	1425
Anomaly rate (Stide)	77.82%
Anomaly rate (pH)	81.01%
Anomaly rate (pHsm)	93.67%
Anomaly rate (Markov Model)	35.08%
Anomaly rate (Neural Network)	13.29%

8.3.3 Samba Box Plots

The box plot of the exploit anomaly rates for samba is provided in Figure 8.13 whereas the box plot for the corresponding attack anomaly rates is provided in Figure 8.14. As with the results on traceroute and restore, the samba results indicate that the anomaly rates of the exploits and corresponding attacks against pHsm are higher compared to the anomaly rates of exploits and corresponding attacks against pH. Furthermore, although the exploit anomaly rate against Stide is fairly high (around 95%), the resulting attacks have low anomaly rates (around 8%). This is due to the fact that the samba preamble, Table 8.23, is lengthy and compared to anomaly rates reported for traceroute and restore preambles, it does not produce high anomaly rates. Thus, although the exploit is anomalous, a shorter exploit, Figure 8.17, can deploy without causing a substantial increase in the attack anomaly rate.

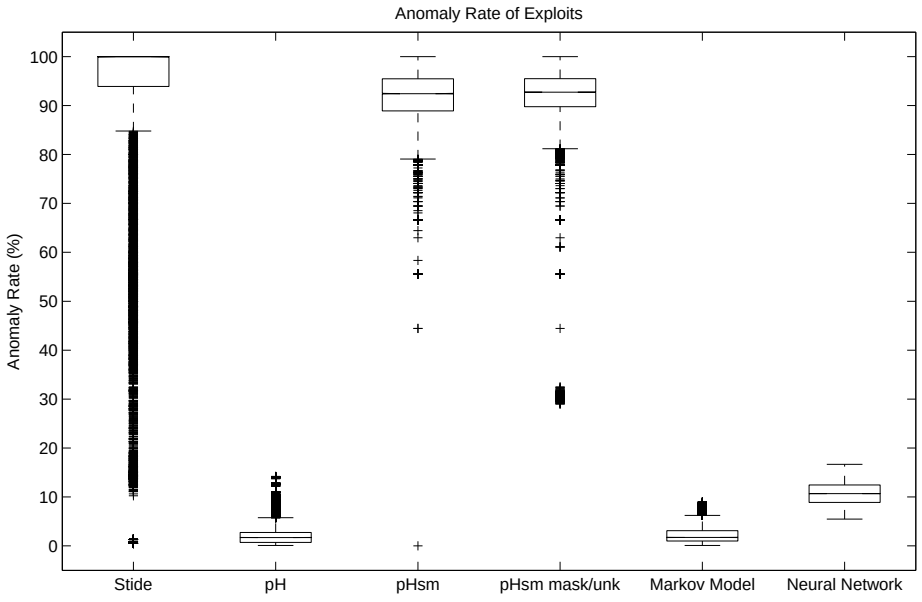

Figure 8.13: Box plot of the mimicry exploit anomaly rate for samba

The delays associated with samba exploits and attacks are detailed in Figures 8.15 and 8.16, respectively. As with the delays observed in the traceroute and restore box plots, pHsm imposes more delays on attacks compared to the delays which pH imposes.

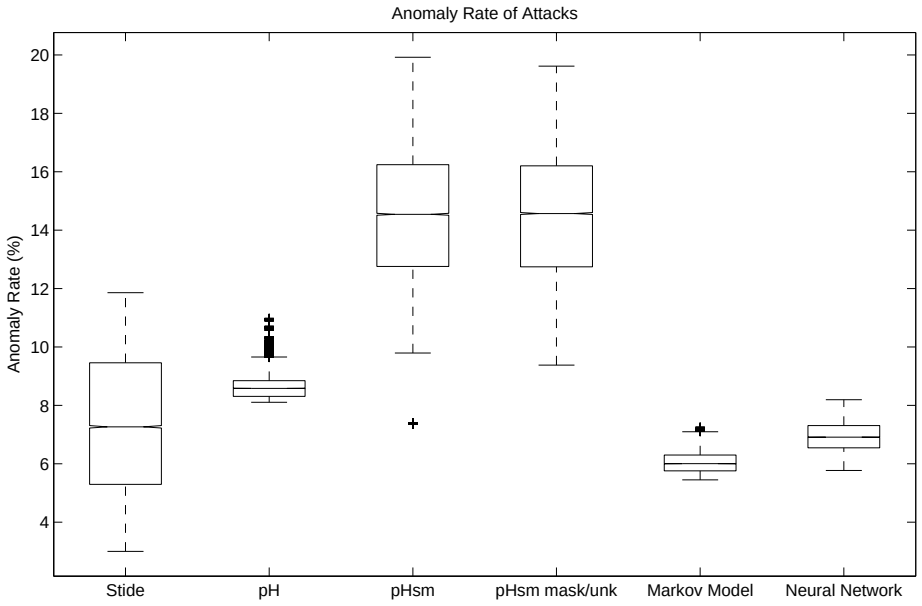

Figure 8.14: Box plot of the mimicry attack anomaly rate for samba

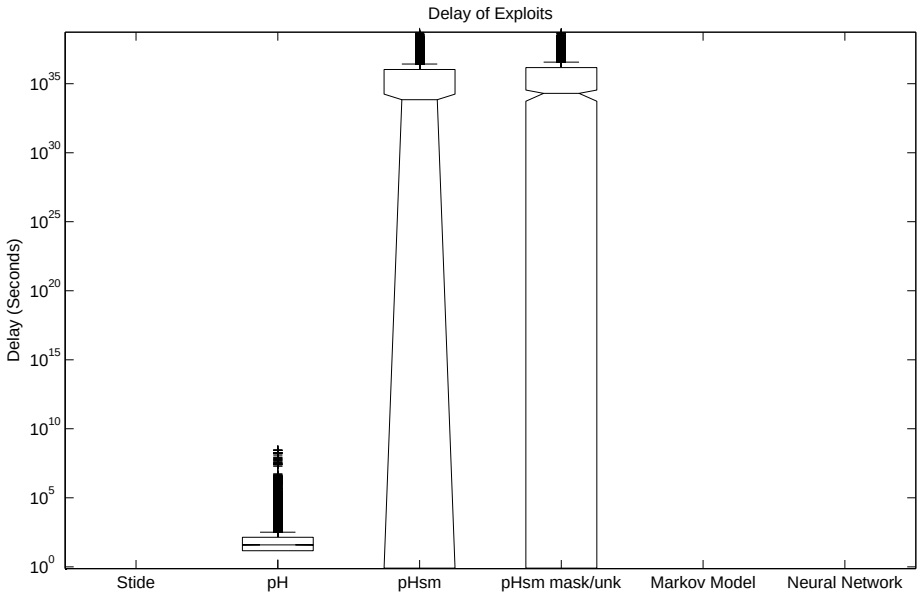

Figure 8.15: Box plot of the mimicry exploit delay for samba

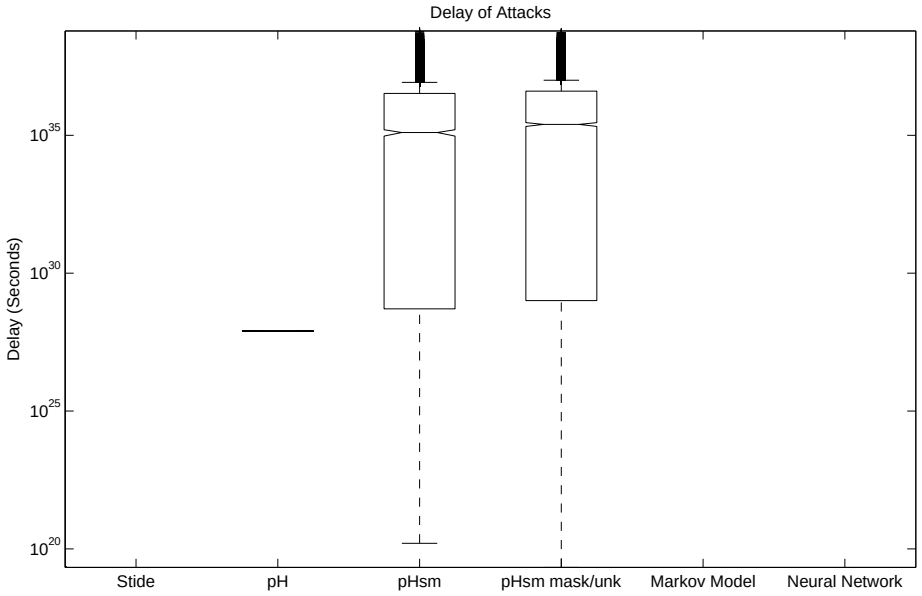

Figure 8.16: Box plot of the mimicry attack delay for samba

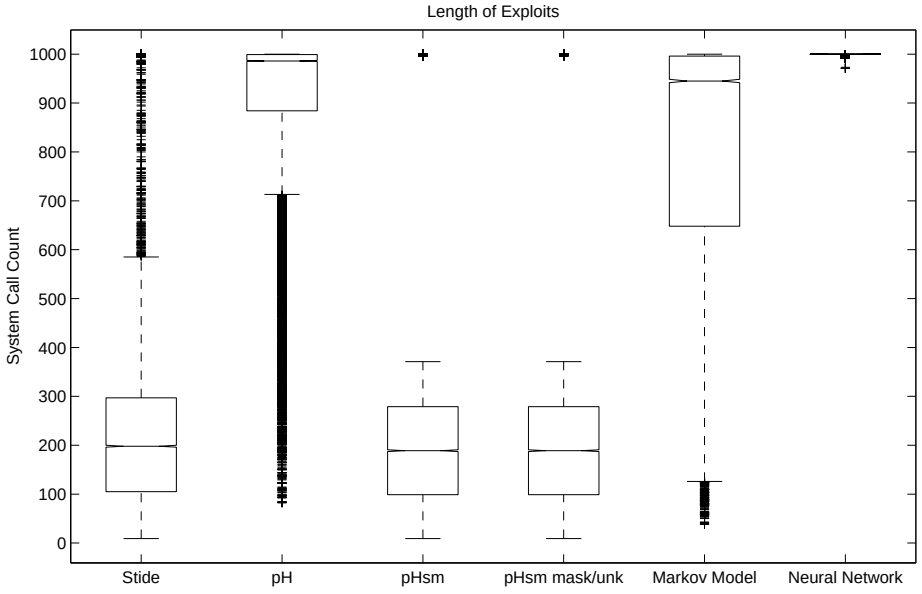

Figure 8.17: Box plot of the mimicry exploit length for samba

Table 8.23: Attributes of the original samba attack preamble

Attribute	**Value**
Preamble length	3868
Anomaly rate (Stide)	3.57%
Anomaly rate (pH)	9.97%
Anomaly rate (pHsm)	12.07%
Anomaly rate (Markov Model)	6.78%
Anomaly rate (Neural Network)	6.34%

8.3.4 Ftpd Box Plots

The box plot which shows the anomaly rate of the exploits for ftpd is detailed in Figure 8.18. Furthermore, Figure 8.19 shows the box plot of the attack anomaly rates for ftpd. The results show that, although the anomaly rate of the exploits against pHsm is greater compared to the anomaly rate of exploits against the original pH, the resulting attack against the original pH has a higher anomaly rate compared to the attacks against pHsm. The box plot of exploit length in Figure 8.22 indicates that this may be due to the fact that the attacks against the original pH are longer than the attacks against pHsm. Thus, the longer exploit size worked against the attacker, hence increasing the attack anomaly rate. Conversely, the anomaly rate of the exploits and the corresponding attacks against Markov Model and Neural Network detectors are fairly low. The box plot of exploit lengths indicates that the exploits against both detectors are fairly long, which is beneficial in reducing the overall anomaly rate of the attacks.

The delays associated with the exploits are detailed in Figure 8.20 whereas the delays associated with the corresponding attacks are provided in Figure 8.21. Delay box plots indicate that the attacks are delayed more if the attacker does not know the schema mask. Furthermore the characteristics of the original ftpd preamble are detailed in Table 8.24.

Table 8.24: Attributes of the original ftpd attack preamble

Attribute	**Value**
Preamble length	2601
Anomaly rate (Stide)	19.04%
Anomaly rate (pH)	21.94%
Anomaly rate (pHsm)	14.30%
Anomaly rate (Markov Model)	6.11%
Anomaly rate (Neural Network)	6.88%

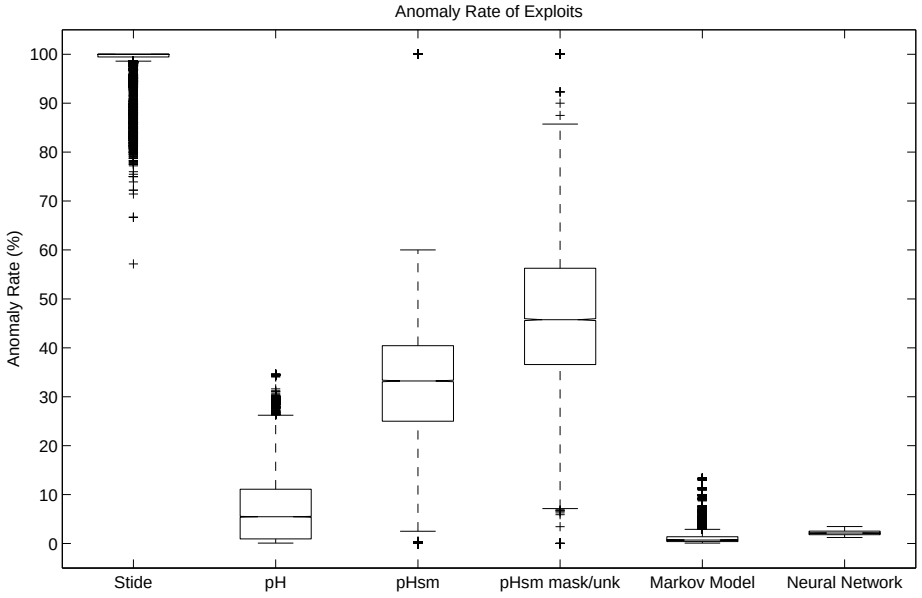

Figure 8.18: Box plot of the mimicry exploit anomaly rate for ftpd

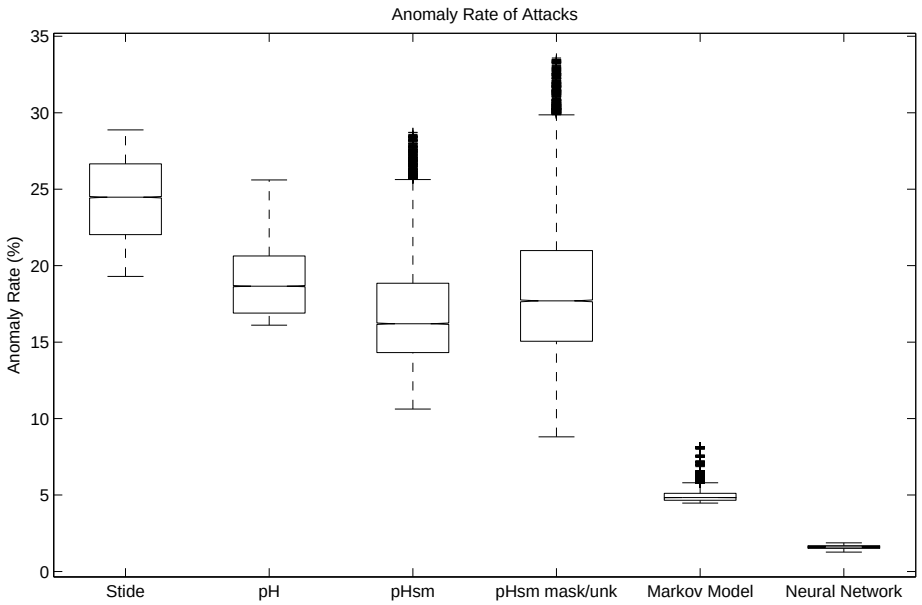

Figure 8.19: Box plot of the mimicry attack anomaly rate for ftpd

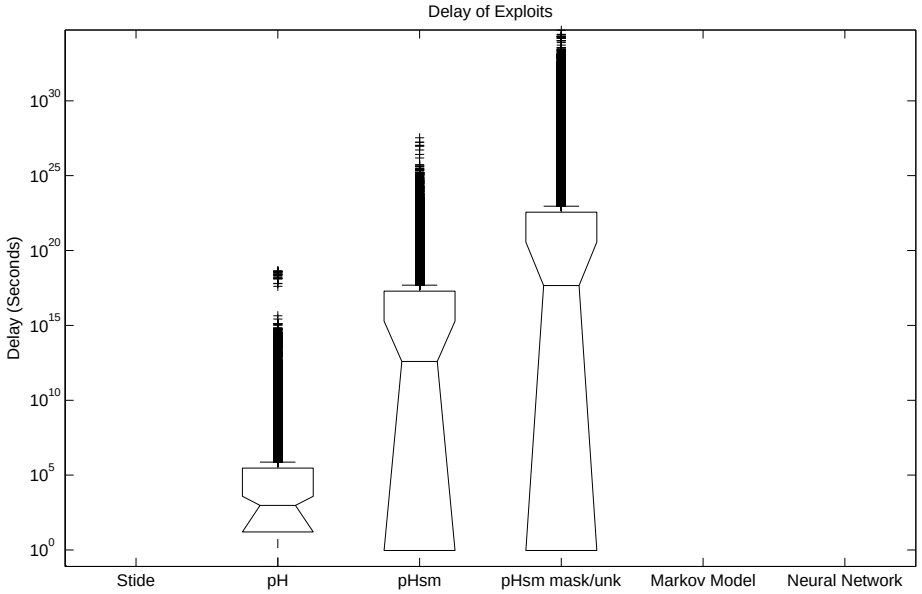

Figure 8.20: Box plot of the mimicry exploit delay for ftpd

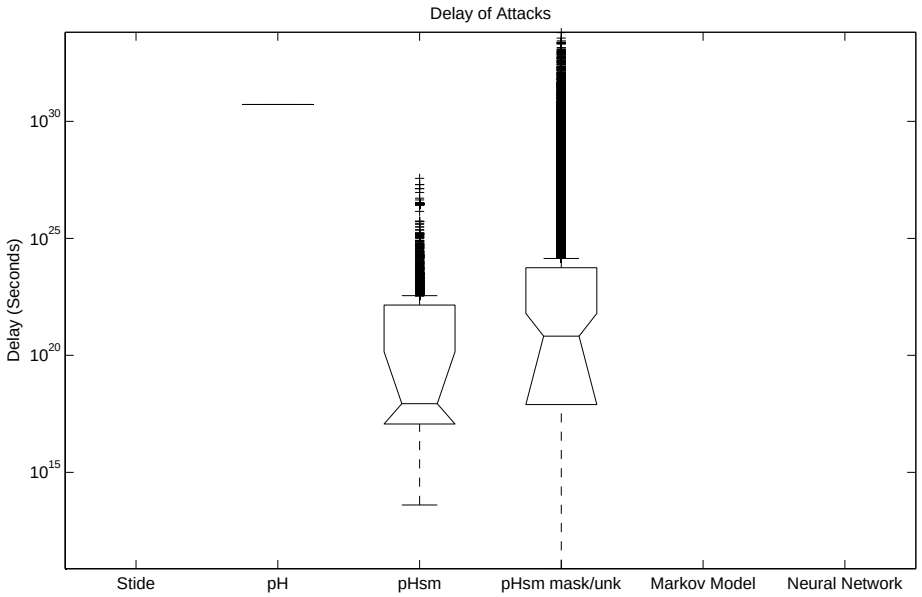

Figure 8.21: Box plot of the mimicry attack delay for ftpd

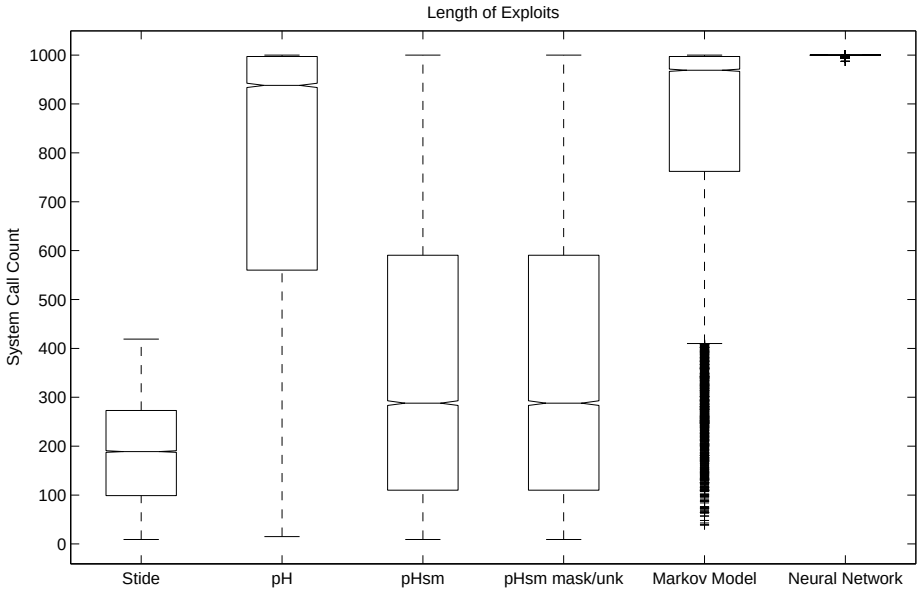

Figure 8.22: Box plot of mimicry exploit length for ftpd

8.4 Training Sensitivity of Anomaly Detectors

Training set selection is an important decision in anomaly detection since it has a considerable impact on the normal behaviour model which the detector builds. Furthermore, testing the anomaly detector on numerous normal behaviour scenarios is crucial in determining a suitable detection threshold.

When anomaly detectors are employed in real-world conditions, an acceptable detection threshold should be established in terms of an anomaly rate to minimize the false positive rate [46]. Thus, the detection occurs if the application activity being monitored produces an anomaly above the threshold. Although such a threshold varies between applications, it is reasonable to assume that it is non-zero. Characteristics of a training set, configuration parameters of a detector and different stages of an attack are some of the factors which may affect the setting of such a threshold. Thus, this section analyses the significance of different training sets for Stide. The same analysis is performed for the remaining anomaly detectors and the results are provided in Appendix A. The objective of the training set analysis is to determine how the anomaly rate changes when the anomaly detectors encounter different normal behaviour scenarios and whether it is possible to determine a suitable anomaly rate threshold.

Tables 8.25, 8.27, 8.26, 8.28, summarize the results of training detectors on different use cases denoting normal behaviour with testing performed over the remaining use cases and the known vulnerability in the case of Stide. The last row – "all normal" – represents the case in which Stide training is conducted over all cases associated with normal behaviour. The last column – "attack" – represents the case in which the original attack is tested on Stide trained with the data set(s) indicated in the corresponding rows. All of the test results are given in terms of percentages where 0% indicates normal and 100% indicates completely anomalous behaviour. The use cases are described in Tables 8.1, 8.2, 8.3 and 8.4.

In Table 8.25, it is apparent that the anomaly rate is sensitive to a wide range of behavioural properties for traceroute. For example, in the case of target host change, Stide produces a 56% anomaly rate when it is trained on the local server trace (case2) and tested on a remote server trace (case1). For instances of Stide trained with a

single trace file anomaly rates on normal use cases vary between 2.8% and 94.3%. Obviously, training Stide over all the normal use cases resulted in a zero anomaly rate for normal use cases (see all normal columns in Tables 8.25, 8.27, 8.26, 8.28). On the other hand, for traceroute, the anomaly rate of the attack varies between 61.26% and 73.87%. The implication of these findings is that, given the 94.3% anomaly rates for some normal behaviour and a 61.26% anomaly rate for the original attack, determining a suitable threshold for anomaly detectors monitoring the traceroute application is not straightforward and will most likely involve a trade-off between the detection rate and the false positive rate.

Table 8.25: Anomaly rates reported by Stide with different training combinations for traceroute

	case1	**case2**	**case3**	**case4**	**case5**	**attack**
case1	0.00%	2.75%	7.43%	15.79%	9.49%	61.26%
case2	56.09%	0.00%	7.43%	15.79%	7.30%	61.26%
case3	78.11%	40.39%	0.00%	15.79%	48.91%	63.06%
case4	94.25%	83.53%	71.62%	0.00%	84.67%	73.87%
case5	73.87%	31.37%	35.14%	15.79%	0.00%	64.26%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	61.26%

Different use cases for samba exhibit different behaviours as shown in Table 8.26. When the detector is trained on case1 (mounting a samba share), it produces high anomaly rates for cases 4 and 5, which involve file I/O on a samba share. Furthermore, it is interesting that the attack produces the second lowest anomaly rate when trained on case1. With the anomaly rate of the attack lower than the anomaly rate of other normal behaviour cases, setting a threshold which can detect the attack without producing false alarms would be very difficult.

In Table 8.27, anomaly rates for the common use cases for restore, in other words typical full and incremental backup operations, are low. Therefore, results indicate that the 'nominal' use of restoring from incremental and full backups (use cases 1 to 4) exhibit similar behavioural properties. On the other hand, when Stide is trained on case 5 (help screen), tests on cases 1 to 4 produce anomaly rates even higher than the attack. Therefore, even though the behaviour of the application is fairly well defined

Table 8.26: Anomaly rates reported by Stide with different training combinations for samba

	case1	**case2**	**case3**	**case4**	**case5**	**case6**	**attack**
case1	0.00%	4.63%	17.13%	91.97%	85.78%	34.28%	14.67%
case2	34.38%	0.00%	66.30%	95.58%	85.81%	49.31%	25.86%
case3	51.66%	55.22%	0.00%	97.99%	99.99%	69.15%	55.76%
case4	72.95%	57.91%	92.27%	0.00%	99.69%	78.91%	59.30%
case5	68.24%	50.30%	92.27%	24.90%	0.00%	73.24%	56.13%
case6	11.87%	3.13%	17.68%	95.58%	85.80%	0.00%	10.55%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	10.16%

for the 'nominal' use cases, it is important to account for the behaviour changes when the program is executed with improper or missing input.

Table 8.27: Anomaly rates reported by Stide with different training combinations for restore

	case1	**case2**	**case3**	**case4**	**case5**	**attack**
case1	0.00%	1.37%	0.00%	0.02%	8.33%	84.69%
case2	1.78%	0.00%	0.01%	0.06%	8.33%	84.69%
case3	1.60%	2.35%	0.00%	0.03%	8.33%	84.69%
case4	1.87%	0.29%	0.01%	0.00%	8.33%	84.69%
case5	98.05%	95.69%	99.97%	99.94%	0.00%	84.76%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	84.69%

Anomaly rates reported for ftpd in Table 8.28 show that uploading and downloading scenarios (cases 1 to 6) are similar in behaviour. However, when legitimate errors are introduced to the FTP sessions (cases 7 to 10), anomaly rates vary between 2.64% and 99.48%. As with restore, the 'nominal' use of the ftpd application (as specified by cases 1 to 6) has few variations and legitimate errors produce high anomaly rates. Given that the original attack produces a 23% anomaly rate whereas normal use produces above 80% anomaly rates for certain use cases, it is difficult to determine an anomaly rate threshold which could distinguish attack behaviour from legitimate errors in the case of ftpd.

From the perspective of generating mimicry attacks, previous work [105] [96] [99]

Table 8.28: Anomaly rates reported by Stide with different training combinations for ftpd

	case1	**case2**	**case3**	**case4**	**case5**	**case6**	**case7**	**case8**	**case9**	**case10**	**attack**
case1	0.00%	0.02%	0.00%	2.32%	0.16%	0.02%	7.59%	0.25%	1.86%	0.25%	24.71%
case2	0.27%	0.00%	0.00%	2.10%	0.14%	0.01%	7.37%	0.00%	1.63%	0.00%	24.54%
case3	0.27%	0.00%	0.00%	2.10%	0.14%	0.01%	7.37%	0.00%	1.63%	0.00%	24.54%
case4	1.70%	0.10%	0.01%	0.00%	0.00%	0.00%	7.23%	0.00%	1.45%	0.00%	23.54%
case5	1.70%	0.10%	0.01%	0.00%	0.00%	0.00%	7.23%	0.00%	1.45%	0.00%	23.54%
case6	1.70%	0.10%	0.01%	0.00%	0.00%	0.00%	7.23%	0.00%	1.45%	0.00%	23.54%
case7	22.24%	94.69%	99.48%	22.66%	94.73%	99.48%	0.00%	15.25%	20.29%	15.25%	34.13%
case8	6.74%	93.64%	99.37%	7.40%	93.69%	99.38%	7.46%	0.00%	5.40%	0.00%	24.61%
case9	2.64%	93.36%	99.35%	2.72%	93.37%	99.35%	7.23%	0.00%	0.00%	0.00%	23.48%
case10	6.74%	93.64%	99.37%	7.40%	93.69%	99.38%	7.46%	0.00%	5.40%	0.00%	24.61%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	22.78%

argues that the selection of the training set is not very crucial because any mimicry attack found within limited normal behaviour will be magnified on detector configurations which include a wider set of normal behaviour, including legitimate errors. Conversely, this thesis argues that training set selection has two important implications for the mimicry attacks: (1) provided that the mimicry attacks produce anomalies (albeit from the exploit), the determination of the threshold would affect whether the resulting mimicry attack remains below the threshold (and evade detection) or not; (2) if the mimicry attacks are generated in a normal behaviour pattern which is not in the target detector's normal behaviour, then they can be detected easily.

Training set experiments indicate that normal behaviour can vary substantially between different normal use case scenarios, which is particularly apparent for ftpd and traceroute. Consequently, anomaly rates for normal behaviour remain high as well, which in turn makes it difficult to identify attacks. For example, if the anomaly rate for normal behaviour is around 70%, it is difficult to detect an attack with a 20% anomaly rate, as in the case of ftpd.

Training set analysis is also performed for pH, pHsm, Markov Model and Neural Network detectors and the results are provided in Appendix A.

8.5 A Closer Look at Preambles

Generally, buffer overflow attacks aim to inject a shellcode into a vulnerable buffer and force the vulnerable application to execute the injected assembly program. The size of the vulnerable buffer is generally too short to inject an entire program, therefore the injected shellcode executes system calls on the target host to spawn a UNIX shell or write to the password file of the host with super-user privileges.

This thesis makes the observation that there are two parts to each attack, the preamble and the exploit. The preamble is composed of the system calls which the application executes during the phase in which the attacker tries to gain control of the vulnerable application. On the other hand, the exploit includes the system calls which the attacker executes after gaining full control of the application.

In a typical buffer overflow exploit, the first step is to corrupt the data types and local variables, which gives the attacker control of the application. For example, in case of the original ftpd attack [5], the attacker achieves this by logging onto the ftpd server anonymously and issuing malformed commands such as `CWD ~{`. The actions taken by the attackers before they gain full control of the application are called the preamble. During the preamble phase, the application is still operational and the attacker does not have full control yet, hence the attacker may not be able to prevent the vulnerable application from generating anomalous behaviour.

After the attacker gains control of the application, the second step is to execute arbitrary code or a command to carry out a malicious action such as spawning a root shell or creating a super-user account. Commonly, this is achieved by injecting a shellcode. A shellcode is a short segment of an assembly program which aims to execute code on the vulnerable host. In the case of the original ftpd attack, the shellcode spawns a UNIX shell with super-user privileges and binds it to a port so that the attacker can login without supplying a password. Attackers can modify the exploit components fairly easily by changing the injected shellcode to evade detection. On the other hand, modifying the preamble requires finding an alternative way to take advantage of the vulnerability or finding another vulnerability, and therefore cannot be modified easily.

Although Wagner et al. [105] were aware of the preambles, they assumed that

the attacker could gain control of the vulnerable application silently (in their paper [105], ftpd application). Specifically, they state that "Moreover, we also assume that the attacker can silently take control of the application without being detected. This assumption is not always satisfied, but for many common attack vectors, the actual penetration leaves no trace in the system call trace. For instance, exploiting a buffer over-run vulnerability involves only a change in the control of the program, but does not itself cause any system calls to be invoked, and thus no syscall-based IDS can detect the buffer overrun itself." [105] Indeed, the results discussed in this thesis agree with the observation of Wagner et al. [105] that the attacker may not always be able to take control of the application silently. Furthermore, the analysis of the preamble in this thesis differs from the observations of Wagner et al. [105] and reveals that the actual penetration (i.e. the preamble) *does* leave anomalous system calls in the trace, at least in the cases of traceroute, restore, samba and ftpd applications.

This thesis proposes that a clear differentiation between the preamble and exploit is necessary since attackers can alter the system calls executed after they have *full control* whereas during the preamble phase, where the attacker prepares the vulnerable application for the buffer overflow, the interaction between the attacker and the application may inevitably cause anomalous system calls.

The boundary between the preamble and the exploit can be determined by locating the first action of the shellcode. The four original attacks [2] [4] [3] [5] employed in the analysis execute an $execve(`/bin/sh')$ system call to spawn a UNIX shell with super-user privileges. Any system call including and after $execve(`/bin/sh')$ is a result of the spawned UNIX shell whereas the system calls before $execve(`/bin/sh')$ are executed while the attacker was corrupting the data types and variables to deploy the exploit.

In previous work [105] [99] [96] [32] [56] [34], the attack was said to be optimal if the exploit component raised no alarms. However, even when the exploit raises no alarms, introducing the preamble can generate alarms for both the preamble itself and the transition between the preamble and the exploit. Therefore, the objective of the preamble analysis is to observe the source of the anomalies for the preamble and exploit components.

To do so, the original attacks for traceroute, samba, restore and ftpd were deployed against the five anomaly detectors, Section 4.2, and the anomaly rates for the preamble and exploit were investigated separately.

8.5.1 Preamble Analysis

Tables 8.29, 8.30, 8.31 and 8.32 detail the ratio of mismatches reported by the detectors for the preamble and exploit components separately for the original traceroute, restore, samba and ftpd attacks, respectively. As discussed in Section 4.2.1, a mismatch is recorded if the current observed behaviour does not match any behaviour in the 'normal database.' The anomaly rate is then calculated by dividing the number of mismatches by the total number of observations. The Neural Network detector employs the frequency of system calls and hence does not support temporal relationships explicitly. Therefore, the mismatch analysis is not applicable to the Neural Network detector in Tables 8.29, 8.30, 8.31 and 8.32.

Table 8.29: Ratio of mismatches for the original traceroute attack

	Preamble Ratio	**Exploit ratio**	**Total Count**
System Calls	15%	85%	344
Mismatches (Stide)	2.01%	97.99%	199
Mismatches (pH)	12.62%	87.38%	214
Mismatches (pHsm)	19.60%	80.40%	250
Mismatches (Markov Model)	5.30%	94.70%	132
Mismatches (Neural Network)	N/A	N/A	N/A

In total, the traceroute attack executed 344 system calls of which 15% belongs to the preamble component and 85% to the exploit. Against Stide, the attack as a whole (i.e. preamble and exploit combined) produced 199 mismatches 97.99% of which was generated by the exploit. Similar observations can be made for the mismatches recorded by the other detectors for traceroute. Therefore, in the case of the traceroute attack, an attacker can alter his exploit and reduce the anomaly rate substantially.

The restore attack executed 4454 system calls where 68% belongs to the exploit. Although the restore attack is longer than traceroute and the ratio of mismatches

Table 8.30: Ratio of mismatches for the original restore attack

	Preamble Ratio	**Exploit ratio**	**Total Count**
System Calls	32%	68%	4454
Mismatches (Stide)	30.83%	69.17%	3613
Mismatches (pH)	30.89%	69.11%	3881
Mismatches (pHsm)	32.21%	67.79%	4272
Mismatches (Markov Model)	26.50%	73.50%	1970
Mismatches (Neural Network)	N/A	N/A	N/A

Table 8.31: Ratio of mismatches for the original samba attack

	Preamble Ratio	**Exploit ratio**	**Total Count**
System Calls	88%	12%	4396
Mismatches (Stide)	31.64%	68.36%	433
Mismatches (pH)	55.62%	44.38%	694
Mismatches (pHsm)	47.22%	52.78%	989
Mismatches (Markov Model)	66.58%	33.42%	395
Mismatches (Neural Network)	N/A	N/A	N/A

Table 8.32: Ratio of mismatches for the original ftpd attack

	Preamble Ratio	**Exploit ratio**	**Total Count**
System Calls	86%	14%	3024
Mismatches (Stide)	73.20%	26.80%	679
Mismatches (pH)	75.20%	24.80%	762
Mismatches (pHsm)	62.84%	37.16%	592
Mismatches (Markov Model)	74.42%	25.58%	215
Mismatches (Neural Network)	N/A	N/A	N/A

from the preamble is higher than traceroute, the attackers can alter their exploit to reduce the anomaly rate.

On the other hand, the ftpd attack executed 3024 system calls, of which 86% belongs to the preamble component and 14% to the exploit. The attack as a whole produced 679 mismatches, 73.20% of which was generated by the preamble. Consequently, modifying the exploit would have less impact on the anomaly rate for ftpd. The samba attack exhibited similar properties, however the preamble produced a smaller proportion of the overall anomaly rate compared with the ftpd attack.

8.5.2 Discussion of the Preamble Analysis

The length of the preamble gains importance when determining the operational limits of the detectors. Specifically, if the preamble is short and if the attacker manages to modify his exploit accordingly (e.g. instead of spawning a root shell, creating a super-user account), the anomaly rate of the attack as a whole can be reduced substantially. However, if the preamble is long, there will be a higher likelihood of raising alarms no matter what type of exploit is being used [47].

Tables 8.29, 8.30, 8.31 and 8.32 indicate that the anomaly rate returned for the exploit alone does not represent the anomaly rate returned for the entire attack since the activities associated with gaining control of the application (preamble) raises alarms. Furthermore, the ratio of the preamble to the exploit and the anomaly rate from the preamble play an important role in the overall anomaly rate of an attack. It is evident that the anomaly rate of an attack can be reduced more effectively when the exploit is relatively longer than the preamble, even though the exploit itself raises some alarms.

Furthermore, in previous work, an attack was considered optimal if the exploit never generated any mismatches against the detector database. Anomaly detectors count mismatches between the candidate trace (in the case of an attack preamble plus exploit) and the sequences of normal behaviour in the detector database. That is to say, a sliding window comparison is made between the database and the candidate trace (i.e. preamble + exploit). Therefore, even though the exploit raises no alarms, introducing the preamble will return mismatches (alarms) for both the preamble itself

and at the transition between the preamble and the exploit. Thus, for a predefined preamble, the best that a mimicry attack can do is to minimize the contribution from the exploit and the transition from the preamble to the exploit. Furthermore, in the cases of pH and pHsm, the anomalies from the preamble can produce considerable delays and stop the attack before the exploit is deployed.

8.6 Discussion of Results

The experiments detailed in this chapter employ the proposed GP with a Pareto ranking approach to generate mimicry attacks against five anomaly detectors which monitor four vulnerable UNIX applications. The mimicry attacks are evolved not manually but automatically, in this case using a linear Genetic Programming-based approach. It is assumed that the attacker can obtain a copy of the detector and run it locally. Realistically, the detector will not be open source and/or the attacker will not know the exact configuration parameters of the detector at the target site. Therefore, the task of the attacker is to craft a mimicry attack that will evade the detector on the victim host based upon the feedback (anomaly rate, delay) obtained from the local copy of the detector. Such a perspective denotes a 'black-box' model of the detector, whereas previous research has relied on a 'white-box' assumption [105] [99] [96] [32] [56] [34]. In the proposed approach, emphasis is placed upon:

1. the identification of an appropriate set of system calls from which exploits are built, in this case informed by the most frequently executed instructions from the vulnerable application;

2. the identification of appropriate goals such as the minimization of the detector anomaly rate and delay while matching key steps in establishing the 'core' exploit;

3. the support for obfuscation, which – in this case – is a direct side effect of the stochastic search operators inherent in EC;

4. measurement of the anomaly rate not after the attacker gains control, but over the entire attack.

Results indicate that the framework is successful in reducing the anomaly rate of the attacks while utilizing only the anomaly rate and the delays which the detector reports without using privileged detector information. The degree of success depends upon various characteristics of the attack such as the preamble length, preamble anomaly rate and the exploit length. In other words, finding a zero anomaly rate exploit does not necessarily imply that the attack can evade detection completely.

Although the previous work was effective in generating exploits with zero anomaly rates, the results discussed in this chapter establish that the preamble component and the transition between the preamble and the exploit raises alarms, hence it is very difficult for an attacker to evade detection completely. Furthermore, the results indicate that for an ftpd exploit which raises no alarms the corresponding attack still produced a 10.62% anomaly rate.

Additionally, experiment results demonstrate that delay associated with locality frame counts is an effective way to stop an attack. Even though the attack achieves low anomaly rates, it can be frozen effectively if the anomalies are clustered together. In particular, mimicry attacks against samba have low anomaly rates yet the delays associated with them are in the billions of centuries. On the other hand, if the delays associated with locality frame count will be employed in the real-world, reducing the false positive for the detector deserves further attention since legitimate behaviour which is unknown to the detector can cause substantial delays as well.

The training sensitivity analysis indicates that the selection of appropriate training sets is crucial since it affects the resulting normal behaviour model. Furthermore the analysis results indicated that normal behaviour can vary substantially. In such a case, identifying a suitable threshold can be problematic since normal behaviour scenarios not included in the normal behaviour model of the detector can produce high anomaly rates which in turn makes it difficult to determine a detection threshold.

Mimicry attacks have various characteristics beyond anomaly rate which can affect their success. Even if an attack has a zero anomaly rate exploit, should the preamble be lengthy and produce several anomalies, it can be detected easily. Furthermore although both the preamble and the exploit raise very few alarms, if the anomalies are clustered together, they can cause an increase in the locality frame count, hence

'freezing' the attack effectively. Another method which an attacker can employ to minimize the anomaly rate of an attack with a detectable preamble is to employ a longer exploit which raises fewer or no alarms, hence injecting more normal behaviour to obfuscate the anomalies.

In the light of these observations, mimicry attack and vulnerability testing research should move from focusing on the anomaly rate alone to incorporating multiple characteristics such as the preamble and exploit length, locality frame counts and associated delays.

Chapter 9

Analysis of Mimicry Attacks

Chapter 8 and the relevant work reported the mimicry attack anomaly rates for the detectors against which they are trained. Although this provides a comparison between the anomaly rate of the original attack and the generated mimicry attack, it does not answer the question of what the anomaly rate would be if the same mimicry attack were deployed against a different detector. An analysis addressing this issue is provided in Section 9.1 and aims to identify whether mimicry attacks can generalize to other detectors or are specific to the detector against which they are trained.

Previous work [105] [99] [96] [32] [56] [34] assumed a 'white-box' access to the anomaly detectors, which implies that the attacker can use the normal database of the detector, the training sets and knowledge of the detection mechanism to facilitate the 'white-box' attack generation process. On the other hand, the 'black-box' approach taken in this thesis implies that the knowledge of the internal workings of the detector is 'hidden' from the attacker. Therefore, the attacker has to interact with the detector during its operation and utilize the feedback from the detector (in the form of anomaly rates) to facilitate the 'black-box' attack generation process. Given that the previous work assumes a 'white-box' approach and this thesis assumes a 'black-box' approach, Sections 9.2 and 9.3 compare and contrast the similarities and differences between the attacks generated by the 'white-box' and 'black-box' approaches.

The analysis in Section 9.2 aims to compare the 'white-box' mimicry attacks provided in the previous work with the 'black-box' mimicry attacks which were generated for this thesis. Given that the training sets utilized in previous work differ from the training sets utilized in this thesis, Section 9.3 expands the analysis by developing methods to generate 'white-box' attacks against each anomaly detector. The 'white-box' attacks generated in Section 9.3 can be compared to the 'black-box' attacks

detailed in Section 9.1.1, because they are generated against the same detector configurations.

9.1 Deploying a Mimicry Attack Against Numerous Detectors

In Chapter 8, the anomaly rates of the mimicry attacks were reported against the detector for which they were trained. Although such a deployment scenario is sufficient to make a comparison between the original and the mimicry attack, a natural extension is to deploy the mimicry attacks against other detectors. In such an extended scenario, it is important to make the distinction between a 'training' detector and a 'test' detector. The training detector is the anomaly detector with which the proposed approach interacted while generating the mimicry attacks. On the other hand, a test detector is another anomaly detector which was not employed during training. In this section, the objective of the analysis is to determine the anomaly rate of the GP-generated mimicry attacks against the test detectors. Such an analysis provides an insight into the success of the mimicry attack against a detector for which it is not trained. In other words, it answers the question: can a mimicry attack against a specific detector work against other detectors? Or, from a different perspective, are mimicry attacks detector specific or can they be applicable to various detectors (sharing similar monitoring and detection techniques)?

To facilitate the analysis, the best attacks determined for each application and detector scenario are deployed against the other detectors monitoring the same application. The analysis in Section 9.1.1 focuses on the anomaly rates whereas the analysis in Section 9.1.2 focuses on the delays.

9.1.1 Analysis of the Anomaly Rates

In Table 9.1, when the best mimicry attacks generated against Stide were tested against pH, pH with a schema mask (pHsm), Markov Model and Neural Network detectors, the anomaly rate of the exploits were fairly low – particularly for the restore and samba attacks. This indicates that exploits generated against Stide can generalize to other detectors very well. On the other hand, given that the Neural Network utilizes a different monitoring method (frequency of system calls as opposed

to sequence of system calls), the exploits generated against Stide and tested on the Neural Network detector do not produce low anomaly rates. Furthermore, anomaly rates of the corresponding attacks (preamble + exploit) are detailed in Table 9.2.

Table 9.1: Anomaly rates of the exploits generated against Stide, tested on the pH, pHsm, Markov Model and Neural Network detectors

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	16.67%	29.63%	81.25%	14.29%	37.24%
restore	0.40%	0.20%	0.81%	0.20%	2.73%
samba	0.50%	0.30%	0.20%	0.20%	57.15%
ftpd	57.14%	33.33%	100.00%	18.18%	34.71%

Table 9.2: Anomaly rates of the attacks generated against Stide, tested on the pH, pH with a schema mask, Markov Model and Neural Network detectors

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	10.96%	33.75%	75.36%	7.95%	15.89%
restore	46.25%	48.69%	59.51%	21.09%	6.44%
samba	3.00%	8.15%	7.41%	5.45%	9.48%
ftpd	19.30%	22.19%	38.21%	6.20%	2.69%

Table 9.3 details the exploit anomaly rates generated against pH. Again, the exploit anomaly rate of the exploits trained on pH are fairly low when they are tested against Stide, pH, pHsm and Markov Model detectors. Another interesting result is that the exploit anomaly rate of the restore attack against the Neural Network detector is 1.80%, which is lower than the exploit anomaly rate of the restore attack which was generated against the Neural Network detector (Table 8.13). This implies that the system call frequency distribution of the exploit trained on pH produces low anomaly rates against the Neural Network detector. Furthermore, Table 9.4 details the attack anomaly rates of the exploits in Table 9.3. The anomaly rates of the attacks generated against pH tested on the Markov Model detector are fairly close to the anomaly rates of the attacks generated against the Markov Model detector (Table 8.14).

Table 9.3: Anomaly rates of the exploits generated against pH, tested on the Stide, pHsm, Markov Model and Neural Network detectors

	Stide	***pH***	**pHsm**	**Markov Model**	**Neural Network**
traceroute	98.25%	11.71%	13.00%	10.92%	73.03%
restore	42.57%	0.10%	0.20%	0.20%	1.80%
samba	81.63%	0.10%	0.71%	0.30%	36.21%
ftpd	24.30%	0.10%	1.12%	0.20%	11.55%

Table 9.4: Anomaly rates of the attacks generated against pH, tested on the Stide, pHsm, Markov Model and Neural Network detectors

	Stide	***pH***	**pHsm**	**Markov Model**	**Neural Network**
traceroute	73.25%	18.29%	30.72%	8.14%	44.95%
restore	63.39%	48.57%	59.43%	21.05%	6.11%
samba	19.74%	8.11%	10.06%	5.47%	9.29%
ftpd	20.60%	16.11%	28.18%	4.50%	3.16%

Table 9.5 details the exploit anomaly rates of the best case attacks generated against pH with schema a mask (pHsm), whereas Table 9.6 provides the anomaly rates of the corresponding attacks. As opposed to the previous results, mimicry attacks generated against pHsm do not seem to generalize over to other detectors. Although the restore exploit produces low anomaly rates against pH and the Markov Model, the remaining exploit and attack anomaly rates are fairly high. This suggests that the evasion technique which GP employs against pHsm does not generalize well to the other detectors.

Table 9.5: Anomaly rates of the exploits generated against pHsm, tested on the Stide, pH, Markov Model and Neural Network detectors

	Stide	**pH**	***pHsm***	**Markov Model**	**Neural Network**
traceroute	100.00%	86.10%	86.15%	15.08%	100.00%
restore	74.37%	0.60%	11.01%	0.20%	16.12%
samba	99.50%	31.22%	29.23%	25.47%	40.35%
ftpd	100.00%	55.12%	38.73%	33.57%	14.38%

Table 9.6: Anomaly rates of the attacks generated against pHsm, tested on the Stide, pH, Markov Model and Neural Network detectors

	Stide	**pH**	***pHsm***	**Markov Model**	**Neural Network**
traceroute	96.15%	83.27%	85.12%	14.42%	100.00%
restore	76.46%	48.87%	63.67%	21.10%	9.55%
samba	23.40%	14.45%	15.84%	10.64%	11.76%
ftpd	41.39%	31.19%	38.43%	13.69%	3.64%

The analysis of the best case exploits generated against the Markov Model detector is provided in Table 9.7. Furthermore, the attack anomaly rates for the corresponding exploits are provided in Table 9.8. The anomaly rates indicate that mimicry attacks against the Markov Model do not generalize well to the other detectors. Given that a first order Markov Model can be described as a detector with a sliding window length of 2 (i.e. the current state depends only upon the immediately previous state) as opposed to Stide, pH and pHsm with sliding window lengths greater than or equal to 6, these results are not surprising.

Table 9.7: Anomaly rates of the exploits generated against the Markov Model detector, tested on the Stide, pH, pHsm and Neural Network detectors

	Stide	**pH**	**pHsm**	***Markov Model***	**Neural Network**
traceroute	100.00%	85.26%	84.56%	0.10%	100.00%
restore	99.30%	41.69%	51.93%	0.10%	32.06%
samba	100.00%	37.81%	28.50%	0.10%	43.39%
ftpd	100.00%	42.50%	26.07%	0.10%	12.50%

Furthermore, the exploit anomaly rates of the best case attacks generated against the Neural Network detector are provided in Table 9.9 and the corresponding attack anomaly rates are provided in Table 9.10. When the attacks generated against the Neural Network detector were deployed against other detectors, the resulting exploit and attack anomaly rates were fairly high. This could be attributed to two factors: (1) as opposed to utilizing system call sequence information, a Neural Network detector utilizes the frequency distribution of system calls which is a more compressed

Table 9.8: Anomaly rates of the attacks generated against the Markov Model detector, tested on the Stide, pH, pHsm and Neural Network detectors

	Stide	**pH**	**pHsm**	***Markov Model***	**Neural Network**
traceroute	95.98%	82.65%	83.97%	0.20%	100.00%
restore	86.63%	65.34%	79.92%	21.05%	14.27%
samba	23.24%	15.72%	15.63%	5.45%	7.40%
ftpd	41.48%	27.76%	35.01%	4.47%	3.92%

representation of system call sequences; (2) the detection methodology is different since the other approaches bases their decision upon system call sequences whereas the neural network approach base the decision upon the output of the model encapsulated by the neural network which describes the frequency distribution of system calls for 'normal behaviour.'

Table 9.9: Anomaly rates of the exploits generated against the Neural Network detector, tested on the Stide, pH, pHsm and Markov Model detectors

	Stide	**pH**	**pHsm**	**Markov Model**	***Neural Network***
traceroute	100.00%	99.50%	99.29%	75.62%	2.47%
restore	100.00%	97.28%	94.50%	58.54%	2.90%
samba	100.00%	87.11%	83.10%	51.25%	16.68%
ftpd	97.39%	61.83%	20.82%	30.07%	3.46%

Table 9.10: Anomaly rates of the attacks generated against the Neural Network detector, tested on the Stide, pH, pHsm and Markov Model detectors

	Stide	**pH**	**pHsm**	**Markov Model**	***Neural Network***
traceroute	96.15%	96.27%	97.97%	71.92%	1.63%
restore	86.88%	87.55%	97.04%	44.52%	5.60%
samba	23.53%	25.84%	26.72%	15.92%	5.77%
ftpd	40.76%	33.08%	41.62%	12.76%	1.26%

9.1.2 Analysis of the Delays

Table 9.11 details the exploit delays for the best case attacks generated against Stide, whereas Table 9.12 provides the attack delays for the corresponding exploits. The results indicate that the attacks generated against Stide produce lower delays when deployed against pH and pHsm, which is particularly apparent in the traceroute attack delays in Tables 9.11 and 9.12. This is a fairly interesting result since GP did not receive any delay feedback from Stide while evolving the attacks.

Table 9.11: Delays for the exploits generated against Stide

	pH	**pHsm**
traceroute	0.27	0.16
restore	9.99	17.81
samba	10.04	9.86
ftpd	0.03	0

Table 9.12: Delays for the attacks generated against Stide

	pH	**pHsm**
traceroute	0.8	0.69
restore	1.90E+38	2.16E+39
samba	7.95E+27	1.01E+21
ftpd	5.26E+30	4.59E+31

The exploit delays for the best case attacks generated against pH are provided in Table 9.13. Moreover, Table 9.14 provides the attack delays for the corresponding exploits. The results demonstrate that the exploit delays (Table 9.13) associated with the attacks generated against pH are fairly low, however the corresponding attack delays (Table 9.14) tend to be higher due to the anomalies in the preambles.

Table 9.15 details the exploit delays for the best case attacks against pHsm. As with the findings for the anomaly rate analysis in the previous section, the attacks generated against pHsm do not seem to generalize well to other detectors. The delays observed over the exploits in Table 9.15 are fairly high as opposed to the delay analysis

Table 9.13: Delays for the exploits generated against pH

	pH	**pHsm**
traceroute	1.11	1
restore	9.94	11.11
samba	9.94	14.02
ftpd	9.94	23.84

Table 9.14: Delays for the attacks generated against pH

	pH	**pHsm**
traceroute	6.39E+06	3.52E+11
restore	1.90E+38	2.17E+39
samba	7.95E+27	1.95E+28
ftpd	5.26E+30	4.59E+31

of the attacks generated against Stide and pH, Tables 9.11 and 9.13, respectively. Consequently the resulting attack delays are also high, as shown in Table 9.16.

Table 9.15: Delays for the exploits generated against pHsm

	pH	**pHsm**
traceroute	1.65E+35	1.65E+35
restore	15.08	3.94E+07
samba	3.88E+13	7.37E+12
ftpd	6.44E+23	9.86E+17

Table 9.17 shows the exploit delays of the best case attacks against Markov Model detectors whereas Table 9.18 shows the attack delays corresponding to these exploits. As with the findings of the anomaly rate analysis in the previous section, as far as the delays are concerned, the attacks generated against Markov Model detectors do not seem to generalize well to pH and pHsm detectors. As discussed in the anomaly rate analysis in Section 9.1.1, given that the Markov Model utilizes only the last system call as opposed to pH and pHsm utilizing the last 6 or more system calls, this is not surprising.

Table 9.16: Delays for the attacks generated against pHsm

	pH	**pHsm**
traceroute	1.65E+35	1.65E+35
restore	1.90E+38	2.16E+39
samba	7.95E+27	1.95E+28
ftpd	5.26E+30	4.59E+31

Table 9.17: Delays for the exploits generated against the Markov Model detector

	pH	**pHsm**
traceroute	3.17E+34	1.10E+34
restore	6.29E+18	6.52E+23
samba	2.02E+19	2.00E+14
ftpd	9.06E+18	7.52E+12

Table 9.18: Delays for the attacks generated against the Markov Model detector

	pH	**pHsm**
traceroute	3.17E+34	1.10E+34
restore	1.90E+38	2.16E+39
samba	7.95E+27	1.95E+28
ftpd	5.26E+30	4.59E+31

Table 9.19 details the exploit delays of the best case attacks generated against the Neural Network detector and Table 9.20 details the attack delays of the corresponding exploits. In terms of delays, as with the attacks generated against Markov Model detectors, the attacks generated against Neural Network detectors do not generalize well to other detectors. As discussed in the anomaly rate analysis in Section 9.1.1, this can be attributed to the fact that a Neural Network utilizes a different (compressed) representation of the system call sequences and to the fact that it utilizes a different detection methodology based upon the model which describes the frequency distribution of system calls.

Table 9.19: Delays for the exploits generated against the Neural Network detector

	pH	**pHsm**
traceroute	2.22E+39	1.86E+39
restore	5.04E+38	5.66E+37
samba	9.26E+35	9.19E+34
ftpd	1.84E+28	3.46E+22

Table 9.20: Delays for the attacks generated against the Neural Network detector

	pH	**pHsm**
traceroute	2.23E+39	1.88E+39
restore	7.01E+38	2.41E+39
samba	9.48E+35	1.13E+35
ftpd	5.58E+30	4.60E+31

9.1.3 Discussion of the Analysis Results

In this analysis, the best attacks, which produced the smallest anomaly rates, against each detector configuration were deployed against the other test detector configurations. The purpose of such analysis is to determine whether mimicry attacks are sensitive to changes in the target detector.

The results show that the attacks generated against Stide have a certain level of success in being applicable to different detector configurations. This may be attributed to the overspecialization of attacks when they are trained against a detector with longer sliding window sizes or additional delay constraints. Furthermore, incorporating a schema mask into pH not only improves its resistance against mimicry attacks but also reduces the applicability of the mimicry attacks to other detector configurations.

The analysis results establish that the applicability of the mimicry attacks diminish when they are deployed against detectors which employ different detection methodologies. This was apparent particularly when the attacks generated against Markov Model and Neural Network detectors were deployed against the remaining detector configurations. A future direction to the approach proposed in this thesis would be to employ GP with Pareto ranking to evolve attacks which aim to reduce the anomaly rate from not only a single detector but all five detectors.

9.2 Comparing with Mimicry Attacks in Previous Work

The main purpose of comparing the mimicry attacks provided in the relevant work with the ones which the GP approach produced in Chapter 8 is to determine the similarities and differences between the attacks provided in the previous work [34] [105] which assume 'white-box' access to the detector and the GP approach, which assumes a 'black-box' access. Therefore, the analysis in this section focuses on how the search methodology affects the resulting attacks.

Giffin et al. [34] published the mimicry attack which they generated against Stide on the traceroute application. Similarly, Wagner et al. [105] published the mimicry attack which they generated on the ftpd application against pH. Although the attack was generated against pH, the window size was set to 6 and the delay which pH enforces on system calls was not employed in their experiments, therefore their pH configuration works effectively like Stide. Both methodologies were 'white-box,' which implies that they have full access to the internal knowledge of the detector such as the normal database.

Additionally, the attacks on the traceroute and ftpd applications that produced

the minimum anomaly rates from the experiments in Chapter 8 were employed. For each application, there exists five GP-generated attacks in which each minimizes the anomaly rate for a detector (namely Stide, pH, pHsm, Markov Model and Neural Network). Finally, the original attack obtained from the SecurityFocus website was included in the experiments increasing the total attack count in the analysis to seven for each application.

It is important to note that the 'white-box' attacks on traceroute [34] and ftpd [105] were generated against detector configurations which were different from the detector configurations employed in this thesis in terms of the training sets. Therefore, the purpose of this analysis is not to make a direct comparison of the anomaly rates or delays but to investigate the differences and similarities between 'white-box' and 'black-box' mimicry attacks.

9.2.1 Comparison with the ftpd Mimicry Attack [105]

The ftpd attack published by Wagner et al. [105] against Stide is provided in Figure 9.1. The ftpd attacks which GP generated against Stide, pH, pHsm, the Markov Model and the Neural Network are provided in Figures B.4, B.9, B.15, B.23 and B.32, respectively. Furthermore, the exploit lengths are detailed in Table 9.21. It is apparent that an attack generated by the 'white-box' approach is shorter than the attacks generated by the 'black-box' GP approach. Given that the 'white-box' exhaustive search detailed by Wagner et al. [105] searches for the existence of an attack sequence starting in a depth-first manner without any consideration of the attack lengths, the search ends when an attack sequence is found. In other words, the exhaustive search employed would favour shorter attacks. On the other hand, the code bloat property of GP implies the existence of system calls which do not affect the success of the attack (i.e. effective NoOPs), hence the attacks tend to be longer, Table 9.21. This can be an advantage particularly if the preamble is long and anomalous. That is to say, a longer exploit would allow the anomaly rate of the attack to decrease over time, hence improving the chances of evading detection whereas short exploits (even though they raise no alarms) may not succeed in minimizing the anomaly rate of the attack.

```
read write close munmap sigprocmask wait4 sigprocmask sigaction alarm time stat read alarm
sigprocmask setreuid fstat getpid time write time getpid sigaction socketcall sigaction
close flock getpid lseek read kill lseek flock sigaction alarm time stat write open fstat
mmap read open fstat mmap read close munmap brk fcntl setregid open fcntl chroot chdir
setreuid lstat lstat lstat lstat open fcntl fstat lseek getdents fcntl fstat lseek getdents
close write time open fstat mmap read close munmap brk fcntl setregid open fcntl chroot
chdir setreuid lstat lstat lstat lstat open fcntl brk fstat lseek getdents lseek getdents
time stat write time open getpid sigaction socketcall sigaction umask sigaction alarm time
stat read alarm getrlimit pipe fork fcntl fstat mmap lseek close brk time getpid sigaction
socketcall sigaction chdir sigaction sigaction write munmap munmap munmap exit
```

Figure 9.1: The ftpd mimicry attack by Wagner et al. [105]

Table 9.21: Exploit lengths of the best ftpd mimicry exploits compared with the mimicry exploit provided by Wagner et al. [105] and the original ftpd exploit [5]

	Exploit Length
ftpd mimicry attack (Wagner et al. [105]) trained on Stide	135
ftpd attack (GP) trained on Stide	11
ftpd attack (GP) trained on pH	1000
ftpd attack (GP) trained on pHsm	994
ftpd attack (GP) trained on the Markov Model	1000
ftpd attack (GP) trained on the Neural Network	1000
original ftpd attack [5]	423

Table 9.22 details the attack anomaly rates obtained from deploying the 'white-box' attack of Wagner et al. [105] and the GP-generated 'black-box' attacks against the five detector configurations employed in the experiments for this thesis. The anomaly rate for the original attack is detailed in Table 9.22 as well. Anomaly rates for the 'white-box' attack are comparable to the anomaly rates produced by the 'black-box' attacks, especially the attacks generated against Stide and pH. The anomaly rate of the 'black-box' attack generated against Stide produced low anomaly rates since the attacks were generated against the detector configuration which is employed in this analysis as well. Given that the ftpd preamble is long and anomalous (Table 8.24), the resulting attacks (i.e. preamble + exploit) produce anomalies.

Table 9.22: Anomaly rates for the best ftpd mimicry attacks compared with the mimicry attack provided by Wagner et al. [105] and the original ftpd attack [5]

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
ftpd mimicry attack (Wagner et al. [105]) trained on Stide	22.81%	25.52%	40.97%	8.38%	3.14%
ftpd attack (GP) trained on Stide	19.30%	22.19%	38.21%	6.20%	2.69%
ftpd attack (GP) trained on pH	20.60%	16.11%	28.18%	4.50%	3.16%
ftpd attack (GP) trained on pHsm	41.39%	31.19%	38.43%	13.69%	3.64%
ftpd attack (GP) trained on the Markov Model	41.48%	27.76%	35.01%	4.47%	3.92%
ftpd attack (GP) trained on the Neural Network	40.76%	33.08%	41.62%	12.76%	1.26%
original ftpd attack [5]	22.78%	25.54%	20.27%	7.15%	6.91%

The exploit anomaly rates of the above-mentioned attacks are provided in Table 9.23. Since the attack by Wagner et al. [105] was generated against a different configuration of Stide, the anomaly rate for the exploit is fairly high compared with the 'black-box' attack generated against Stide. This supports the argument that the training set selection is important in generating mimicry attacks [46].

Table 9.23: Anomaly rates for the best ftpd mimicry exploits compared with the mimicry exploit provided by Wagner et al. [105] and the original ftpd exploit [5]

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
ftpd mimicry attack (Wagner et al. [105]) trained on Stide	95.38%	94.49%	99.14%	51.85%	53.06%
ftpd attack (GP) trained on Stide	57.14%	33.33%	100.00%	18.18%	34.71%
ftpd attack (GP) trained on pH	24.30%	0.10%	1.12%	0.20%	11.55%
ftpd attack (GP) trained on pHsm	100.00%	55.12%	38.73%	33.57%	14.38%
ftpd attack (GP) trained on the Markov Model	100.00%	42.50%	26.07%	0.10%	12.50%
ftpd attack (GP) trained on the Neural Network	97.39%	61.83%	20.82%	30.07%	3.46%
original ftpd attack [5]	47.52%	47.85%	57.29%	13.65%	18.86%

The attack delays associated with the 'white-box' attack by Wagner et al [105], GP-generated 'black-box' attacks and the original ftpd attack are provided in Table 9.24. Similarly, the exploit delays for these attacks are detailed in Table 9.25. The results show that, in terms of the attack delay, the 'white-box' attack produces delays comparable to the delays of the 'black-box' attacks generated against Stide and pH. In this case, the fact that the 'white-box' exploit is shorter provides an advantage in minimizing the delays. On the other hand, the exploit generated against pH is fairly long (1000 system calls) but succeeds in minimizing the exploit delay below 10 seconds (Table 9.25).

9.2.2 Comparison with the traceroute Mimicry Attack [34]

Figure 9.2 provides the traceroute attack published by Giffin et al. [34]. The GP-generated traceroute attacks against Stide, pH, pHsm, Markov Model and Neural network detectors are provided in Figures B.1, B.5, B.10, B.17 and B.25, respectively.

Table 9.24: Delays for the best ftpd mimicry attacks compared with the mimicry attack provided by Wagner et al. [105] and the original ftpd attack [5]

	pH	**pHsm**
ftpd mimicry attack (Wagner et al. [105]) trained on Stide	3.93E+35	2.50E+37
ftpd attack (GP) trained on Stide	5.26E+30	4.59E+31
ftpd attack (GP) trained on pH	5.26E+30	4.59E+31
ftpd attack (GP) trained on pHsm	5.26E+30	4.59E+31
ftpd attack (GP) trained on the Markov Model	5.26E+30	4.59E+31
ftpd attack (GP) trained on the Neural Network	5.58E+30	4.60E+31
original ftpd attack [5]	5.26E+30	4.89E+25

Table 9.25: Delays for the best ftpd mimicry exploits compared with the mimicry exploit provided by Wagner et al. [105] and the original ftpd exploit [5]

	pH	**pHsm**
ftpd mimicry attack (Wagner et al. [105]) trained on Stide	1.27	1.16
ftpd attack (GP) trained on Stide	0.03	0
ftpd attack (GP) trained on pH	9.94	23.84
ftpd attack (GP) trained on pHsm	6.44E+23	9.86E+17
ftpd attack (GP) trained on the Markov Model	9.06E+18	7.52E+12
ftpd attack (GP) trained on the Neural Network	1.84E+28	3.46E+22
original ftpd attack [5]	3.78E+22	4.89E+25

Additionally, the exploit lengths are detailed in Table 9.26. Among the attacks utilized in this analysis, the 'white-box' attack by Giffin et al. [34] has the shortest length, which suggests that the search method employed by Giffin et al. [34] may favour short attacks. Furthermore, the GP-generated attacks against Stide and pH are fairly short as well, Table 9.26.

```
close munmap open fcntl64 fcntl64 fstat64 mmap2 read close munmap write
```

Figure 9.2: The traceroute mimicry attack by Giffin et al. [34]

Table 9.26: Exploit length for the best traceroute mimicry exploits compared with the mimicry exploit provided by Giffin et al. [34] and the original traceroute exploit [2]

	Exploit Length
traceroute mimicry attack (Giffin et al. [34]) trained on Stide	11
traceroute attack (GP) trained on Stide	34
traceroute attack (GP) trained on pH	118
traceroute attack (GP) trained on pHsm	1000
traceroute attack (GP) trained on the Markov Model	957
traceroute attack (GP) trained on the Neural Network	1000
original traceroute attack [2]	261

The anomaly rates for the 'white-box,' 'black-box' and the original attacks are detailed in Table 9.27. The anomaly rate for the corresponding exploits are detailed in Table 9.28. Generally, the anomaly rates are minimized only for the detector for which the attack was trained. For example, in terms of the anomaly rates for the attacks against Stide, the two attacks which produced the lowest anomaly rates were (1) the 'black-box' attack trained on Stide, with a 10.96% anomaly rate and (2) the 'white-box' attack trained on Stide with a 20.00% anomaly rate. This indicates that the traceroute application is sensitive to the training set and detector configurations. Similarly, the anomaly rates for the exploits in Table 9.28 indicate that the anomaly rate of the exploit is sensitive to the detector against which the attack is trained. This suggests that the mimicry attacks against traceroute are not as widely applicable as

the mimicry attacks against ftpd and attackers may need to craft different exploits for different target detectors.

Table 9.27: Anomaly rates for the best traceroute mimicry attacks compared with the mimicry attack provided by Giffin et al. [34] and the original traceroute attack [2]

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute mimicry attack (Giffin et al. [34]) trained on Stide	20.00%	36.84%	67.39%	6.15%	100.00%
traceroute attack (GP) trained on Stide	10.96%	33.75%	75.36%	7.95%	15.89%
traceroute attack (GP) trained on pH	73.25%	18.29%	30.72%	8.14%	44.95%
traceroute attack (GP) trained on pHsm	96.15%	83.27%	85.12%	14.42%	100.00%
traceroute attack (GP) trained on the Markov Model	95.98%	82.65%	83.97%	0.20%	100.00%
traceroute attack (GP) trained on the Neural Network	96.15%	96.27%	97.97%	71.92%	1.63%
original traceroute attack	61.26%	66.27%	81.79%	38.78%	31.19%

The delays associated with the traceroute attacks are given in Table 9.29, whereas Table 9.30 details the delays with the corresponding exploits. The 'white-box' attack against Stide and the 'black-box' attacks against Stide and pH produced the least amount of delays both in terms of the attack and exploit delay. Furthermore, the delay analysis shows that the 'white-box' attack produced the fewest delays both for the attack and the exploit. Although the anomaly rate for the 'white-box' exploit was fairly high (Table 9.28), the short delay for the 'white-box' attack could be attributed to the short attack length (Table 9.26).

Table 9.28: Anomaly rates for the best traceroute mimicry exploits compared with the mimicry exploit provided by Giffin et al. [34] and the original traceroute exploit [2]

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute mimicry attack (Giffin et al. [34]) trained on Stide	100.00%	100.00%	100.00%	16.67%	100.00%
traceroute attack (GP) trained on Stide	16.67%	29.63%	81.25%	14.29%	37.24%
traceroute attack (GP) trained on pH	98.25%	11.71%	13.00%	10.92%	73.03%
traceroute attack (GP) trained on pHsm	100.00%	86.10%	86.15%	15.08%	100.00%
traceroute attack (GP) trained on the Markov Model	100.00%	85.26%	84.56%	0.10%	100.00%
traceroute attack (GP) trained on the Neural Network	100.00%	99.50%	99.29%	75.62%	2.47%
original traceroute attack	71.48%	73.91%	83.06%	47.89%	70.21%

Table 9.29: Delays for the best traceroute mimicry attacks compared with the mimicry attack provided by Giffin et al. [34] and the original traceroute attack [2]

	pH	**pHsm**
traceroute mimicry attack (Giffin et al. [34]) trained on Stide	0.57	0.46
traceroute attack (GP) trained on Stide	0.8	0.69
traceroute attack (GP) trained on pH	6.39E+06	3.52E+11
traceroute attack (GP) trained on pHsm	1.65E+35	1.65E+35
traceroute attack (GP) trained on the Markov Model	3.17E+34	1.10E+34
traceroute attack (GP) trained on the Neural Network	2.23E+39	1.88E+39
original traceroute attack	4.39E+35	8.51E+35

Table 9.30: Delays for the best traceroute mimicry exploits compared with the mimicry exploit provided by Giffin et al. [34] and the original traceroute exploit [2]

	pH	**pHsm**
traceroute mimicry attack (Giffin et al. [34]) trained on Stide	0.04	0
traceroute attack (GP) trained on Stide	0.27	0.16
traceroute attack (GP) trained on pH	1.11	1
traceroute attack (GP) trained on pHsm	1.65E+35	1.65E+35
traceroute attack (GP) trained on the Markov Model	3.17E+34	1.10E+34
traceroute attack (GP) trained on the Neural Network	2.22E+39	1.86E+39
original traceroute attack	4.39E+35	8.51E+35

9.2.3 Discussion of the Analysis Results

In this analysis, a comparison of the published 'white-box' attacks [105] [34] was made with the GP-generated 'black-box' attacks. Each attack was deployed against different detector configurations and the results were provided in terms of anomaly rates, delays and exploit lengths. Mimicry attacks generated by GP provide anomaly rates comparable to the anomaly rates of 'white-box' attacks although the access to the detector is limited to the anomaly rate alone. Such a 'black-box' assumption is particularly suitable if the attacker does not posses internal knowledge of the detector.

The analysis revealed that mimicry attacks are sensitive to detector configurations and to the target detector upon which they were trained. Therefore, the 0% exploits reported by Wagner et al. [105] and Giffin et al. [34] produced fairly high anomaly rates in this analysis. Furthermore, among the GP-generated 'black-box' attacks, the results indicated that the anomaly rate for the exploit can vary substantially depending upon the detector configuration against which the attack is deployed.

The search methodology employed in 'white-box' approaches involve an exhaustive search on the normal database of the detector until an attack is found, therefore the attack lengths tend to be shorter than the length of the attacks generated by the 'black-box' GP approach. Although this may provide some advantage in reducing delay, it is a disadvantage since a longer exploit can help in minimizing the anomaly rate of the attack, particularly if the preamble is long and anomalous.

In terms of investigating the difficulty of deploying mimicry attacks, the results indicate that the attacks are fairly sensitive to detector or training set changes [46]. The findings of the comparison experiments support and expand upon the findings of the analysis provided in Section 9.1.

The 'white-box' mimicry attacks [105] [34] employed in this section were not trained on the detector configurations employed in this thesis. Therefore, Section 9.3 continues the analysis by developing 'white-box' attacks against the anomaly detectors employed in this thesis. Furthermore, the methodologies for generating 'white-box' attacks are discussed as well with the purpose of demonstrating that 'white-box' approaches do not generalize over numerous detectors in terms of applicability. In

other words, for each detector to be tested, the attacker needs to develop a 'white-box' methodology which is suitable for the internal knowledge and data structures of the detector.

9.3 Comparison of 'White-Box' Attacks

As discussed in Section 8, the approach taken in this thesis to generate mimicry attacks is a 'black-box' one in which the attacker has limited access to the detector. Conversely, previous work on mimicry attack generation [105] [99], [96] [32] [56] [34] assumed that the access to the detector was 'white-box,' which implies that the attacker can make use of the internal data structures of the detector or any other knowledge regarding the operation of the detector.

The analysis in Section 9.2, compared the attacks generated with the previous 'white-box' approaches. However, as discussed in Section 9.2, since the training sets utilized to generate the 'white-box' attacks [105] [34] differ from the training sets utilized in this work, a direct comparison could not be made. The analysis in this section implements 'white-box' approaches to generate mimicry attacks against the anomaly detectors utilized in this thesis in order to make a direct comparison between the two approaches.

The 'black-box' methodology proposed in this thesis utilizes the feedback from the detector without using any internal knowledge. This implies that it can be applied to numerous anomaly detectors as long as the detector provides a feedback such as anomaly rate and delay. Conversely, the 'white-box' assumption requires a focused and exhaustive search against the detection mechanism and can employ internal knowledge such as the training data, normal database, or the detection methodology utilized. Therefore, for each detector, the attack generation methodology needs to be revised to fit the characteristics and constraints of the detection mechanism. For example, mimicry attacks generated against a detector which monitors the frequency distribution of system calls may not succeed against a detector which monitors not only the frequencies but also the sequences of system calls.

To this end, the 'white-box' mimicry attack generation methodologies implemented against each anomaly detector are discussed separately in the following subsections. As with the comparison experiments in Section 9.1, each mimicry attack is tested against not only the anomaly detector for which it was created, but also against the remaining anomaly detectors.

The exploit lengths of the 'white-box' exploits are detailed in Table 9.31, which can be compared with the exploit lengths of the 'black-box' exploits detailed in Table 8.20.

Table 9.31: Lengths of the exploits generated against five anomaly detectors in terms of system calls

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	77	38	158	97	736
restore	92	60	60	95	167207
samba	213	91	90	124	65648
ftpd	135	43	54	106	334252

9.3.1 The 'White-Box' Attacks Against Stide

Wagner et al. [105] employed language theory to formulate two sets: a set of malicious sequences and a set of normal behaviour sequences. If the intersection of these sets is not an empty set, it implies that a mimicry attack can be constructed. On the other hand, Tan et al. [96] viewed the problem as increasing the length of the malicious sequence beyond the sliding window length, which means that the sliding window patterns generated on the malicious sequence would be within normal behaviour. The common trait of both methods is to generate a mimicry attack which does not contain a sequence which will create an anomaly. In other words, the sliding window patterns which the mimicry attack produces should be in the normal database of the detector.

Sharing this goal, the 'white-box' mimicry attack generation developed for Stide in this work takes the Stide normal database (i.e. the sequences of system calls) and builds a connectivity graph where each pattern in the normal database is represented

as a node. A connection between nodes A and B means that node (i.e. sliding window pattern) B can follow node (i.e. sliding window pattern) A. The search was implemented as a depth-first graph search where the search terminates when the resulting pattern contains the malicious system calls (in this case open - write - close).

Results

The anomaly rates for the exploits generated using this 'white-box' approach are detailed in Table 9.32, whereas the anomaly rates for the corresponding attacks are provided in Table 9.33. Clearly, exploits achieve a 0% exploit anomaly rate against the detector upon which they were trained. Furthermore, the restore exploit achieved a 0% exploit anomaly rate against all the anomaly detectors except the Neural Network detector. This suggests that the exploit employs a sequence which exists in the normal data structure of Stide, pH, pHsm and the Markov Model. Since the Neural Network employs system call frequency distributions, the 100% anomaly rate can be attributed to the disparity in the system call frequency distribution between the exploit and the normal behaviour database of the Neural Network detector.

Table 9.32: Anomaly rates for the exploits generated against Stide by using the 'white-box' approach, tested against the five anomaly detectors utilized in this work

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	0.00%	5.80%	18.97%	0.00%	100.00%
restore	0.00%	0.00%	0.00%	0.00%	100.00%
samba	0.00%	4.39%	19.07%	0.00%	100.00%
ftpd	0.00%	7.87%	14.66%	0.00%	100.00%

Although the exploit produced no anomalies, it is important to emphasize that the anomaly rates of the overall attacks produced anomalies and are comparable to the anomaly rates of the attacks generated by the 'black-box' GP approach (Table 9.2). In the case of the restore attack, although the exploit raised no alarms, the resulting attack produced anomaly rates greater than the attack generated by the 'black-box' GP approach, Table 9.2. This is due to the fact that 'white-box' attacks are generally more concise than 'black-box' attacks generated by GP, Table 9.31.

Table 9.33: Anomaly rates for the attacks generated against Stide by using the 'white-box' approach, tested against the five anomaly detectors utilized in this work

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	2.61%	18.85%	36.04%	1.54%	100.00%
restore	73.46%	76.78%	86.23%	33.10%	100.00%
samba	3.51%	9.87%	9.70%	6.45%	100.00%
ftpd	18.29%	21.51%	14.73%	5.84%	100.00%

The delays associated with the exploits generated by the 'white-box' methodology are detailed in Table 9.34. Similarly, the delays associated with the 'white-box' attacks are provided in Table 9.35. The delays show that although the exploits do not produce substantial delays, the anomalies in the preambles increase the delays substantially except for the traceroute application where the preamble is short.

Table 9.34: Delays for the exploits generated against Stide by using the 'white-box' approach

	pH	**pHsm**
traceroute	0.69	0.58
restore	0.84	0.73
samba	246.86	5.00E+10
ftpd	1.27	1.16

Table 9.35: Delays for the attacks generated against Stide by using the 'white-box' approach

	pH	**pHsm**
traceroute	1.22	1.11
restore	1.90E+38	4.33E+38
samba	7.95E+27	1.01E+21
ftpd	5.26E+30	7.84E+17

9.3.2 The 'White-Box' Attacks Against pH

Wagner et al. [105] employed their 'white-box' mimicry attack generation methodology against pH but the way pH was utilized was the same as for Stide (i.e. a sliding window of 6 was employed). pH stored the patterns in the form of look-ahead pairs, which means the sliding window patterns were not stored directly, but in a compressed form. As Inoue et al. [38] discussed, the database format may make pH more susceptible to mimicry attacks than Stide. As a simple example, consider that both Stide and pH are being trained on the sequence in Table 9.36, where the right hand side of the sequence is the more recent.

Table 9.36: An example sequence for which Stide and pH are trained

2	2	1	3	2	2	1

Given that the sliding window length of both detectors for 4, the Stide normal database stores the sliding window patterns encountered in the training set, Table 9.37. While generating attacks against this Stide configuration, there are four sliding window patterns (or length 4) which will not generate any anomalies.

Table 9.37: The Stide normal database which was trained on the sequence provided in Table 9.36

	Current	**Position 1**	**Position 2**	**Position 3**
pattern 1	1	2	2	3
pattern 2	2	2	3	1
pattern 3	2	3	1	2
pattern 4	3	1	2	2

The pH normal database – although originally stored as look-ahead pairs – can be represented as a table as well. The main difference between the Stide and pH normal databases is that sequences 2 2 3 1 and 2 3 1 2 are compressed into one row in Table 9.38.

An exhaustive search of the pH normal database reveals that, given the same training set and sliding window lengths, pH allows more sequences than Stide does,

Table 9.38: The pH normal database, which was trained on the sequence provided in Table 9.36

Current	**Position 1**	**Position 2**	**Position 3**
1	{2}	{2}	{3}
2	{2, 3}	{3, 1}	{1, 2}
3	{1}	{2}	{2}

Figure 9.3. This condition is likely to be magnified when longer sliding windows are employed. In Figure 9.3, the sequences which Stide allows are printed in bold. Although pH produces a larger set from which the mimicry attacks are built, it also expands the search space which the 'white-box' methodology needs to cover. Therefore, this attribute can work in attacker's favour (by providing a more extensive normal database) or to his/her disadvantage (by increasing the computational cost of the search). This research, however, does not focus on the impacts of this situation on 'white-box' methodologies. The main purpose of the analysis in this section is to generate 'white-box' attacks to compare against the 'black-box' attacks which are generated by the approach proposed in this thesis.

1	**2**	**2**	**3**	2	3	3	1
2	**2**	**3**	**1**	2	3	3	2
2	2	3	2	2	3	1	1
2	2	1	1	**2**	**3**	**1**	**2**
2	2	1	2	**3**	**1**	**2**	**2**

Figure 9.3: The list of sequences of length 4 permitted by the pH normal database

As with the methodology proposed by Wagner et al. [105], the methodology developed against pH in this thesis exploits the use of sliding windows in pH and builds attacks from the sliding window patterns encountered during training. Simply put, the approach developed by Stide in Section 9.3.1 is utilized with a sliding window length of 9. This is an acceptable approach since a 'white-box' approach implies that the attacker can utilize all and any knowledge about the detector, including the detection methodology. This reduces the search space size of a 'white-box' search, hence shortening the time it takes to find an attack.

Results

The anomaly rates for the exploits generated by the 'white-box' approach are provided in Table 9.39 and the anomaly rates for the corresponding attacks are detailed in Table 9.40. As with the results against Stide, the 'white-box' exploits produced a 0% anomaly rate against the detector for which they were trained. Furthermore, when the exploits were tested against other anomaly detectors, the anomaly rate remained low. Therefore, the exploits generated against pH generalize well to other detectors (with the exception of Neural Network detectors which employ system call frequency distributions).

Table 9.39: Anomaly rates for the exploits generated against pH by using the 'white-box' approach, tested against the five anomaly detectors utilized in this work

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	0.00%	0.00%	5.26%	0.00%	100.00%
restore	0.00%	0.00%	0.00%	0.00%	100.00%
samba	0.00%	0.00%	4.17%	0.00%	100.00%
ftpd	0.00%	0.00%	0.00%	0.00%	100.00%

Compared with the 'black-box' attacks generated by GP in Table 9.4 , the attack anomaly rates for the 'white-box' attacks in Table 9.40, perform better against numerous detector configurations (e.g. the traceroute attack against Stide) but the 'black-box' attacks perform better against other configurations such as the restore attack against all anomaly detectors. This shows that although the 'white-box' approach produces lower anomaly rates on exploits, when the anomaly rates for the attacks are calculated, 'black-box' attacks produce comparable or better anomaly rates by utilizing more system calls to hide the true intention of the attack.

The delays associated with the exploits are detailed in Table 9.41 whereas the delays associated with the corresponding attacks are detailed in Table 9.42. As with the delays reported for the 'white-box' attacks generated against Stide, the exploit delays are fairly short (less than one second in all cases) but the corresponding attack delays are substantially long due to the length and anomalous preambles, with the exception of traceroute.

Table 9.40: Anomaly rates for the attacks generated against pH by using the 'white-box' approach, tested against the five anomaly detectors utilized in this work

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	3.95%	22.89%	43.06%	2.20%	100.00%
restore	75.03%	78.31%	87.97%	33.79%	100.00%
samba	3.59%	9.94%	9.14%	6.65%	100.00%
ftpd	18.92%	21.88%	14.49%	6.05%	100.00%

Table 9.41: Delays for the exploits generated against pH by using the 'white-box' approach

	pH	**pHsm**
traceroute	0	0.19
restore	0	0
samba	0	0.72
ftpd	0	0

Table 9.42: Delays for the attacks generated against pH by using the 'white-box' approach

	pH	**pHsm**
traceroute	0.83	0.72
restore	1.90E+38	4.33E+38
samba	7.95E+27	1.01E+21
ftpd	5.26E+30	7.84E+17

9.3.3 The 'White-Box' Attacks Against pHsm

Although, as of this writing, pHsm has not been employed in previous mimicry attack research, the methodology discussed in Section 9.3.1 applies to pHsm. Two attributes of pHsm differentiate it from Stide and pH: (1) the sliding window length is 20 and (2) a schema mask is applied to the sliding window. However, given a 'white-box' access to the detector, it is realistic to assume that the attacker knows both the sliding window length and the schema mask. Thus, the task can be formulated as generating a mimicry attack against a detector with a sliding window of 20 system calls. This implies that the methodology detailed in Section 9.3.1 is utilized with a sliding window length of 20. If the attack does not raise any alarms with this condition, the application of a schema mask will not generate any anomalies since the purpose of the schema mask is to select 9 values from a sliding window of length 20.

Results

The anomaly rates for the exploits generated by the 'white-box' approach against pHsm are detailed in Table 9.43 and the anomaly rates for the corresponding 'white-box' attacks are detailed in Table 9.44. Compared with the 'white-box' exploit anomaly rates against Stide and pH (Tables 9.32 and 9.39, respectively), the 'white-box' exploits generated against pHsm produced 0% exploit anomaly rates for all the anomaly detectors, except the Neural Network detector. A possible reason for the 0% anomaly rates is the fact that pHsm utilizes longer sliding window lengths for detection, therefore, if the attack can evade pHsm, it can also evade other detectors with shorter sliding window lengths. This also implies that from an attacker's perspective, overestimating the sliding window length of the detector is better than underestimating it since an attack which evades a detector with a longer sliding window length is likely to evade the detectors with shorter sliding window lengths, but the opposite may not hold true (as discussed in the next section).

A comparison of attack anomaly rates between the attacks generated by the 'white-box' method (Table 9.44) and 'black-box' method (Table 9.6) shows that the 'white-box' approach produces lower anomaly rates in most configurations, although

Table 9.43: Anomaly rates for the exploits generated against pHsm by using the 'white-box' approach, tested against the five anomaly detectors utilized in this work

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	0.00%	0.00%	0.00%	0.00%	100.00%
restore	0.00%	0.00%	0.00%	0.00%	100.00%
samba	0.00%	0.00%	0.00%	0.00%	100.00%
ftpd	0.00%	0.00%	0.00%	0.00%	100.00%

in some configurations, such as the restore attack, the anomaly rates are comparable. It is important to emphasize that anomaly rates under a 'white-box' assumption makes use of all the information available to the attacker (sliding window length, schema mask employed, the normal database) whereas the 'black-box' approach succeeds in producing comparable results using only the anomaly rate as a guide for the search. This shows that although the 'black-box' assumption presents a more difficult problem, it can still provide solutions which are comparable to the 'white-box' scenarios.

Table 9.44: Anomaly rates for the attacks generated against pHsm by using the 'white-box' approach, tested against the five anomaly detectors utilized in this work

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	1.53%	9.36%	15.10%	0.95%	69.03%
restore	75.03%	78.31%	87.97%	33.79%	100.00%
samba	3.59%	9.95%	9.06%	6.65%	100.00%
ftpd	18.84%	21.79%	14.39%	6.02%	35.49%

The delays associated with the 'white-box' exploits generated against pHsm are detailed in Table 9.45 and the delays associated with the corresponding attacks are detailed in Table 9.46. In the experiments discussed in this thesis, the 'white-box' mimicry attack generation against pHsm is the only methodology, which produces no delays on the exploits. Since the exploit anomaly rate is 0% in all cases (except against the Neural Network detector), the associated delays are zero as well. As with the delays associated with 'white-box' attacks against Stide and pH, addition of the preamble introduces substantial delays.

Table 9.45: Delays for the exploits generated against pHsm by using the 'white-box' approach

	pH	**pHsm**
traceroute	0	0
restore	0	0
samba	0	0
ftpd	0	0

Table 9.46: Delays for the attacks generated against pHsm by using the 'white-box' approach

	pH	**pHsm**
traceroute	99659.7	4.85E+07
restore	1.90E+38	4.33E+38
samba	7.95E+27	1.01E+21
ftpd	5.26E+30	7.84E+17

9.3.4 The 'White-Box' Attacks Against the Markov Model

Tan et al. [97] utilized a methodology to generate mimicry attacks against a Markov Model detector. The main focus of their experiments was to determine the operational limits of Stide, therefore the attacks were generated against Stide and tested against both Stide and the Markov model detector.

Given that the Markov Model detector employed in this thesis is a first order model, Section 4.2.4, the current state (i.e. the system call) depends solely upon the previous system call. This differs from Stide, pH and pHsm, where the current state depends upon a history of previous system calls. The 'white-box' search on the Markov Model can be formulated as a graph search where each system call is a state. States A and B are connected in the graph if the detector encounters any transitions from A to B in the training data. The search for mimicry attacks now boils down to a depth-first search on the graph. The search terminates if the generated sequence contains the malicious sequence (i.e. open - write - close).

Results

The anomaly rates for the 'white-box' exploits are detailed in Table 9.47 and the anomaly rates for the attacks are detailed in Table 9.48. Although the anomaly rates for the exploits are 0% when the exploits are tested against the Markov Model detector, the exploit anomaly rates against the other detectors are fairly high. As discussed in the previous section, the exploits against pHsm were generated against a longer sliding window. Conversely, a first order Markov Model can be implemented as Stide with sliding window of 2. In other words, the normal database would record only the current system call and the previous system call, as opposed to recording a history of previous system calls. From that perspective, the exploits generated against a sliding window of size 2 do not generalize to the other detectors (namely, Stide, pH and pHsm) which employ longer sliding window patterns. Therefore, this indicates that, from an attacker's perspective, it is better to overestimate the sliding window length than to underestimate it.

As with the 'black-box' attacks (Table 9.8), the anomaly rates for the 'white-box' attacks (Table 9.48) are substantially higher when they are tested against other

Table 9.47: Anomaly rates for the exploits generated against the Markov Model detector by using the 'white-box' approach, tested against the five anomaly detectors utilized in this work

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	96.74%	96.63%	100.00%	0.00%	100.00%
restore	97.78%	75.86%	80.26%	0.00%	100.00%
samba	99.16%	81.90%	81.90%	0.00%	100.00%
ftpd	98.02%	90.82%	91.95%	0.00%	100.00%

detectors. The anomaly rates for the 'black-box' attacks are lower – particularly against the Neural Network detector, which suggests that, comparing the attacks against Markov Model detectors, 'black-box' attacks are more successful producing system call frequency distributions which are closer to the training than the 'white-box' attacks.

Table 9.48: Anomaly rates for the attacks generated against the Markov Model detector by using the 'white-box' approach, tested against the five anomaly detectors utilized in this work

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	68.15%	73.94%	87.02%	1.33%	100.00%
restore	79.06%	80.83%	90.35%	33.04%	100.00%
samba	6.54%	12.24%	11.27%	6.60%	100.00%
ftpd	22.13%	24.65%	17.25%	5.91%	40.29%

The delays associated with the 'white-box' exploits generated against the Markov Model detector are provided in Table 9.49 and the delays associated with the corresponding 'white-box' attacks are detailed in Table 9.50. The exploits do not cause substantial delays but the resulting attacks are delayed by more than 10^{29} seconds, which means the attacks are effectively 'frozen' before the exploit is deployed.

9.3.5 The 'White-Box' Attacks Against the Neural Network

The Neural Network detector differs from the rest of the anomaly detectors utilized in this thesis since the detection is based upon the frequency distribution of system calls

Table 9.49: Delays for the exploits generated against the Markov Model detector by using the 'white-box' approach

	pH	**pHsm**
traceroute	0.89	0.78
restore	0.87	0.76
samba	1.16	1.05
ftpd	0.98	0.87

Table 9.50: Delays for the attacks generated against the Markov Model detector by using the 'white-box' approach

	pH	**pHsm**
traceroute	8.11E+29	3.89E+32
restore	1.90E+38	4.35E+38
samba	1.32E+32	1.30E+32
ftpd	5.24E+32	2.60E+31

as opposed to the sequence of system calls. Such a representation is more compressed than the sequence-based methods, which means the attackers do not have to worry about determining sequences which contain malicious system calls. For example, from the perspective of the Neural Network detector which monitors the frequency distribution of system calls, sequence 2 2 1 3 2 2 1 is equivalent to 1 1 2 2 2 2 3. Furthermore, a mimicry attack can evade detectors which monitor system call sequences by repeating patterns which are in the normal database (e.g. the GP-generated traceroute mimicry attack against Stide in Figure B.1). However, such an approach may not work for a detector monitoring system call frequencies because although the sequences employed in the mimicry attack are in the normal database, the resulting frequency distribution may deviate from the frequency distributions encountered in the training set.

The 'white-box' methodology developed against the Neural Network detector analyses the frequency distributions in the training sets and generates mimicry attacks to match the frequency distribution. Again, this conforms to a 'white-box' assumption since the attacker can use internal knowledge including the training sets. Given that

the detector does not employ sequence information, the methodology arranges the ordering of the system calls randomly (i.e. as long as the frequency distributions match and the malicious open-write-close sequence exists, the 'white-box' attacker does not care about the ordering).

Results

The anomaly rates for the 'white-box' exploits generated against the Neural Network detector are detailed in Table 9.51 and the anomaly rates for the corresponding attacks are detailed in Table 9.52. The anomaly rates for the 'white-box' exploits are fairly low when tested against the Neural Network detector, but as opposed to the other detectors, the anomaly rates for the exploits are above zero even for the training detector. This is due to the fact that, in case of detectors which monitor sequences of system calls, if the 'white-box' attack produces sequences which are in the normal database of the detectors, it will remain undetected. However, in case of the Neural Network detector, the main purpose of employing a Neural Network on the frequency distribution of system calls is to develop a model which 'approximates' the frequency distributions encountered in the training set. Such an 'approximation' implies that a Neural Network provides a generic model which captures the generic characteristics of frequency distributions. Although this may raise some false positive concerns, it does increase the sensitivity of the detector to slight changes in normal behaviour, which has the potential to make it difficult for the attackers to generate mimicry attacks.

Table 9.51: Anomaly rates for the exploits generated against the Neural Network detector by using the 'white-box' approach, tested against the five anomaly detectors utilized in this work

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	100.00%	100.00%	71.41%	66.58%	1.87%
restore	11.17%	0.07%	0.07%	0.03%	0.22%
samba	85.67%	42.91%	42.91%	42.80%	2.38%
ftpd	74.78%	49.90%	0.17%	24.96%	0.46%

When the attack anomaly rates of the 'black-box' attacks against the Neural Network detector in Table 9.10 are compared with the 'white-box' attacks in Table

9.52, it is evident that the 'white-box' attacks produce low anomaly rates on the restore application. This is due to the facts that: (1) 88% of the system calls executed by restore are the `write` system call, therefore the sequence for such an attack is likely to contain blocks of the `write` system call, and (2) the restore exploit contains 167207 system calls (Table 9.31), which reduces the effect of anomalies from the preamble.

Table 9.52: Anomaly rates for the attacks generated against the Neural Network detector by using the 'white-box' approach, tested against the five anomaly detectors utilized in this work

	Stide	**pH**	**pHsm**	**Markov Model**	**Neural Network**
traceroute	94.83%	95.52%	71.04%	62.36%	3.84%
restore	11.74%	0.78%	0.86%	0.34%	0.27%
samba	81.14%	41.09%	41.04%	40.79%	3.12%
ftpd	74.35%	49.68%	0.28%	24.82%	0.95%

Delays associated with the 'white-box' exploits generated against the Neural Network detector are detailed in Table 9.53 whereas the delays associated with the corresponding 'white-box' attacks are provided in Table 9.54. Both the exploit and the attack delays are high for the 'white-box' attacks against the Neural Network. In this case, the lack of ordering produces clustered anomalies, which increases delays.

Table 9.53: Delays for the exploits generated against the Neural Network detector by using the 'white-box' approach

	pH	**pHsm**
traceroute	2.05E+39	3.82E+38
restore	4.84E+22	6.19E+24
samba	9.49E+40	9.49E+40
ftpd	5.66E+41	7.32E+38

Table 9.54: Delays for the attacks generated against the Neural Network detector by using the 'white-box' approach

	pH	**pHsm**
traceroute	2.07E+39	4.40E+38
restore	1.90E+38	4.33E+38
samba	9.49E+40	9.49E+40
ftpd	5.66E+41	7.32E+38

9.3.6 Discussion of the 'White-Box' Search Space Size

Based upon the search space size calculations employed for the 'black-box' approach in Section 8.2.4, two factors determine search space size: attack length and the number of available system calls. Given an attack which has l system calls and n distinct system calls, the total number of attacks of length l will be n^l. Although the number of system calls which an operating system implements varies, 223 system calls were encountered in the thesis experiments. Assuming that there are 223 system calls available to build an attack of length 1000, which is the same length utilized in the 'black-box' search space discussion, the search space size will be 223^{1000}, which is over 10^{2341} for all four vulnerable applications and for all the anomaly detectors employed in this thesis.

Compared with the 'black-box' search space size in Section 8.2.4, which is 10^{1301}, the 'white-box' search space is considerably larger. An exhaustive search performed on this extensive search space would be infeasible. Thus, a common aspect of 'white-box' mimicry attack research [105] [99] [96] [98] [32] [56] [34] is to employ 'white-box' information to reduce the search space size so that an exhaustive search can be performed.

Previous work on mimicry attacks employed various 'white-box' information such as patterns in the normal databases [105] [99] [96] [98] [32], detector parameters [96] [98] [32] and source code [56] to reduce the search space size.

For example, Wagner et al. [105] generated an automaton based upon the normal behaviour database and performed a depth-first search on the automaton. They state that their exhaustive search runs in less than a second [105] which indicates that the search space defined by their methodology is fairly small. As with Wagner et al. [105], Gao et al. [32] built an automaton as well – consisting of the normal behaviour database, program counter values and the set of return addresses on which the exhaustive search was conducted. On the other hand, Tan generated mimicry attacks manually based upon the normal behaviour database information of the detector [96] [99] or upon the training set of the detector [98]. Based upon the foreign sequence lengths reported in their experiments [99] [96] [98], it is evident that their exhaustive search focused on attacks of length 20 or shorter, which puts the estimation of their

search space around 223^{20}, which is approximately 10^{46}. Kruegel et al. [56] employed symbolic execution and reduced the application state to a polynomial by employing the source code and return address information. If the resulting polynomial has a solution, it implies that a mimicry attack exists. Giffin et al. [34] employed 'white-box' information such as the threat, program and operating system models to create abstractions (based upon their interpretation of the systems employed), on which model checking was performed.

In all the above mentioned work [105] [99] [96] [98] [32] [56] [34], 'white-box' knowledge from the detector was employed to reduce the search space size so that an exhaustive search could be conducted. On the other hand, the 'black-box' approach proposed in this thesis does not make use of any 'white-box' information, hence it works on a search space larger than the previous 'white-box' approaches [105] [99] [96] [98] [32] [56] [34]. Although this represents a more difficult problem from a machine learning perspective, the proposed approach succeeds in generating mimicry attacks which are comparable to the 'white-box' mimicry attacks, as demonstrated by the results in Sections 9.1 for 'black-box' attacks and 9.3 for 'white-box' attacks.

Moreover, a disadvantage of the 'white-box' approaches is that the search is dependent upon the availability of the needed 'white-box' information. For example, if the search requires access to the normal database of the detector, it cannot be applied to a scenario in which such information is not available. Furthermore, as discussed in Section 9.3, the 'white-box' search methodologies need to be tailored for each detector. For example, a 'white-box' methodology which employs the normal database of the Stide detector cannot be applied directly to pH, pHsm and Markov Model detectors. It has to be modified substantially to make it compatible with the data structures of the normal databases of those detectors. However, it cannot be applied to a Neural Network detector at all since not only is the data structure of the normal database (i.e. consisting of weight values) incompatible but also, the detector employs system call frequencies as opposed to sequences to perform detection. On the other hand, the proposed 'black-box' approach is employed against all five anomaly detectors without any changes in search methodology since the only requirement is suitable detection feedback (such as an anomaly rate) from the detector.

9.3.7 Discussion of the Analysis Results

In Section 9.1, the 'black-box' attacks generated by GP were deployed against multiple detectors to determine how the anomaly rate changes when the target detector is changed. In Section 9.2, two mimicry attacks which were published in relevant work [105] [34] were deployed against the anomaly detectors employed in this thesis. The results in Section 9.2 indicated that the mimicry attacks published in the relevant work created higher anomalies in the anomaly detector configurations employed in this research since the training sets employed while creating these attacks differ from the training sets employed to train the detectors. The results show that mimicry attacks are sensitive to training set changes, even though the previous work argued against this phenomenon [96] [97] [99]. Having established this phenomenon, Section 9.3 extended the analysis by focusing on developing 'white-box' methodologies to generate mimicry attacks against each anomaly detector configuration employed in this research.

The results demonstrated that utilizing a longer sliding window allows attackers to generate mimicry attacks which can generalize over to other detectors which employ the sliding window concept with shorter sliding window lengths. In particular, the 'white-box' mimicry attacks generated against pHsm (with sliding window length of 20) produced low anomaly rates against other detectors but the 'white-box' mimicry attacks generated against the Markov Model detector (with sliding window length of 2) produced high anomaly rates when deployed against other anomaly detectors.

Furthermore, the results indicate that a 'white-box' access to the detector provides a more straightforward way to facilitate the search for mimicry attacks if there is sufficient internal knowledge of the target detector. This extra internal knowledge is crucial to limit the 'white-box' search, otherwise the search process may become too computationally costly to perform. However, such internal knowledge also implies that a certain abstraction is made about the target detector. Hence, the resulting attacks can be detected if the abstraction is not accurate. On the other hand, the comparison of anomaly rates between the 'black-box' attacks in Section 9.1 and the 'white-box' attacks developed in this section revealed that although a 'black-box' access presents a more difficult problem, the proposed approach can generate attacks

with anomaly rates comparable to the 'white-box' attacks. This implies that mimicry attacks can be generated by assuming a 'black-box' approach, which does not require internal knowledge of the detector. Moreover, it is also established that the proposed approach utilizes the code bloat property of GP causing an increase in the mimicry attack length. In return, this enables the attacker to generate longer attacks, hence reducing the effect of long and anomalous preambles.

Moreover, the detection mechanism plays a crucial role in the success of mimicry attacks. For example, the mimicry attacks generated against the Neural Network detector did not perform well against the remaining detectors which employ sliding windows. This is most likely due to the fact that the mimicry attacks generated against the Neural Network do not aim to optimize the sequence of system calls. Similarly the mimicry attacks generated against the detectors which employ sliding windows did not perform well against the Neural Network detector because utilizing sequences which remain in the normal database of the detector does not necessarily mean that the resulting system call frequency distribution will be similar to the frequency distribution for normal behaviour.

The main property of a 'white-box' approach is that the attacker can utilize any relevant knowledge on the detector such as the training sets, detector normal database or configuration parameters to facilitate the mimicry attack search. During the development of 'white-box' mimicry attack generation methodologies it became apparent that each detector would require a custom search methodology depending upon the structure of the normal database, configuration parameters and detection mechanism. On the other hand, the proposed 'black-box' mimicry attack generation methodology using GP has the advantage of working against all the anomaly detectors utilized in this work without any major modifications or any abstraction. All the 'black-box' methodology needs to work is the anomaly rate from the detector. This implies that a 'black-box' approach is more extensible to other anomaly detectors, hence the vulnerability testing of the detectors can be performed without requiring multiple search methods for each detector.

Chapter 10

Analysis of Vulnerable Applications

Chapter 9 focused on analysing mimicry attacks where both 'white-box' and 'black-box' mimicry attacks were deployed against different detector configurations. This chapter, however, investigates the applications which the mimicry attacks were trained on. The aim of the analysis is to identify the characteristic of each application which was employed in the mimicry attack experiments discussed in this thesis and to investigate the effects of the application characteristics in mimicry attack generation. Furthermore, this chapter examines the best attacks generated against each detector employed in the mimicry attack experiments in Chapter 8 and identifies the techniques which the attacks employed to evade detection.

10.1 Analysis of System Calls and Normal Databases

Given that the aim of the mimicry attack is to hide within the normal behaviour of the detector, the analysis detailed in this section aims to determine the effects of normal behaviour characteristics in the resulting mimicry attacks. Within this analysis, Table 10.1 details the number of system calls collected for each application by executing the use cases detailed in Tables 8.1, 8.2, 8.3 and 8.4 for traceroute, restore, samba and ftpd, respectively. Traceroute and restore, which are local UNIX programs, utilize smaller sets of system calls compared with samba and ftpd, which provide services over the network. In terms of the total number of system calls in the traces, ftpd and restore have the largest sets since they both have use cases which process large files (i.e. longer execution hence more system calls). As a result, from Table 10.1, samba and ftpd, which are the two remote UNIX services, have the larger sets of system calls in terms of unique system call counts. On the other hand restore and ftpd, which deal with large amounts of file I/O, have the largest number of system calls in total.

The next step in analysing the characteristic of each application is to investigate the size of the normal behaviour data structure for the detector. Naturally, as the normal behaviour of an application becomes more complex, the size of the data structure will increase. To this end, the data structure lengths for the Stide, pH, pH with a schema mask (pHsm) and Markov Model detectors are provided in Table 10.2. The Neural Network detector is not included in the analysis because it employs the same number of weight values regardless of the complexity of the normal behaviour. In other words, the length of the data structure, provided in Table 4.5, is pre-determined by the user, hence it does not change.

Stide stores the sequences which it encounters in the training data, therefore the unit of measure for Stide is the number of sequences stored in the normal database. pH and pHsm, on the other hand, store the tuple (current system call, previous system call, location of the previous system call in the sliding window) and therefore the unit of measure is the number of tuples. A first order Markov Model can be implemented as a matrix in which the normal behaviour model encapsulates the transitions between the system calls. Given that detection is based upon transitions which have a zero probability on the Markov Model, an appropriate measure for the normal database is the number of non-zero probabilities, or, in other words, the number of distinct transitions which the Markov Model encountered during training.

Two local applications, traceroute and restore, have the smaller normal behaviour data structures in Table 10.2, which seem to indicate that network services (samba and ftpd) have more complex behaviour. This can provide the attacker a wider set of system calls from which to build mimicry attacks. Furthermore, incorporating the schema mask in pH seems to increase the normal data structure size for network services.

Tables 10.1 and 10.2 suggest that not all applications are created equal and certain applications may have more extensive sets of normal behaviour. This can make it easier for an attacker to generate a mimicry attack since it represents a larger set within which the attacker can hide. Although ftpd seems to have a wide set of normal behaviour and the GP-generated attacks produced low exploit anomaly rates, the anomalies from the preamble resulted in attacks which produced high anomaly

rates. This means that deploying the attack over the network can present a more difficult problem since the attacker has limited access to the application and may not be able to prevent the preamble from producing anomalies.

Table 10.1: A summary of the system calls collected for traceroute, restore, samba and ftpd

	traceroute	**restore**	**samba**	**ftpd**
Number of unique system calls	29	30	58	58
Total number of system calls	1315	238728	69451	747300

Table 10.2: The analysis of the detector normal behaviour data structures for the traceroute, restore, samba and ftpd applications

	traceroute	**restore**	**samba**	**ftpd**
Number of unique sequences (Stide)	198	198	847	764
Number of unique tuples (pH)	1124	845	3353	3229
Number of unique tuples (pHsm)	1247	892	7452	4185
Number of transitions with non-zero probabilities (Markov Model)	81	84	259	241

The system call counts were analysed on a per-system-call basis for the traceroute, restore, samba and ftpd applications in Tables 10.3, 10.4, 10.5 and 10.6, respectively. Compared with the system call distribution of the restore, samba and ftpd applications, traceroute had a more even distribution. Conversely, in the case of restore, over 88% of all system calls executed were the `write` system calls. The two most frequently used system calls for samba were `read` and `_llseek`, both of which perform I/O operations. In the case of ftpd, the most frequently executed system calls perform I/O timing and signalling functions. The discussion of the system calls are detailed further in Section 10.2 while analysing mimicry attacks.

Table 10.3: System call counts and percentages for traceroute

ID	System Call	Count	(%)	ID	System Call	Count	(%)
1	gettimeofday	220	16.730%	16	connect	20	1.521%
2	write	142	10.798%	17	ioctl	15	1.141%
3	mmap	113	8.593%	18	uname	14	1.065%
4	select	99	7.529%	19	getpid	12	0.913%
5	sendto	99	7.529%	20	time	10	0.760%
6	close	93	7.072%	21	send	10	0.760%
7	open	86	6.540%	22	poll	10	0.760%
8	read	75	5.703%	23	setsockopt	6	0.456%
9	fstat	73	5.551%	24	_exit	5	0.380%
10	munmap	49	3.726%	25	execve	5	0.380%
11	mprotect	34	2.586%	26	personality	5	0.380%
12	socket	29	2.205%	27	stat	4	0.304%
13	recvfrom	28	2.129%	28	setuid	3	0.228%
14	brk	27	2.053%	29	getuid	3	0.228%
15	fcntl	26	1.977%				

Table 10.4: System call counts and percentages for restore

ID	System Call	Count	(%)	ID	System Call	Count	(%)
1	write	211704	88.680%	16	chown	8	0.003%
2	read	26378	11.049%	17	ioctl	8	0.003%
3	lseek	132	0.055%	18	stat	8	0.003%
4	mmap	91	0.038%	19	rt_sigaction	8	0.003%
5	open	67	0.028%	20	_llseek	8	0.003%
6	close	56	0.023%	21	fchmod	7	0.003%
7	fstat	51	0.021%	22	fchown	7	0.003%
8	mprotect	30	0.013%	23	_exit	5	0.002%
9	munmap	29	0.012%	24	execve	5	0.002%
10	brk	24	0.010%	25	getpid	5	0.002%
11	fcntl	23	0.010%	26	umask	5	0.002%
12	rt_sigprocmask	23	0.010%	27	personality	5	0.002%
13	utime	15	0.006%	28	setuid	4	0.002%
14	unlink	8	0.003%	29	getuid	4	0.002%
15	chmod	8	0.003%	30	mkdir	2	0.001%

Table 10.5: System call counts and percentages for samba

ID	System Call	Count	(%)	ID	System Call	Count	(%)
1	read	28329	40.790%	30	setsockopt	15	0.022%
2	_llseek	9612	13.840%	31	socket	14	0.020%
3	select	9398	13.532%	32	getpid	11	0.016%
4	gettimeofday	9395	13.528%	33	setrlimit	11	0.016%
5	send	9389	13.519%	34	getrlimit	11	0.016%
6	fcntl64	1133	1.631%	35	connect	11	0.016%
7	stat	440	0.634%	36	getpeername	8	0.012%
8	open	218	0.314%	37	getuid32	8	0.012%
9	close	205	0.295%	38	chdir	4	0.006%
10	munmap	153	0.220%	39	fchmod	4	0.006%
11	mmap	126	0.181%	40	uname	4	0.006%
12	mmap2	126	0.181%	41	recvfrom	4	0.006%
13	getegid32	99	0.143%	42	poll	4	0.006%
14	geteuid32	97	0.140%	43	rt_sigprocmask	4	0.006%
15	time	91	0.131%	44	_exit	3	0.004%
16	getsockopt	72	0.104%	45	fork	3	0.004%
17	mprotect	46	0.066%	46	chmod	3	0.004%
18	umask	44	0.063%	47	utime	3	0.004%
19	setresgid32	44	0.063%	48	pipe	3	0.004%
20	setresuid32	44	0.063%	49	bind	3	0.004%
21	setgroups32	41	0.059%	50	accept	3	0.004%
22	rt_sigaction	34	0.049%	51	getsockname	3	0.004%
23	write	32	0.046%	52	rename	1	0.001%
24	brk	31	0.045%	53	times	1	0.001%
25	getgroups32	26	0.037%	54	ioctl	1	0.001%
26	alarm	24	0.035%	55	setpriority	1	0.001%
27	pwrite	24	0.035%	56	fgetxattr	1	0.001%
28	ftruncate64	18	0.026%	57	getxattr	1	0.001%
29	getdents64	16	0.023%	58	chown32	1	0.001%

Table 10.6: System call counts and percentages for ftpd

ID	System Call	Count	(%)	ID	System Call	Count	(%)
1	read	182325	24.398%	30	getrlimit	67	0.009%
2	rt_sigaction	181876	24.338%	31	setrlimit	60	0.008%
3	alarm	181649	24.307%	32	setsockopt	46	0.006%
4	write	181596	24.300%	33	setresuid	44	0.006%
5	close	11443	1.531%	34	select	33	0.004%
6	open	1045	0.140%	35	_llseek	32	0.004%
7	time	1037	0.139%	36	getuid	30	0.004%
8	mmap	948	0.127%	37	dup2	30	0.004%
9	fstat	716	0.096%	38	getsocknam	30	0.004%
10	munmap	506	0.068%	39	wait4	27	0.004%
11	chdir	364	0.049%	40	execve	20	0.003%
12	fcntl	271	0.036%	41	getpeername	20	0.003%
13	getcwd	267	0.036%	42	personality	20	0.003%
14	socket	256	0.034%	43	getdents	18	0.002%
15	connect	256	0.034%	44	bind	13	0.002%
16	fchdir	240	0.032%	45	kill	12	0.002%
17	mprotect	230	0.031%	46	nanosleep	12	0.002%
18	lstat	227	0.030%	47	gettimeofday	11	0.001%
19	send	195	0.026%	48	_exit	10	0.001%
20	brk	187	0.025%	49	fork	10	0.001%
21	stat	183	0.024%	50	sigreturn	10	0.001%
22	poll	160	0.021%	51	accept	10	0.001%
23	recvfrom	140	0.019%	52	shutdown	10	0.001%
24	getpid	131	0.018%	53	getsockopt	10	0.001%
25	rt_sigprocmask	101	0.014%	54	setregid	9	0.001%
26	umask	99	0.013%	55	setgroups	9	0.001%
27	uname	78	0.010%	56	quotactl	9	0.001%
28	lseek	74	0.010%	57	pipe	7	0.001%
29	flock	74	0.010%	58	vfork	7	0.001%

10.1.1 Discussion of the Analysis Results

Since the main objective of mimicry attacks (in this context) is to hide the true intention of the code within normal behaviour, the analysis in this section focused on the analysis of the system calls collected during the normal use of each application as defined in Tables 8.1, 8.2, 8.3 and 8.4 for traceroute, restore, samba and ftpd, respectively. Furthermore, the normal behaviour data structures of the detectors were investigated to determine whether certain applications have larger normal behaviour models resulting in an increase in normal data structure size.

The analysis of the system calls revealed that different applications utilize different sets of system calls with different frequencies. Network services samba and ftpd have more unique system calls than the local services traceroute and restore. Furthermore, analysis of the normal behaviour data structures of the detectors revealed that the network services have larger normal behaviour data structures compared with the local services. Larger normal behaviour models are an indication of more complex application behaviour and since the network services need to communicate with network clients, their normal operation includes a wider set of system calls. This implies that an attacker has a wider set of system calls with which to hide the intent of the attack. However, the preamble lengths (in Tables 8.21, 8.22, 8.23 and 8.24) indicate that the break-in process for remote services tends to be lengthy and anomalous, which can provide an additional challenge to the attacker.

Furthermore, the analysis of system call frequency distributions for the four applications indicates that the system calls are not distributed evenly. For example, 99% of the system calls which restore executes are `read` and `write` system calls. Section 10.2 furthers the discussion on the system call frequency distributions while discussing the mimicry attacks which GP generated.

10.2 Analysis of Mimicry Attacks

The analysis detailed in this section utilizes the 25000 attacks per application (50 runs, 500 attacks per run) provided in Chapter 8. The box plot results in Chapter 8 compared the results among the anomaly detectors employed in the experiments by comparing the characteristics of the generated mimicry attacks on different detectors, given an application. On the other hand, the analysis in this section analyses the same attacks by investigating how the characteristics of the generated mimicry attacks change for different applications, given a detector. The purpose of such analysis is to investigate how the application characteristics affect the search for mimicry attacks.

Within the analysis of attacks against each detector, first, all the attack populations generated against the given detector were compared using box plot analysis. Subsequently, the best exploits from each population (one for each application) were selected, where the 'best' was defined as the attack with the lowest attack anomaly rate. The 'best' performing attacks were then discussed in terms of the system calls which they employed to camouflage the attack.

10.2.1 Attacks Against Stide

The box plot of the exploit anomaly rates against Stide is shown in Figure 10.1 and the box plot of the corresponding attack anomaly rates is shown in Figure 10.2. Furthermore, the box plot of the exploit lengths is shown in Figure 10.3. In Figure 10.3, the exploit lengths are comparable except for the attacks against the restore application where the exploits have a wider range of lengths. This suggests a correlation between the distribution of the system calls and the exploit lengths since restore differs from the other applications with the unbalanced distribution of system calls in the normal behaviour traces, Table 10.4.

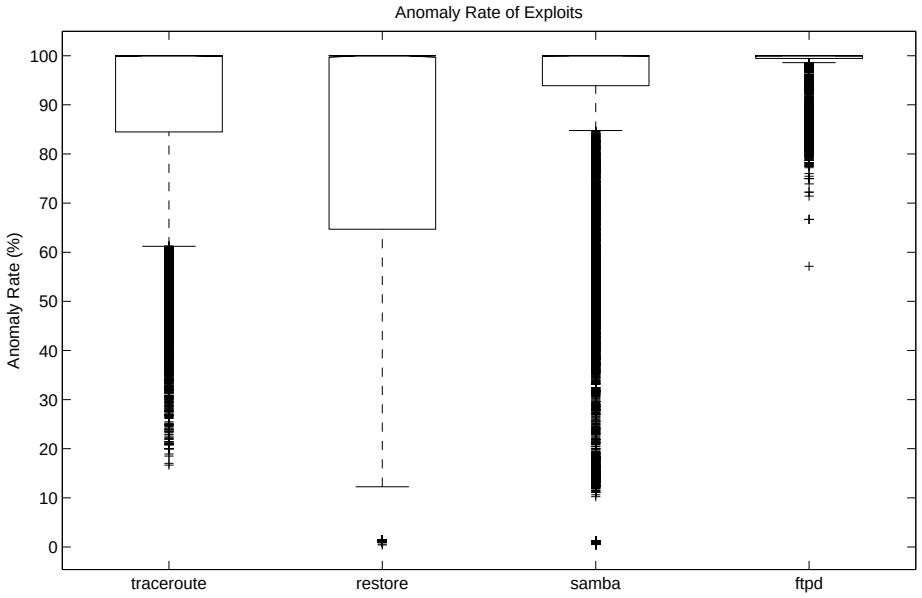

Figure 10.1: Box plot of the mimicry exploit anomaly rates for Stide on traceroute, restore, samba and ftpd

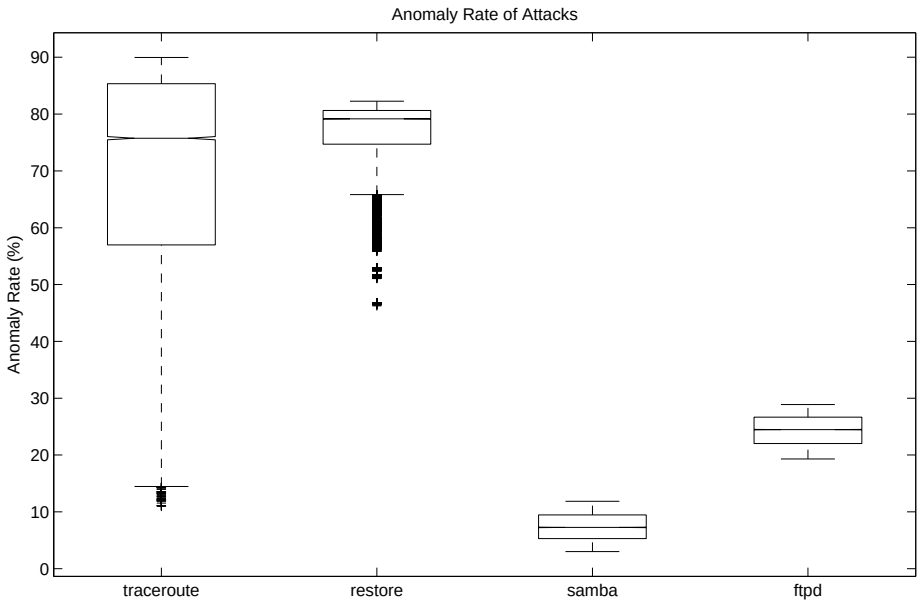

Figure 10.2: Box plot of the mimicry attack anomaly rates for Stide on traceroute, restore, samba and ftpd

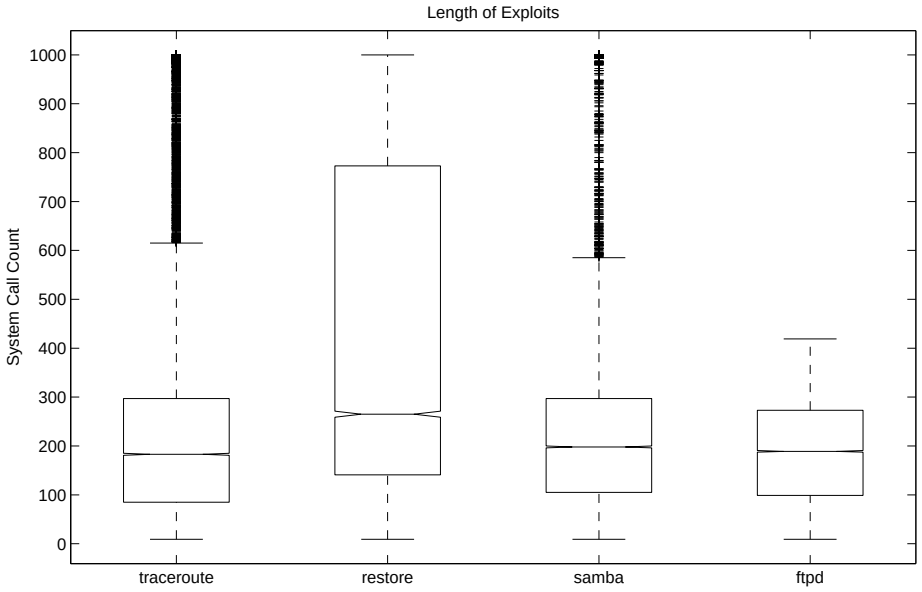

Figure 10.3: Box plot of the mimicry exploit lengths for Stide on traceroute, restore, samba and ftpd

A Closer Look at GP-Generated Mimicry Attacks Against Stide

The best mimicry exploit (i.e. that with the lowest attack anomaly rate) against traceroute is provided in Figure B.1. It is apparent that 6 system calls are repeated in the exploit: `gettimeofday sendto gettimeofday select write write`. The traceroute exploit against Stide hides its true intention within timing and I/O system calls. `gettimeofday` provides the timing functionality needed to obtain the round trip time for the traceroute packets, whereas `sendto`, `select` and `write` provide various I/O functions.

The best mimicry exploit against samba, Figure B.2 repeats a pattern which consists of a `read` followed by a number of `_llseek` system calls. `read` implements a read from a resource whereas `_llseek` moves the file pointer. GP employs this pattern as a smokescreen in between the `open` and `write close` system calls to evade detection. This is not surprising since `_llseek` and `read` are the mostly utilized system calls in the system call traces.

Similarly, the best mimicry exploit against restore in Figure B.3 employs a pattern in which a number of `write` system calls precede a `read` system call. The exploit deploys this smokescreen pattern and follows up with the attack system calls as the last three system calls.

The best mimicry exploit against ftpd in Figure B.4 is concise compared with the other exploits generated against Stide. In this exploit, no repeating pattern is employed by the GP. However GP employs system calls which involve file checks such as `getcwd fchdir fstat` along with the system calls which involve timing, such as `alarm`.

The common characteristics of the GP-generated attacks against Stide is that GP identifies and employs a set of system calls which were used frequently by the application during its normal operation. In the case of traceroute, GP repeats a certain pattern, whereas in samba and restore, the repetition of system calls is not as clear. On the other hand, GP chooses to utilize a shorter exploit with no repeating patterns in ftpd, which seems to indicate that different exploit techniques exist against different applications.

10.2.2 Attacks Against pH

The box plots of the exploit and attack anomaly rates against pH are detailed in Figures 10.4 and 10.5, respectively. Compared with the anomaly rates of the exploits and the corresponding attacks against Stide in the previous subsection, the anomaly rates in Figures 10.4 and 10.5 are comparatively low, which suggests that generating exploits against the look-ahead pair method in pH may be easier than generating exploits against the full sequence method in Stide. This condition is discussed in detail by Inoue et al. [38] and briefly in Section 9.3.2 of this thesis.

A box plot of the delays associated with the exploits is provided in Figure 10.6 and the delays associated with the corresponding attacks are given in Figure 10.7. Furthermore, the box plot of the attack lengths is provided in Figure 10.8. It is interesting to note that the exploit anomaly rates (and associated delays) on traceroute are higher that the exploit anomaly rates (and associated delays) on the remaining applications. Given that traceroute utilizes a smaller set of system calls and produces smaller normal behaviour data structures, the expectation would be that traceroute has a simpler normal behaviour characteristic. The results indicate that there are other factors which affect the success of the mimicry attacks. In addition, the anomaly rates and delays associated with the exploits and attacks support the finding that traceroute presents a more difficult problem for mimicry attack generation. This can be attributed to the fact that the frequency distribution of system calls for traceroute (Table 10.3) is more even than for restore, samba and ftpd (Tables 10.4, 10.5 and 10.6, respectively).

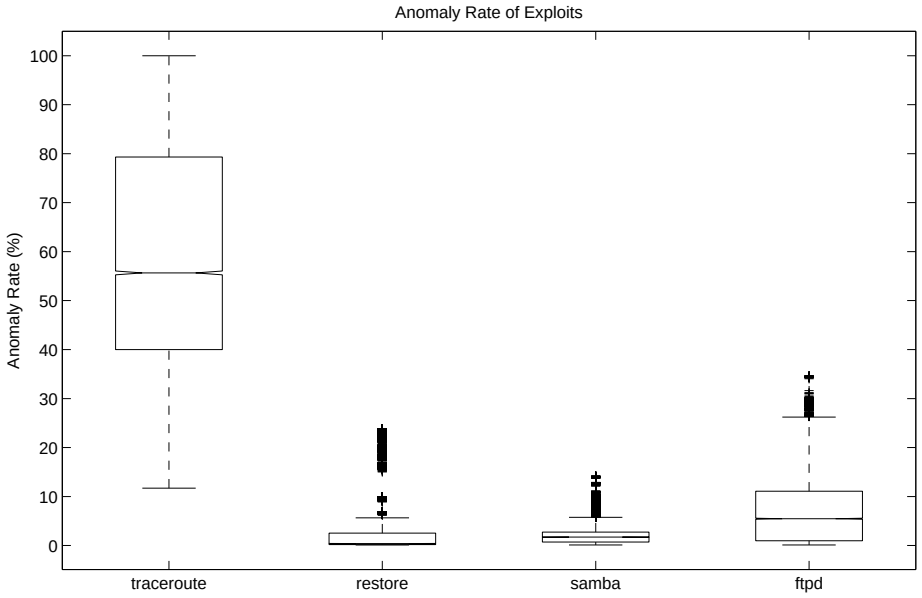

Figure 10.4: Box plot of the mimicry exploit anomaly rates for pH on traceroute, restore, samba and ftpd

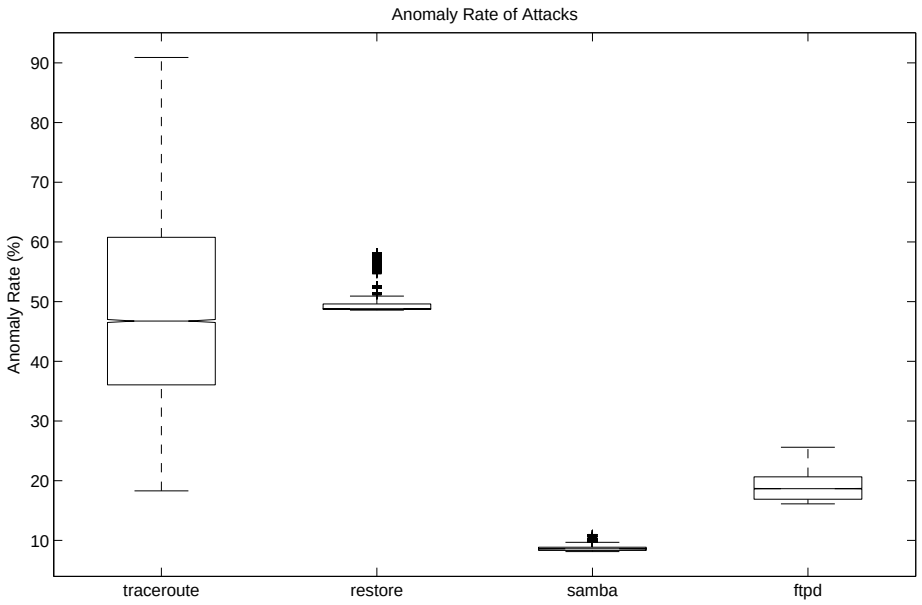

Figure 10.5: Box plot of the mimicry attack anomaly rates for pH on traceroute, restore, samba and ftpd

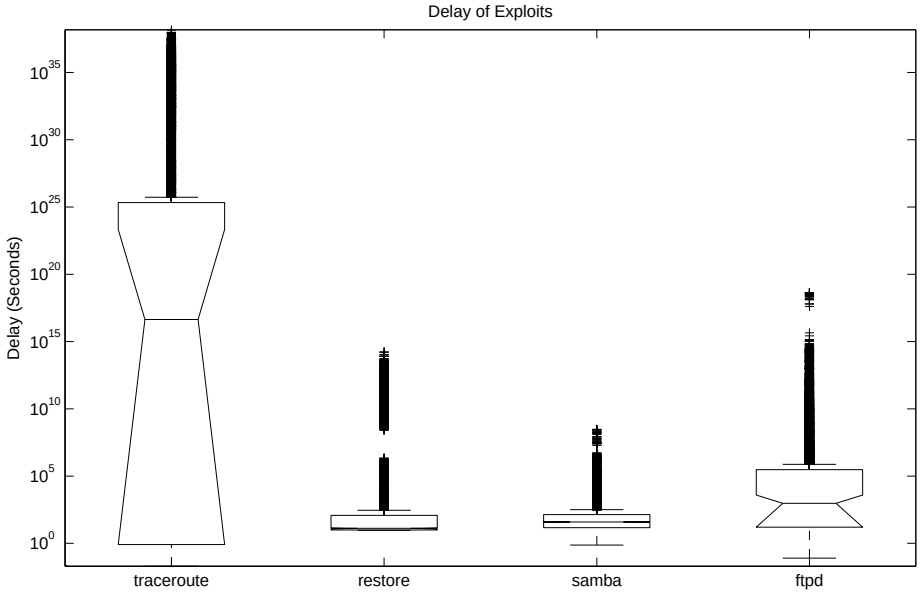

Figure 10.6: Box plot of the mimicry exploit delays for pH on traceroute, restore, samba and ftpd

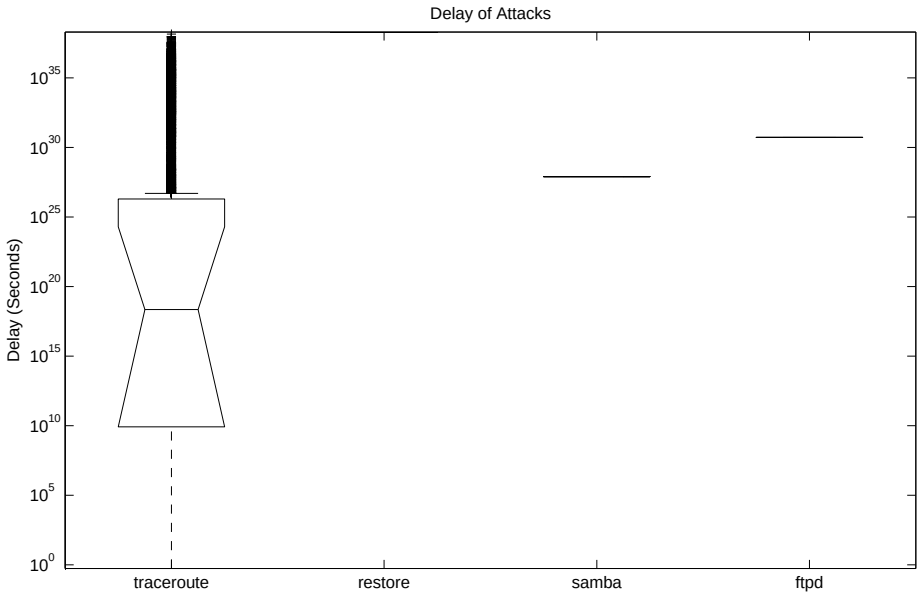

Figure 10.7: Box plot of the mimicry attack delays for pH on traceroute, restore, samba and ftpd

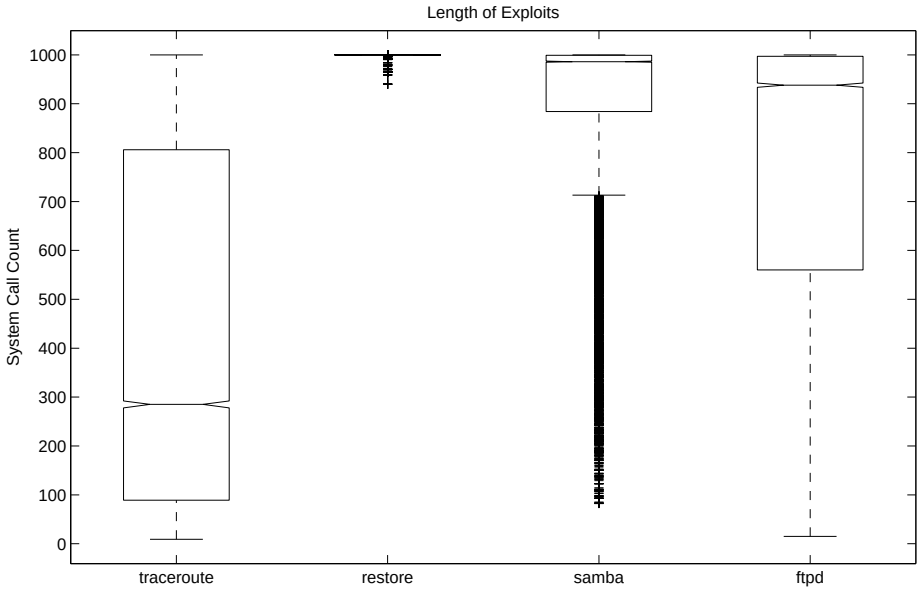

Figure 10.8: Box plot of the mimicry exploit lengths for pH on or traceroute, restore, samba and ftpd

A Closer Look at GP-Generated Mimicry Attacks Against pH

The best exploit against pH for traceroute is provided in Figure B.5. As opposed to the traceroute attack against Stide, the attack against pH does not employ any clear repeating pattern but employs different combinations of memory access system calls, such as `mmap` and `munmap`, and file access system calls, such as `open` `fstat` and `close`, to hide the true intent.

The best mimicry exploit against pH for samba in Figures B.6 and B.7 is fairly long. Although there are no clear repeating patterns, the strategy which the attack employs is to deploy various memory access and file I/O system calls (such as `mmap2`, `munmap`, `stat`) followed by a block of `fcntl64` system calls which manipulates a given file descriptor. This is different from the samba mimicry attack against Stide in which the exploit focuses on utilizing `read` and `_llseek` system calls. In terms of the attack system calls, the `open` and `write` system calls are toward the beginning of the exploit whereas the `close` system call is toward the end.

The best mimicry exploit against pH for restore in Figure B.8, shows that the exploit alternates between a single `lseek` and a block of `write` system calls, where `lseek` provides random access to a file. `lseek` is the third most frequent system call in the system call traces, whereas `write` is the most frequent (Table 10.4). Similarly, the restore attack against Stide employed `write` and `read` system calls primarily, which are the two most frequent system calls.

Figure B.9 shows the best exploit against pH on ftpd. Compared to the ftpd exploit against Stide, this exploit is longer and follows a different strategy and employs combinations of `read`, `write` and `close` system calls whereas the ftpd exploit against Stide is shorter and employs system calls related to timing and file checks.

As in the exploit techniques which GP employed against Stide, the exploits against pH contain system calls which the applications execute frequently during their normal operation, albeit utilizing different system calls from the attacks against Stide. Although no clear repeating pattern exists, different combinations of the more frequent system calls were injected between the malicious system calls to hide the true intention of the attack.

10.2.3 Attacks Against pHsm

The analysis of applications for pHsm involves two scenarios: (1) the attacker knows the schema mask which the detector uses or (2) the mask is unknown. In the former scenario, the attacker uses the appropriate schema mask, whereas in the latter scenario a different schema mask is used. Therefore, the measurements are obtained for each scenario separately.

Figure 10.9 shows the box plot for the exploit anomaly rates for pHsm whereas Figure 10.10 shows the exploit anomaly rates when the schema mask is unknown to the attacker. Similarly, Figure 10.11 details the anomaly rates for the attack, compared to Figure 10.12, which details the anomaly rates for the attack when the schema mask is unknown to the attacker. Furthermore, the delays associated with the exploits are detailed in Figures 10.13 and 10.14, for the scenarios where the schema mask is known and unknown to the attacker, respectively. Similarly, Figures 10.15 and 10.16 demonstrate the delays associated with the corresponding attacks for the scenarios where the schema mask is known and unknown to the attacker, respectively. Finally, the box plot for the exploit lengths is provided in Figure 10.17.

A comparison of the box plots between the scenarios where the schema mask is known and unknown indicates that a schema mask is crucial to the success of mimicry attacks. In other words, the anomaly rate and the delay box plots demonstrate that when the attacker needs to generate mimicry attacks against pH without knowing the schema mask, the anomaly rates and delays for both the exploits and the attacks increase. Although knowing the schema mask provides GP valuable information which helps to reduce anomaly rates, the 'black-box' approach assumed by GP succeeded in working without the schema mask information, thus producing exploits with comparable (albeit higher) anomaly rates.

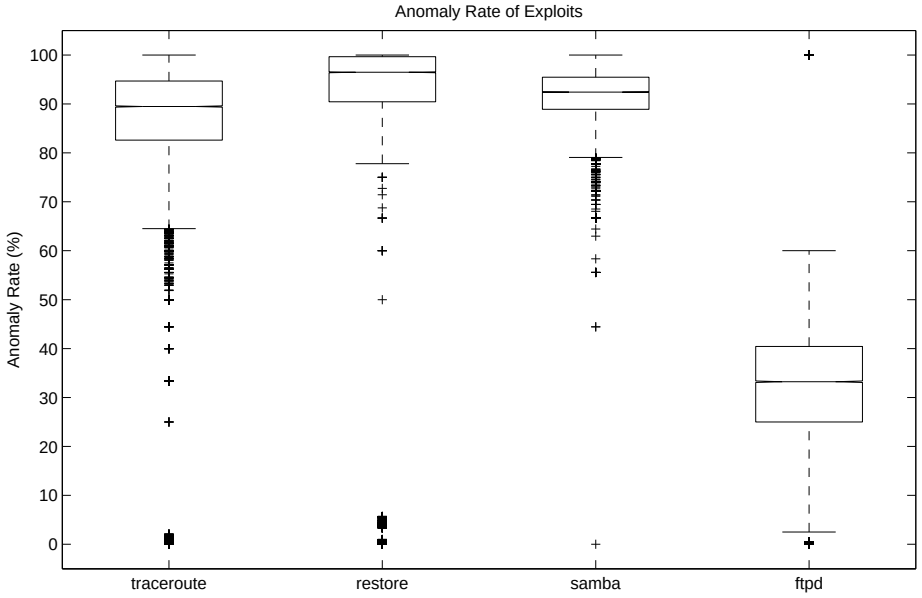

Figure 10.9: Box plot of the mimicry exploit anomaly rates for pHsm on traceroute, restore, samba and ftpd

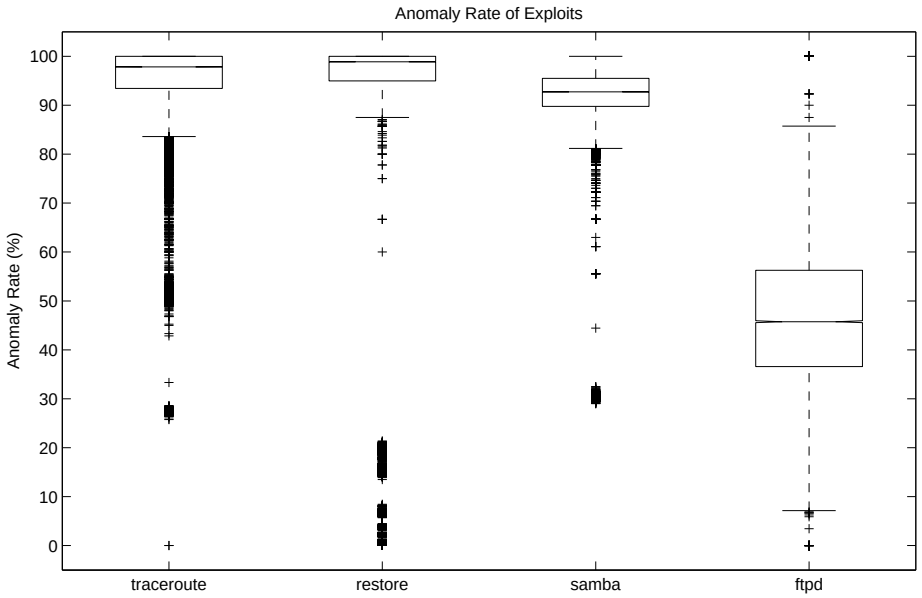

Figure 10.10: Box plot of the mimicry exploit anomaly rates for pHsm (mask unknown) on traceroute, restore, samba and ftpd

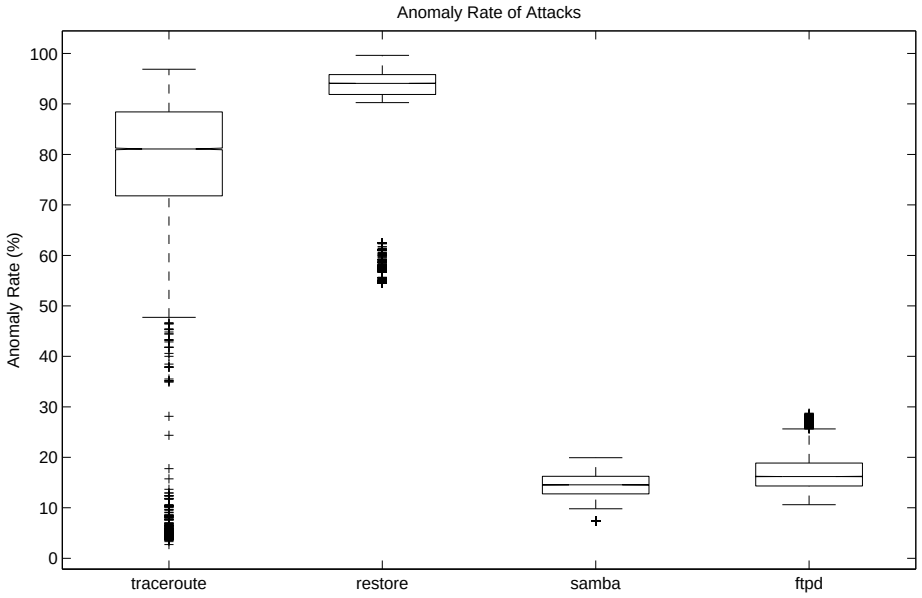

Figure 10.11: Box plot of the mimicry attack anomaly rates for pHsm on traceroute, restore, samba and ftpd

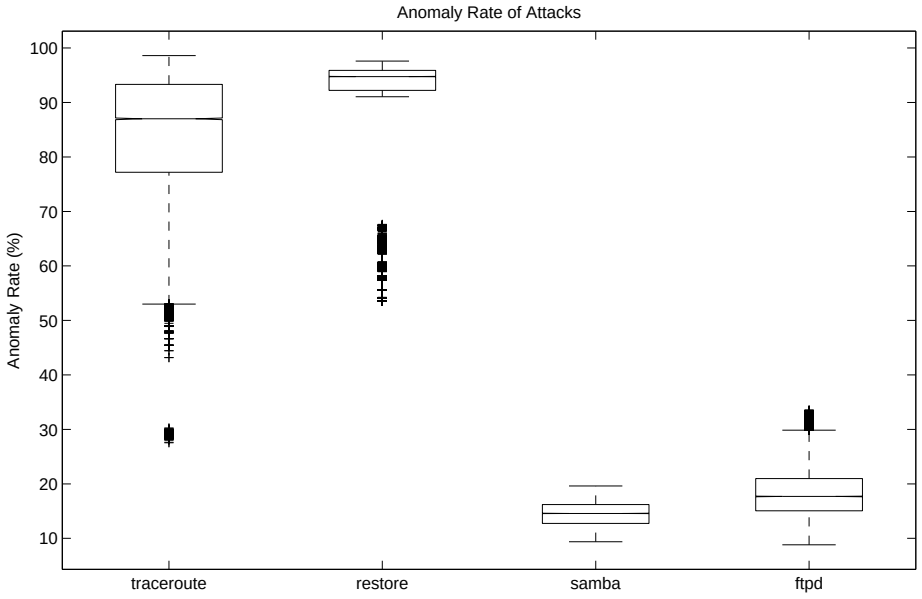

Figure 10.12: Box plot of the mimicry attack anomaly rates for pHsm (mask unknown) on traceroute, restore, samba and ftpd

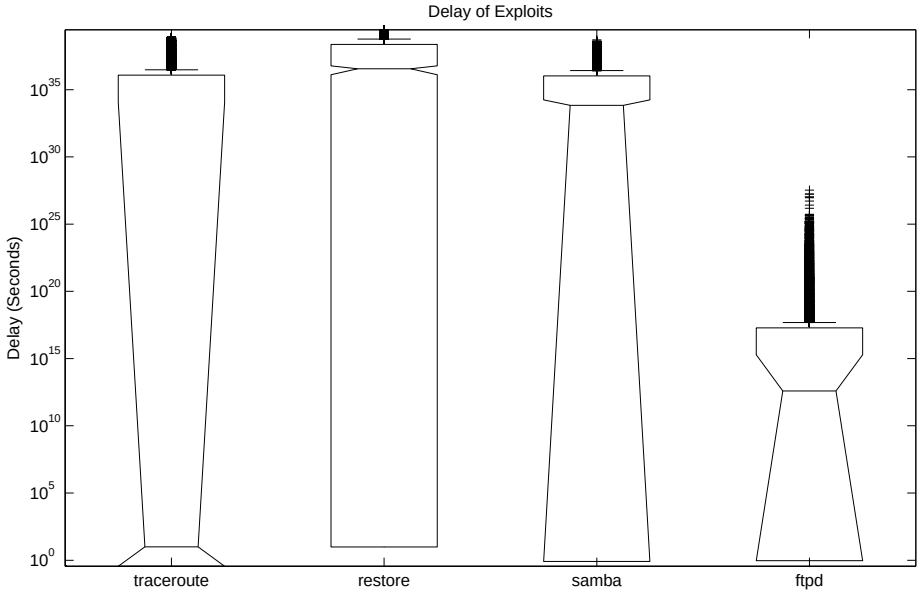

Figure 10.13: Box plot of the mimicry exploit delays for pHsm on traceroute, restore, samba and ftpd

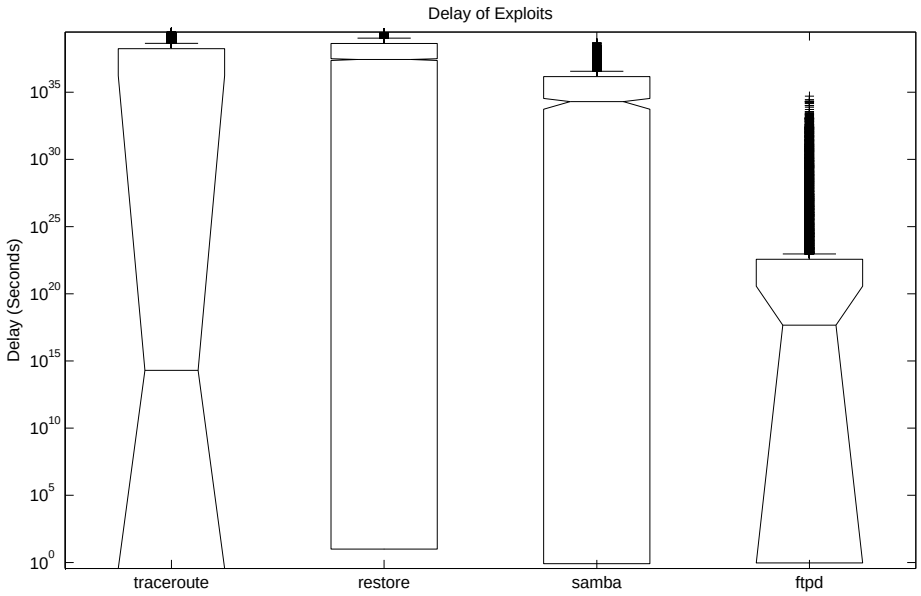

Figure 10.14: Box plot of the mimicry exploit delays for pHsm (mask unknown) on traceroute, restore, samba and ftpd

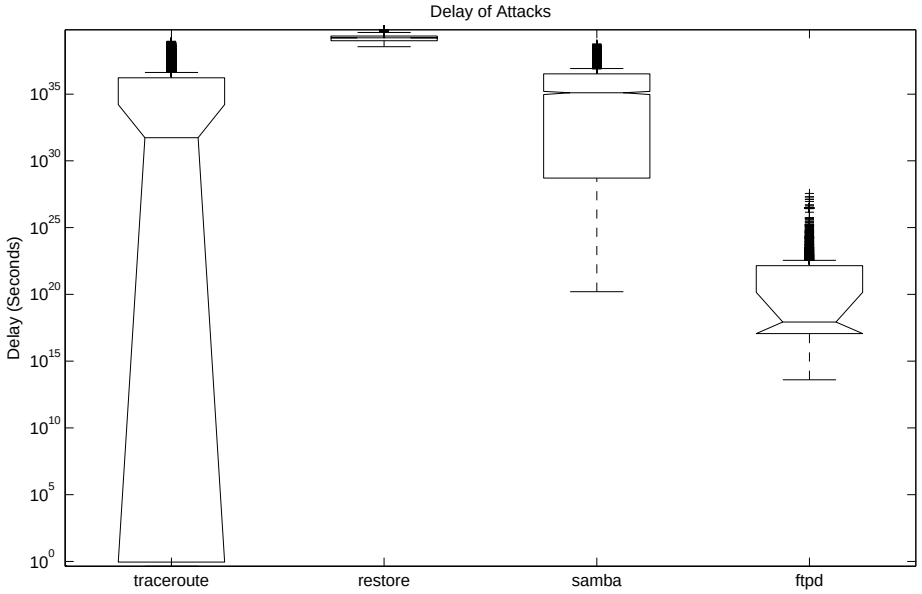

Figure 10.15: Box plot of the mimicry attack delays for pHsm on traceroute, restore, samba and ftpd

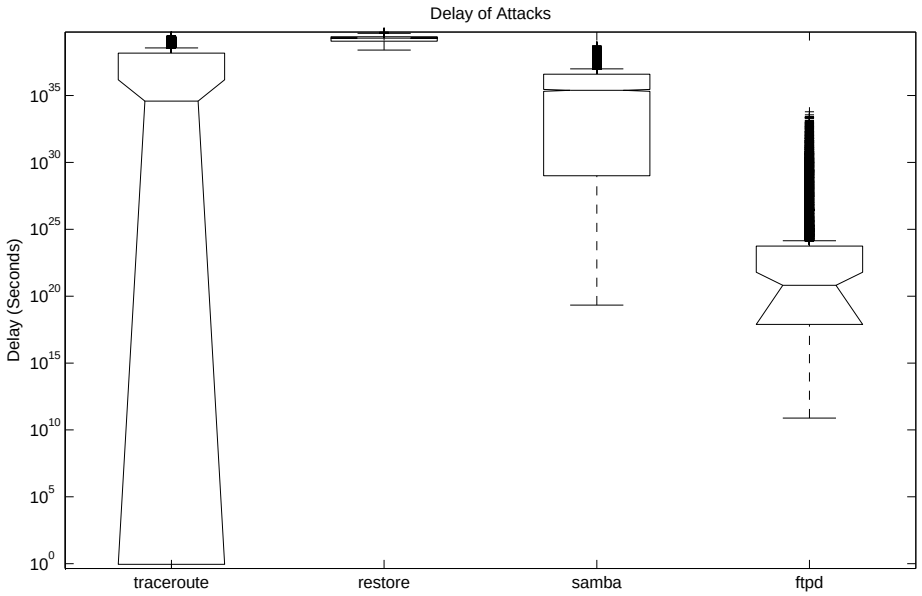

Figure 10.16: Box plot of the mimicry attack delays for pHsm (mask unknown) on traceroute, restore, samba and ftpd

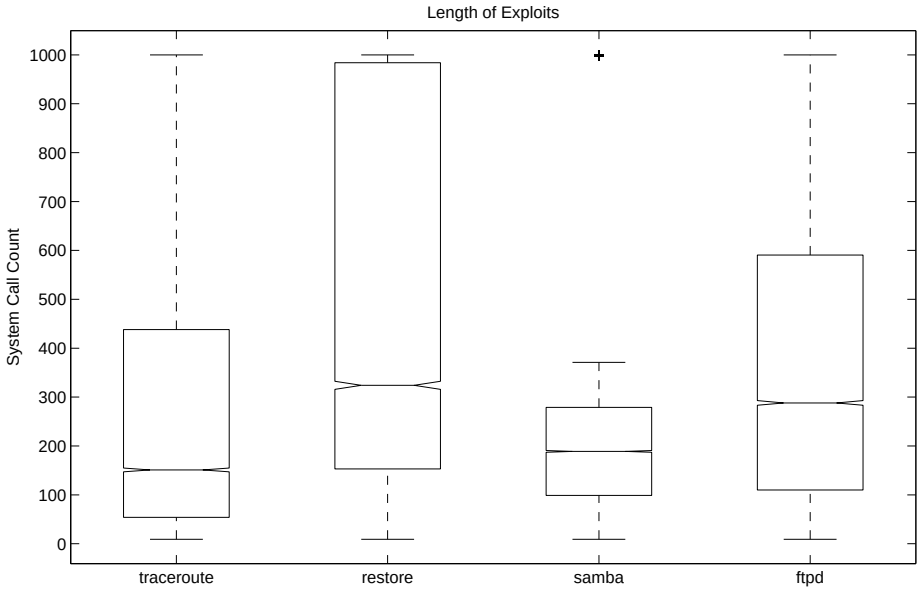

Figure 10.17: Box plot of the mimicry exploit lengths for pHsm on traceroute, restore, samba and ftpd

A Closer Look at GP-Generated Mimicry Attacks Against pHsm

The best mimicry exploit against pHsm on traceroute is provided in Figure B.10. In this exploit, the true intention of the attack is hidden between blocks of `open` and `mmap` system calls which handle file I/O and memory mapping functions, respectively. This trait is similar to the mimicry exploit against pH but different from the Stide exploit which utilizes timing system calls.

The best mimicry exploit on samba is provided in Figures B.11 and B.12, where the exploit does not utilize a clear repeating pattern but uses a combination of `read`, `stat`, `munmap`, `mmap` and other numerous system calls. The system calls related to the attack is within the first 10 system calls. Given that the most frequent system calls are somewhat evenly distributed for samba, the exploit utilizes a number of system calls to hide the true intention of the code. This is a common trait shared among the samba exploits against Stide and pH, where similar system calls are utilized.

Similarly, the best mimicry exploit on restore in Figures B.13 and B.14 learns to hide within normal behaviour by utilizing a series of `write` and `read` system calls which make up the 99% of the executed system calls by restore (Table 10.4). Restore exploits against both Stide and pH utilize the same technique to hide the true intention of the exploit.

As opposed to the concise ftpd exploits for Stide and pH, the ftpd exploit against pHsm (Figures B.15 and B.16) is comparatively long. Although there are no clear repeating patterns, the ftpd exploit utilizes frequently used system calls such as `rt_sigaction`, `open`, `read`, `close`, `time` (Table 10.6) to formulate the padding within which the attack system calls are hidden.

While employing numerous system calls which do not appear on exploits against Stide and pH, GP follows similar techniques for hiding the true intent of the code. That is to say, although no repeating pattern clearly exists, the exploits generated by GP against pHsm utilize the system calls which are used frequently during the normal operation of the applications.

In cases where the frequency of the system call distribution is even, such as in the case of traceroute, Table 10.3, GP utilizes a variation of system calls. On the other hand, if the distribution is biased toward certain system calls, such as in the case of restore, Table 10.4, GP utilizes a smaller set of system calls to construct the exploits.

10.2.4 Attacks Against the Markov Model

The box plots of exploit and attack anomaly rates against the Markov Model detector are detailed in Figures 10.18 and 10.19, respectively and the box plot of the exploit lengths is provided in Figure 10.20. The box plots show that the anomaly rates against traceroute have a wider range than the anomaly rates for the other applications. Similarly, the box plot of exploit lengths in Figure 10.20 indicates that GP favours longer exploits for restore, samba and ftpd, whereas in the case of traceroute, the exploit length varies over a greater range. As discussed in Sections 10.2.3 and 10.2.3, this can be attributed to the system call frequency distribution for traceroute in which the distribution is more even than for restore, samba and ftpd. Arguably, within the machine learning context, from the perspective of the characterization of search space, such an even distribution of the system calls requires GP to search a wider search space than the rest of the applications. For example, given the system call frequency distribution for restore, Table 10.4, GP can identify the set of system calls to use relatively easily since 99% of the system calls encountered during normal operation are `read`, `write` and `lseek` system calls. Compared to the case of traceroute, fewer system calls result in fewer combinations of system call sequences, which implies that, given a limited number of system calls, the effective size of the search space is smaller than the upper limit provided in Section 8.2.4.

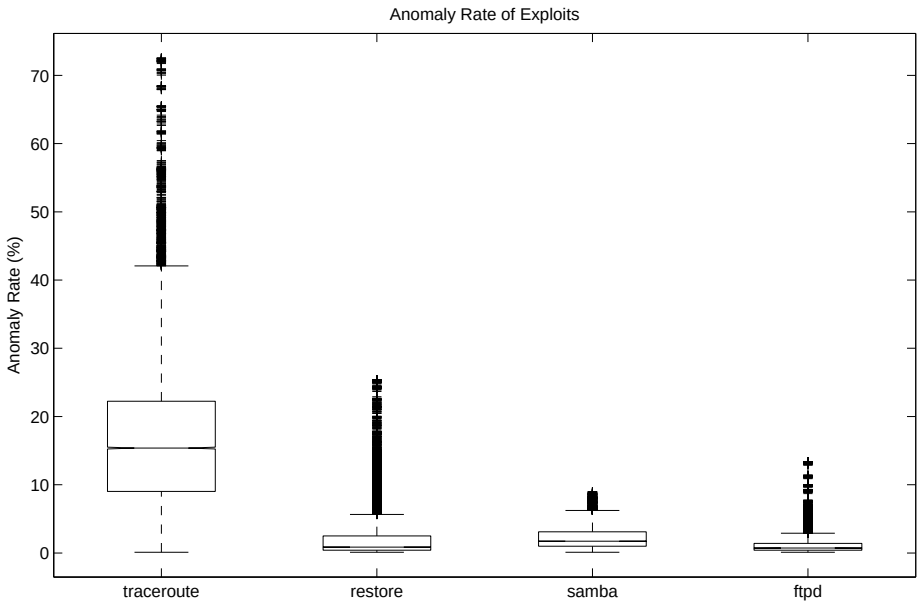

Figure 10.18: Box plot of the mimicry exploit anomaly rates for the Markov Model detector on traceroute, restore, samba and ftpd

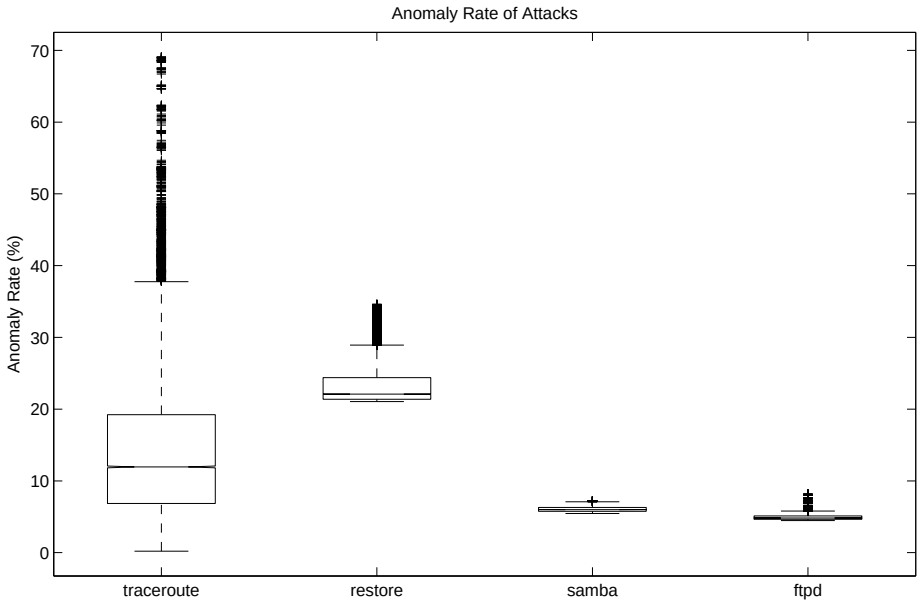

Figure 10.19: Box plot of the mimicry attack anomaly rates for the Markov Model detector on traceroute, restore, samba and ftpd

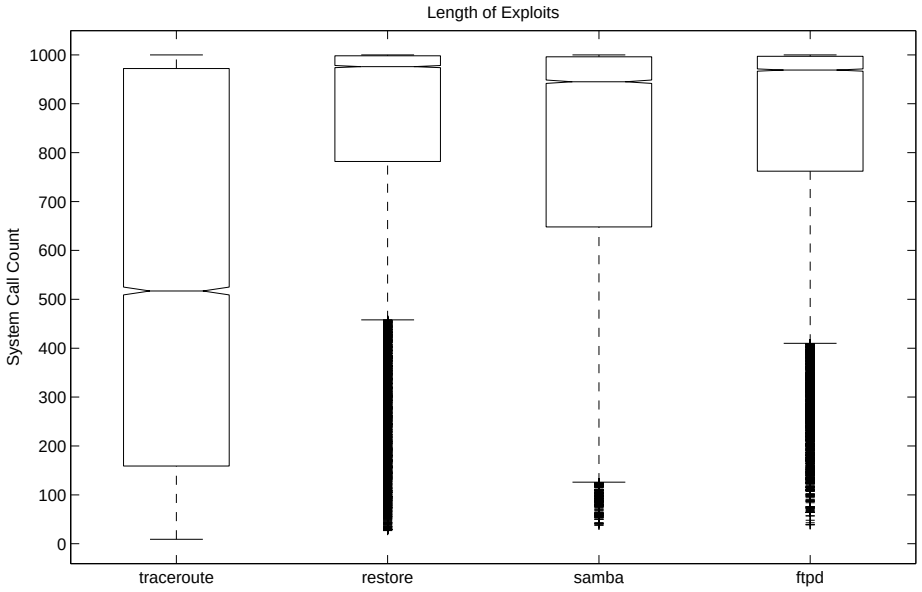

Figure 10.20: Box plot of the mimicry exploit lengths for the Markov Model detector on traceroute, restore, samba and ftpd

A Closer Look at GP-Generated Mimicry Attacks Against the Markov Model

The best mimicry exploit on traceroute is provided in Figures B.17 and B.18. System calls which perform I/O and memory access operations such as `gettimeofday`, `select`, `write` and `munmap` are utilized to hide the true intent of the code. As in the traceroute exploit against Stide, GP utilizes system calls related to timing.

The best exploit on the samba application against the Markov Model detector is provided in Figures B.19 and B.20. The exploit employs system calls involving file manipulation such as `_llseek`, `stat` and `fcntl64` with `munmap`, which deallocates memory. Sequences `_llseek munmap stat` and `_llseek munmap time` are repeated in various sections of the exploit. Similar to the samba attacks against Stide, pH and pHsm, the true intent of the code is hidden within numerous types of system calls related to I/O operations.

The best restore mimicry exploit against the Markov Model detector in Figures B.21 and B.22 shows that the exploit utilizes mainly file I/O system calls such as `open`, `read`, `write`, `rt_sigprocmask` and `lseek`. As opposed to the restore exploits generated against Stide, pH and pHsm, this exploit employs system calls which occur rarely during normal operation such as `rt_sigprocmask` and `lseek`. This is likely to be an effect of the shorter sliding window size of the Markov Model. Specifically, the longer sliding windows produce more sequences to be stored in the normal database due to the fact that longer sliding window patterns have more variations (Table 10.2). Therefore, against a smaller normal database, the search is more 'thorough' in the sense that rare system calls are utilized in the exploits as well.

As in the ftpd exploits against Stide, pH and pHsm, the best ftpd mimicry exploit against the Markov Model detector in Figures B.23 and B.23 utilizes a set of system calls performing I/O (i.e. `open`, `read`, `write`, `close`) to hide the true intent of the attack. Given an application which makes frequent `open` / `write` / `close` system calls, it is fairly difficult to detect an attack without monitoring the system call parameters and the outcome (i.e. the return values or error messages). It is important to note that the proposed approach generates system call parameters, but the anomaly detectors (employed in this thesis) do not employ them.

In summary, the exploits generated against the Markov Model detector share the same traits with the exploits against Stide, pH and pHsm mainly by employing system calls that are encountered frequently during the normal operation of the applications. However, given that the normal database is comparatively smaller than for the above-mentioned detectors, GP performs a more thorough search and, in addition to the frequent system calls, employs system calls which are encountered seldom during the normal operation of the applications.

10.2.5 Attacks Against the Neural Network

The box plots for the exploit and attack anomaly rates are provided in Figures 10.21 and 10.22, respectively, and the box plot for exploit lengths is provided in Figure 10.23. In the case of the Neural Network detector, the attacks on samba produced higher anomaly rates than the attacks on the remaining applications in Figure 10.21. A similar case was observed for traceroute and pH which seems to suggest that the application and detector combination affects the anomaly rates for mimicry attacks. In other words, pH provides a better defense against traceroute, whereas the Neural Network detector provides better defense on the samba application. Therefore, it is important for the defensive operations not only to identify the suitable parameters for the detectors but also to select the appropriate detector for the applications. In other words, one type of detector may not be suitable for all types of applications and furthermore the identification of suitable parameters can improve detector performance. The mimicry attack generation proposed in this thesis provides a suitable tool for determining the right detector for each application.

A Closer Look at GP-Generated Mimicry Attacks Against the Neural Network

The best traceroute mimicry exploit against the Neural Network Detector is provided in Figures B.25, B.26 and B.27. The traceroute attack against the Neural Network utilizes system calls which do not appear on the other traceroute exploits such as `recvfrom`, `poll`, `brk`. The `recvfrom` system call receives messages from a network socket whereas `brk` changes the data segment size of the process and `poll` provides

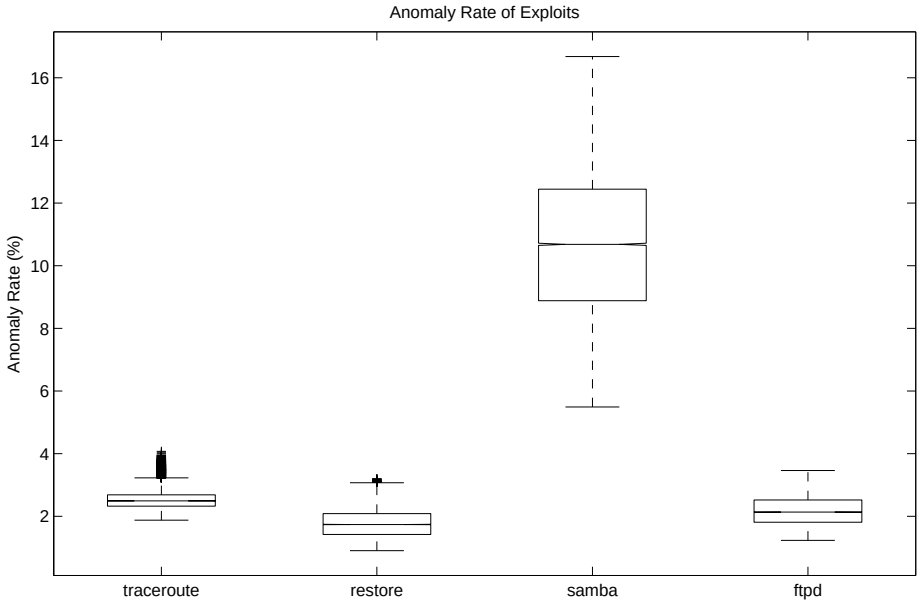

Figure 10.21: Box plot of the mimicry exploit anomaly rates for the Neural Network detector on traceroute, restore, samba and ftpd

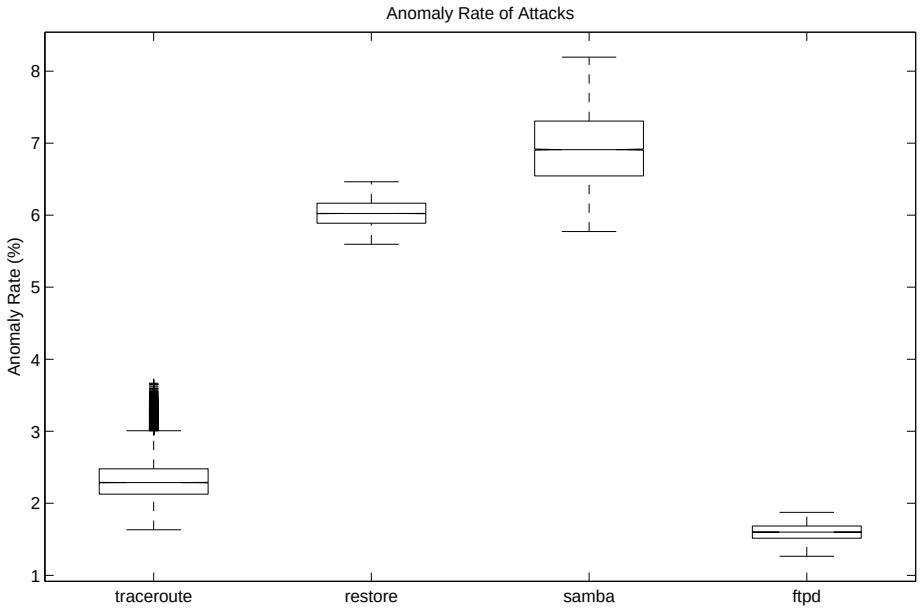

Figure 10.22: Box plot of the mimicry attack anomaly rates for the Neural Network detector on traceroute, restore, samba and ftpd

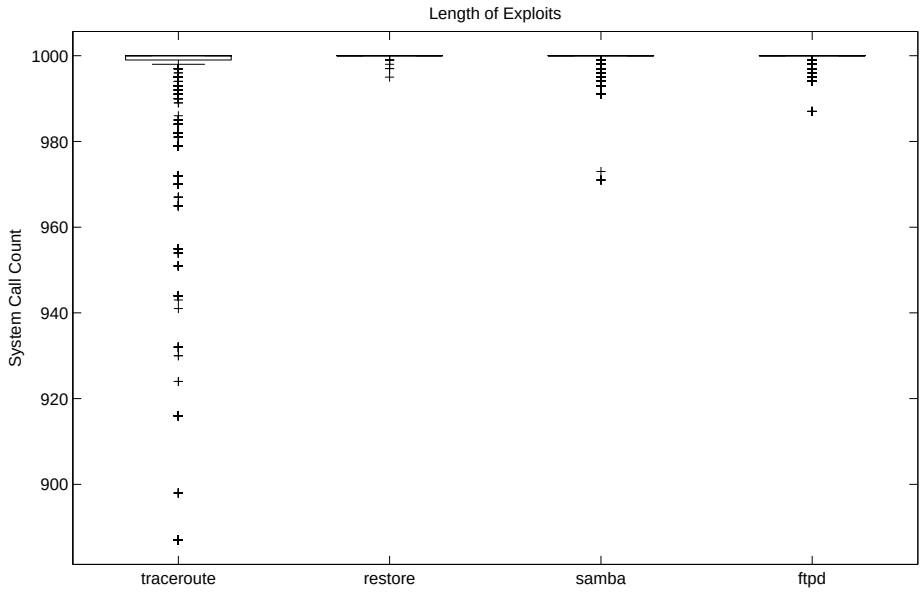

Figure 10.23: Box plot of the mimicry exploit lengths for the Neural Network detector on traceroute, restore, samba and ftpd

support for multiplexing several data streams. The use of different system calls indicates that GP employs different strategies to evade different detection methodologies.

The best samba mimicry exploit against the Neural Network detector is detailed in Figures B.28 and B.29. As in the traceroute case, the exploit contains system calls such as `getegid32` and `getsockopt` which are not utilized in the other samba attacks against other detectors.

The best restore mimicry exploit against the Neural Network detector in Figures B.30 and B.31 shows that the exploit uses file I/O system calls such as `open`, `write`, `close`, `fstat`, `fcntl` to hide the true intent along with numerous rare system calls such as `chown` and `unlink`. Unlike other exploits on restore, the `read` system call is rarely employed in this exploit, which indicates that GP employed a different strategy while evading the system call frequency distribution model of the Neural Network detector.

The best ftpd mimicry exploit against the Neural Network detector is provided in Figures B.32 and B.33. As with the other ftpd exploits against the other detectors, various memory and file I/O system calls were utilized with blocks of `close` system

calls, although there is no clear repeating pattern. However, the ftpd exploit against the Neural Network detector employs rare system calls as well, such as `brk`, which makes up 0.025% of all system calls executed during the normal operation of the ftpd application, Table 10.6.

Since the detection methodology which the Neural Network utilizes looks for a match in the frequency distribution of system calls in a trace, GP builds exploits which match the frequency distribution for which the detector seeks by utilizing rarely used system calls as well as the frequently used system calls.

10.2.6 Summary of the Analysis

The previous sections compared the mimicry exploits which GP generated for each detector, whereas this section focuses on summarizing the results of the previous sections and comparing the GP-generated exploits for each application. To this end, four summary tables are presented: Tables 10.7, 10.8, 10.9 and 10.10 for traceroute, restore, samba and ftpd, respectively. Each summary table details the characteristics of the five exploits (one against each detector) for the given application. Each attack is detailed with the following characteristics:

- **System Call Types (ST).** System calls on an IA32 architecture are organized according to the functions they perform [8]. The high level categories would include file system (file), inter-process communication (ipc), kernel (kernel), memory management (memory), network communication (network) and architecture-dependent system calls (arch). Therefore, ST fields in the summary tables provide the categories for the system calls. Appendix C details the system call categories and provides a list of system calls for each category.

- **System Call Indices (SI).** In Tables 10.3, 10.4, 10.5 and 10.6, which detail the frequency of system calls in the training data, each system call is assigned an index based upon its rank according to the frequency values. That is to say, the most frequent system call will obtain index number 1 and so on. Summary tables provide the minimum, maximum and median values for system call indices in the SI fields. Minimum and maximum values provide information on the most

and least frequent system call which GP used in the attack, respectively, whereas the median is intended to give the reader an idea of the spread of the indices. A better view of the spread of indices is provided in box plot analyses of the system call indices in Figures 10.24, 10.25, 10.26 and 10.27 for the exploits on traceroute, restore, samba and ftpd, respectively.

- **Unique System Call Count (SU).** Although the instruction sets in Tables 8.6, 8.8, 8.9 and 8.7 provide the 20 most frequently executed system calls for GP to use, GP is free to employ any subset of the instruction set to build the exploit. To this end, the SU fields in the summary tables provide the number of unique system calls in the exploit.

- **Repeating Pattern (RP).** In addition to the types and indices of system calls, another observable characteristic is how GP employs the system calls to build the exploits. The RP field in the summary tables discusses whether a pattern (a clear pattern or a combination of certain system calls) exists in the exploit.

- **Exploit Length (LN).** The LN fields in the summary tables provide the exploit lengths which was detailed previously in Table 8.20.

Table 10.7 summarizes the exploits on traceroute for the five anomaly detectors. It is apparent that GP employs different strategies to evade different detectors. For example, when evading Stide, GP employs a smaller subset of system calls and builds a shorter exploit. On the other hand, GP evades the Neural Network detector by employing a larger subset of system calls in which rare system calls are utilized along with frequent system calls. This is apparent in Figure 10.24 as well which details the spread of system calls (where lower indices indicate more frequent system calls).

Table 10.8 summarizes the mimicry exploits on restore. Compared with traceroute, GP utilizes a smaller set of system calls to build the exploit. This is not surprising since 99% of all system calls in the normal behaviour traces are `read` and `write` system calls. As with the traceroute exploits, GP utilizes different strategies to build the exploits where GP employs more types of system calls while generating exploits against the Markov Model and Neural Network detectors. As discussed in the

Table 10.7: An overview of the attacks generated by GP on traceroute

Target Detector	**Attack Characteristics**	
Stide	**ST:**	kernel, file, memory, network
	SI:	min: 1, med: 2, max: 9
	SU:	8
	RP:	Pattern (`gettimeofday sendto gettimeofday select write`) exists.
	LN:	34 system calls
pH	**ST:**	file, memory
	SI:	min: 2, med: 6, max: 14
	SU:	8
	RP:	None.
	LN:	118 system calls
pHsm	**ST:**	kernel, file, memory
	SI:	min: 1, med: 7, max: 10
	SU:	8
	RP:	Different combinations of `mmap` and `open`.
	LN:	1000 system calls
Markov Model	**ST:**	kernel, file, memory
	SI:	min: 1, med: 2, max: 14
	SU:	9
	RP:	Different combinations of `gettimeofday` and `write`.
	LN:	957 system calls
Neural Network	**ST:**	kernel, file, memory, network
	SI:	min: 1, med: 7, max: 22
	SU:	20
	RP:	None.
	LN:	1000 system calls

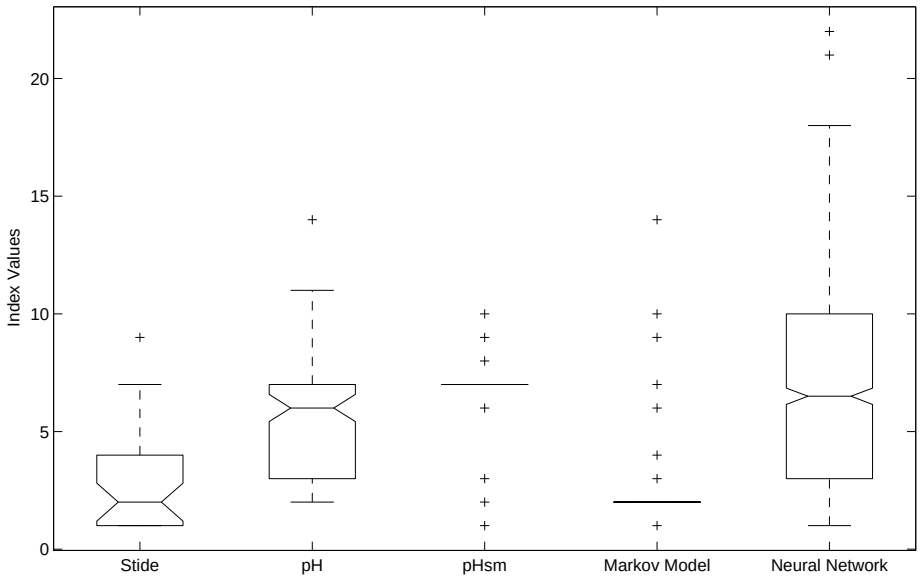

Figure 10.24: Box plot of the system call indices for the traceroute exploits

previous sections, this can be attributed to two factors. First, given that the Markov Model has a smaller normal database, GP explores infrequent system call sequences as well as frequent ones. Second, since the Neural Network detector monitors deviations from the normal system call frequency distribution, GP builds exploits to match the normal frequency distribution which implies that rare system calls are employed as well. Figure 10.25, which provides the box plot for the system call indices, provides further support for this argument.

The summary of the GP-generated exploits on samba is provided in Table 10.9. Compared to the traceroute and restore exploits, the samba exploits utilize rare system calls, although the number of unique system calls are comparable. As in the previous results, exploits against the Markov Model and Neural Networks utilize more types of system calls than the exploits against Stide and pH. Furthermore, the box plot demonstrates that GP employs more types of system calls against pHsm as well, Figure 10.26. This indicates that the implementation of a schema mask causes GP to employ a different strategy for building the exploits.

Table 10.10 summarizes the ftpd exploits, whereas Figure 10.27 shows the box plot analysis of the system calls. The exploit against Stide stands out since GP did

Table 10.8: An overview of the attacks generated by GP on restore

Target Detector	**Attack Characteristics**	
Stide	**ST:**	file
	SI:	min: 1, med: 1, max: 6
	SU:	4
	RP:	Different combinations of `read` and `write`.
	LN:	1000 system calls
pH	**ST:**	file
	SI:	min: 1, med: 1, max: 11
	SU:	6
	RP:	Different combinations of `read`, `write` and `lseek`. Large blocks of `write`.
	LN:	1000 system calls
pHsm	**ST:**	file, memory
	SI:	min: 1, med: 1, max: 6
	SU:	6
	RP:	Different combinations of `read`, `write` and `lseek`.
	LN:	999 system calls
Markov Model	**ST:**	kernel, file, memory
	SI:	min: 1, med: 2, max: 12
	SU:	8
	RP:	Different combinations of `read`, `write` and `lseek`.
	LN:	1000 system calls
Neural Network	**ST:**	kernel, file, memory
	SI:	min: 1, med: 5, max: 20
	SU:	19
	RP:	None.
	LN:	1000 system calls

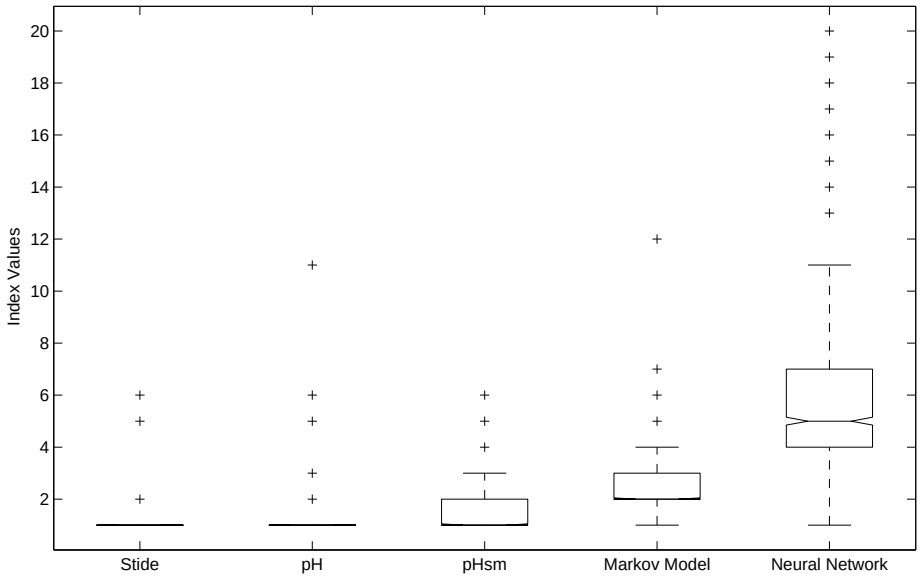

Figure 10.25: Box plot of the system call indices for the restore exploits

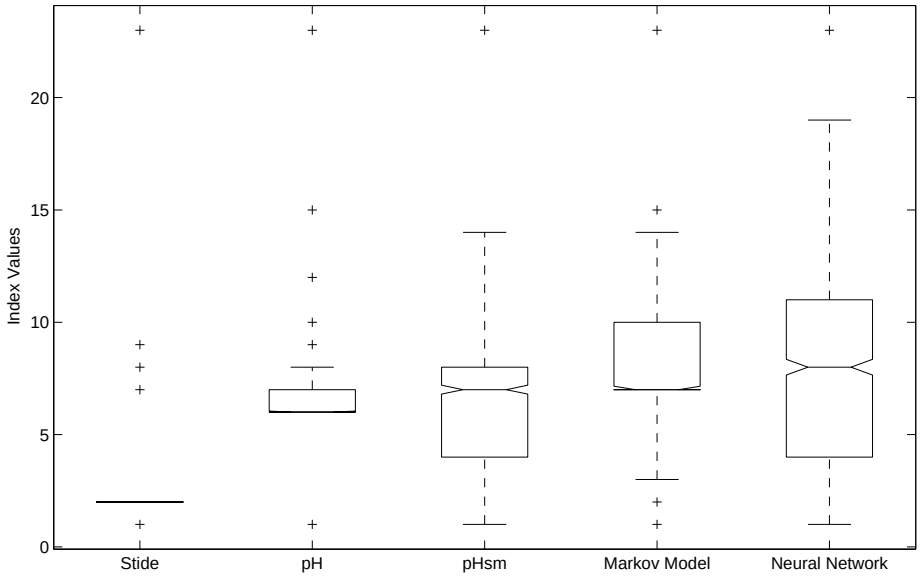

Figure 10.26: Box plot of the system call indices for the samba exploits

Table 10.9: An overview of the attacks generated by GP on samba

Target Detector	**Attack Characteristics**	
Stide	**ST:**	file
	SI:	min: 1, med: 2, max: 23
	SU:	6
	RP:	Different combinations of `read` and `lseek`. Large blocks of `llseek`.
	LN:	1000 system calls
pH	**ST:**	kernel, file, memory
	SI:	min: 1, med: 6, max: 23
	SU:	9
	RP:	Different combinations of `fcntl64`, `munmap` and `stat`. Long blocks of `write`.
	LN:	1000 system calls
pHsm	**ST:**	kernel, file, memory
	SI:	min: 1, med: 7, max: 23
	SU:	11
	RP:	None.
	LN:	1000 system calls
Markov Model	**ST:**	kernel, file, memory
	SI:	min: 1, med: 7, max: 23
	SU:	12
	RP:	Different combinations of `fcntl64`, `munmap` and `stat`.
	LN:	983 system calls
Neural Network	**ST:**	kernel, file, memory, network
	SI:	min: 1, med: 8, max: 23
	SU:	20
	RP:	None.
	LN:	1000 system calls

not employ the three most frequent system calls in this exploit. Furthermore, among the GP exploits analysed, the ftpd exploit against Stide is the shortest one. As with the results discussed in previous summary tables, the exploits against the Markov Model and Neural Network detectors utilize more types of system calls (i.e. a higher number of unique system calls) and utilize infrequent system calls in the exploits. As discussed previously, this is because in both detection methodologies rare system calls play a role as important as the frequent system calls.

Table 10.10: An overview of the attacks generated by GP on ftpd

Target Detector	**Attack Characteristics**	
Stide	**ST:**	kernel, file, network
	SI:	min: 4, med: 7, max: 16
	SU:	8
	RP:	None.
	LN:	11 system calls
pH	**ST:**	kernel, file
	SI:	min: 1, med: 5, max: 7
	SU:	5
	RP:	Different combinations of `open`, `read`, `write` and `close`. Long blocks of `close`.
	LN:	1000 system calls
pHsm	**ST:**	kernel, file, memory
	SI:	min: 1, med: 5, max: 11
	SU:	10
	RP:	Different combinations of `open`, `read`, `write` and `close`.
	LN:	994 system calls
Markov Model	**ST:**	kernel, file, memory
	SI:	min: 1, med: 4, max: 13
	SU:	10
	RP:	Different combinations of `open`, `read`, `write`, `close`, and `rt_sigaction`.
	LN:	1000 system calls
Neural Network	**ST:**	kernel, file, memory, network
	SI:	min: 1, med: 5, max: 20
	SU:	19
	RP:	None.
	LN:	1000 system calls

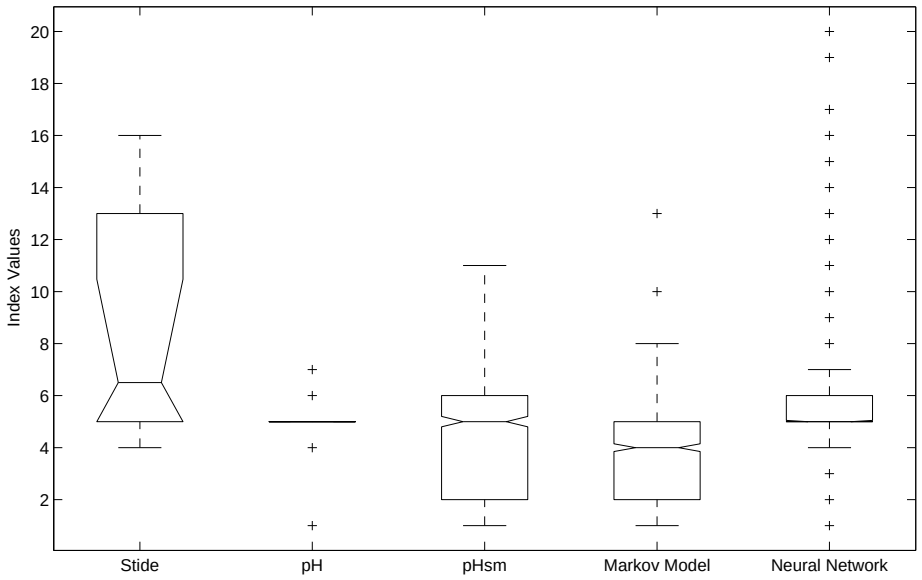

Figure 10.27: Box plot of the system call indices for the ftpd exploits

10.2.7 Discussion of the Analysis Results

The purpose of the analysis in this section is to investigate the effects of the vulnerable application in the resulting attacks. To this end, 100,000 attacks were generated against each detector (4 applications; 25000 for each application, given 50 runs per application and 500 attacks per run) which were analysed using box plots. Additionally the best mimicry attacks were analysed in order to identify the system calls which they employed to camouflage the attack.

The results indicate that the distribution of the system calls in the traces on which the detector is trained plays an important role in determining the patterns which the attack uses to hide the true intent of the attack. For example, attacks against restore commonly utilize blocks of `read` and `write` system calls which are executed frequently by restore during normal use. Some attacks employed repeating patterns, such as in the traceroute attack against Stide, whereas other attacks do not utilize a clear repeating pattern, as in the ftpd attack against Stide.

An interesting finding of the analysis is that the attacks generated against traceroute have higher anomaly rates, which seems to suggest that generating attacks against traceroute was harder. Given that the normal behaviour data structure and the number of system calls that traceroute utilizes are low compared to other applications, this indicates that there are other factors which affect the difficulty of generating mimicry attacks against an application. One possible reason behind this can be the frequency distribution of system calls. Given that the frequency distribution of system calls for traceroute is more balanced compared to those for restore, samba and ftpd, this may affect the difficulty of generating attacks against traceroute.

It is evident that, when attacks against different detectors are compared (on the same application), GP identifies an evasion strategy on a per detector basis. In other words, the system call types and frequency distributions in the exploits vary based upon the target detector. The common finding suggests that GP utilizes infrequent system calls to build exploits against the Markov Model and Neural Network detectors, whereas the exploits against the remaining detectors generally utilize a smaller set of system calls (i.e. the frequent system calls).

Furthermore, the normal behaviour of the application plays an important role in

determining what strategies GP will use under the 'black-box' assumption to evade detection. If the frequency distribution of system calls in normal behaviour is fairly even (i.e. distributed among a number of system calls), as with traceroute, GP aims to build exploits which share similar characteristics. On the other hand, if a small subset of system calls constitutes a substantial portion of the system calls made during normal operation, then GP identifies the frequent subset and builds the attacks accordingly. The analysis of the best 'black-box' mimicry attacks indicates that the attacks use system calls related to file I/O, memory allocation/deallocation and timing to hide the true intent of the attack. In the cases of samba and ftpd, the 'black-box' exploits utilized system calls related to network communications as well, which differs from the traceroute and restore exploits. This is expected since samba and ftpd are the only network-based applications among the four employed in this thesis.

An advantage of the proposed approach is that if 'white-box' access to the detector is not possible or feasible, analysis of the 'black-box' approach provides valuable insight into the normal behaviour model of the detector. That is to say, based upon the attacks which the proposed 'black-box' approach generates, it is possible to identify the system calls employed frequently by GP. Furthermore, the repeating system call sequences and the types of system calls which the proposed 'black-box' approach employs can help detector developers to identify a subset of inputs which reveals the shortcomings of the detector, hence making it easier to identify and eliminate the detector's vulnerabilities.

Chapter 11

Conclusion

Buffer overflow attacks are a common threat to computer systems in which the attackers exploit software vulnerabilities. Subsequently, attackers inject their malicious code and obtain control of the systems on which the vulnerable software is found. Commonly, the buffer overflow attack leads to various criminal activities such as vandalism, fraud, identity theft, intellectual property theft and many other types of crime. Buffer overflows become more critical, if the vulnerable software provides network services or runs in privileged mode as the vulnerability can be exploited from a remote location and provide super-user access to the system.

Realistically, one can assume that vulnerabilities exist even when the software is well written and well tested. Therefore, it is important to have defense mechanisms in place against buffer overflow attacks. To this end, the detection of buffer overflow attacks is possible using misuse detection or anomaly detection. However, no system is infallible, including the detectors. Thus, vulnerability testing of the detectors is crucial for identifying the vulnerabilities and ensuring that the detection methodology is adequate.

This thesis proposes a 'black-box' approach (which is based upon Evolutionary Computation) to generate buffer overflows automatically. The purpose of generating buffer overflow attacks automatically is to perform vulnerability testing on detectors which aim to detect buffer overflow attacks. If the test reveals an attack which can evade detection while deploying successfully, it implies that the detector is vulnerable.

The proposed approach can be considered as 'white-hat' hacking and contributes to computer security by making it easy to determine the limitations of the detectors and eliminate their vulnerabilities before the attacker can exploit them to evade detection. The proposed approach is employed against a misuse detector, namely Snort, and against numerous anomaly detectors, namely Stide, pH, pH with a schema mask

(pHsm), the Markov Model and the Neural Network. The aforementioned detectors utilize different detection methodologies to detect the stack overflow attacks and they provide suitable detection feedback, in the form of detection rates and delays which are employed to guide the search for the evasion attacks.

Based upon the access to the detector, the relevant work on detector vulnerability testing can be categorized in two groups: (1) 'white-box' approaches which assume the availability of internal knowledge of the detector [105] [99] [96] [98] [32] [56] [34] and (2) 'black-box' approaches which assume a more limited access to the detector [82], [73] [102] [48] [44] [49] [45] [46]. For example, in the case of an anomaly detector testing effort, a 'white-box' approach assumes that internal knowledge such as the detector parameters and normal behaviour database are available for use in the tests, whereas the 'black-box' approach assumes that the method of interaction with the detector is limited to the detector outputs, such as the anomaly rate which is returned by the detector as a part of its normal operation.

Within the context of detector vulnerability testing, both 'black-box' and 'white-box' detector vulnerability testing aim to implement search methods to develop evasion attacks which can evade detection by avoiding signatures or by altering the malicious behaviour to resemble 'normal' use. The number of distinct attacks which the search can yield comprises the search space. In other words, the search space signifies the number of candidate evasion attacks which the search method can generate. The search space depends upon parameters of the vulnerability testing such as the attack length, evasion methodology and the information available from the detector. A 'white-box' approach employs the additional knowledge to reduce the search space so that an exhaustive search can be performed, automatically or manually. On the other hand, the 'black-box' method employs the limited information from the detector to perform the search. Although performing the search with limited information presents a more difficult scenario, the 'black-box' approach can be applied to various detectors as long as they provide a similar means of interaction. From this perspective, the 'white-box' approaches are at a disadvantage since they do not generalize well to other detectors due to their detector specific search methods.

In particular, this thesis focuses on stack buffer overflows on Intel 32 bit architecture and employs Evolutionary Computation to evolve evasion attacks where the general objective is to produce attacks which can deploy without being detected by the target misuse and anomaly detectors. The Evolutionary Computation approach proposed in this thesis aims to improve various buffer overflow components using different evasion techniques.

Stack overflow attacks are suitable for establishing the effectiveness of the proposed approach. Not only their ground truth is widely available in the current literature but also they have been studied in the previous evasion attack research. The proposed approach relates to a wider scope of attacks where the attacker aims to alter the code he/she injects to evade detection.

Buffer Overflow Components. As discussed in Chapter 2, a common buffer overflow attack has three components: a NoOP sled, a shellcode and a block of approximated return addresses. Chapter 6 focuses mainly on developing suitable NoOP and return address components in which the objective of the attacker is to utilize shorter NoOP sleds to evade the misuse detectors. In order to accomplish this, the approximated address value in the return address component needs to be more accurate since utilizing fewer NoOPs implies that fewer return address values can cause the shellcode to deploy successfully. Chapter 7 focuses on the shellcode component and aims to evade misuse detectors by reordering the assembly instructions or by discovering other assembly instructions which can achieve the attack goals. Similarly, Chapter 8 focuses on shellcode but aims to generate attacks at the system call level by identifying system call sequences which can deploy the exploit while being recognized as normal by the anomaly detector.

Evasion Techniques. Among the various evasion techniques [100], those of interest to this thesis are obfuscation and mimicry attacks. The GE approach discussed in Chapter 6 sets the scene for an obfuscation evasion attack against Snort by improving the detectable characteristics of the malicious buffer other than the payload. Following such an optimization, the linear GP discussed in Chapter 7 obfuscates the shellcode by altering the ordering of the shellcode instructions so

that the signatures do not match with the shellcode, hence evading detection. On the other hand, when the evasion attacks are targeted toward the anomaly detectors which monitor application behaviour at the system call level, attackers can employ an evasion technique called a mimicry attack where the attackers alter their shellcode so that the attack is recognized as normal behaviour. To generate mimicry attacks, the linear GP with Pareto optimization discussed in Chapter 8 was employed to modify the 'core' attack to evade the anomaly detectors which monitor system call sequences.

A central theme in the approach taken in this thesis is the utilization of stochastic search processes, namely Grammatical Evolution and Genetic Programming, against a 'black-box' detector to automate the process of malicious code design. The specific emphasis was placed on three areas.

1. Identification of an appropriate instruction set from which the exploits are built is crucial. In this thesis, the exploits are built at the assembly (Chapters 6 and 7) and at the system call levels (Chapter 8).

2. Identification of appropriate goals is needed, where these can take two basic forms:

   (a) minimizing or eliminating detection. In the case of misuse detection (Chapters 6 and 7), this implies minimizing the number of alarms. On the other hand, in the case of anomaly detection (Chapter 8), this implies minimizing the anomaly rate.

   (b) matching key steps in establishing the 'core' exploit. In the case of misuse detection (Chapters 6 and 7) this implies having suitable NoOP, shellcode and return address components which can execute a system call. Whereas, in the case of anomaly detection (Chapter 8), the focus is on the shellcode, and as opposed to the assembly level case, the granularity is at a higher level, namely that of system calls.

3. Support for obfuscation, which in this case is a direct side effect of the stochastic search operators inherent in EC.

11.1 Contributions

As discussed earlier in this chapter, the general objective of this thesis is to provide an automated 'black-box' method for generating buffer overflow attacks for testing misuse and anomaly detectors. The contributions of this work are detailed below

1. **An Artificial Arms Race.** The proposed vulnerability testing approach represents an artificial arms race between attackers and detectors for the purpose of eliminating detector weaknesses. In this arms race, the attacker interacts with the target detector by utilizing the detector feedback (i.e. detection and anomaly rates, delays) to build evasion attacks, which can evade detection while achieving the objectives of the attacker. If the attacker can deploy an evasion attack while remaining undetected, it indicates that the detector is susceptible to evasion attacks. Thus, the defenders analyse the evasion attacks and eliminate the weaknesses of the target detector. A future extension of the arms race is to utilize adaptive detection methods and facilitate a co-evolution between the attackers and the adaptive detectors in which the objective is to build detectors which are robust against evasion attacks, as discussed in Section 11.4.

2. **Access to the Detector.** The proposed approach assumes a 'black-box' approach, where the input from the detector is limited to the anomaly rate in anomaly detection scenarios and detection information in misuse detection scenarios. Assuming a 'black-box' access to the detector is more suitable for testing both open and closed source detectors where the developer is unable or unwilling to share the design details and internal data structures with the tester. In 'white-box' testing, the test methodology is biased heavily upon the internal knowledge employed to build the abstractions and to facilitate the test [81]. This implies that 'white-box' approaches tend to be detector specific and may not work on other detectors when the required 'white-box' information is not available. Moreover, the proposed 'black-box' approach can be employed for detector parameterization as well, where the objective is to identify suitable deployment parameters such as the sliding window length and locality frame count to provide effective detection.

3. **Analysis of 'Normal Behaviour'.** In the case of evading anomaly detectors using mimicry attacks, the attacker employs the 'normal behaviour' model in a search, where the objective is to locate a subset of the model which allows the attacker to perform a malicious action. Given that normal behaviour has an impact on the search space, the analysis of normal behaviour is of interest. Within this context, this thesis provides a methodology for the analysis of the training set sensitivity for the anomaly detectors. Furthermore, given two applications which are monitored by the same detector, the normal behaviour model of an application (i.e. the normal behaviour database size, the number of system calls and the distribution of system calls) can have an impact on the difficulty of generating attacks against that application. To this end, this thesis provides a methodology for the analysis of the normal behaviour for four UNIX applications and the corresponding normal behaviour databases in the anomaly detectors. Such analysis is important for identifying the elements in the normal behaviour model which the attackers are likely to exploit such as open-write-close sequences. As a result, additional measures can be built around these unsafe elements to prevent evasion attacks in advance.

4. **Evaluation of the Attacks.** As discussed earlier, in the case of evading anomaly detectors using mimicry attacks, buffer overflow attacks have two stages, the preamble and the exploit. The preamble may have substantial effects on the anomaly rate of the attack if it is long and anomalous. Therefore, in this thesis, the anomaly rate is calculated for the attacks (i.e. preamble and exploit), whereas the previous work on mimicry attacks [105] [99] [96] [98] [32] [56] [34] reported anomaly rates for the exploits alone. Even if an exploit were to raise no alarms, the anomalies caused by the break-in attempt (i.e. the preamble) may cause anomalies, which – depending upon the attack and the vulnerable application – can become obvious [46] [47]. Furthermore, the transition from preamble to exploit may also produce anomalous behaviour [98] [47]. Therefore, including preambles and employing additional metrics such as delays and exploit lengths provides a better perspective for detection. Basing the vulnerability testing upon these additional metrics potentially can reveal future

attacker trends such as deploying longer exploits or distributing the anomalies to minimize delays.

5. **Analysis of the Attacks.** In the case of evading anomaly detectors using mimicry attacks, the attacks provide a valuable source of data which can be utilized as a guideline for improving anomaly detectors or for developing new detection techniques which are robust in preventing mimicry attacks. To this end, this thesis tests mimicry attacks which are trained on a detector, against numerous other anomaly detectors. The mimicry attack which can evade numerous detectors presents a critical security flaw which needs to be addressed by detector developers. The set of attacks generated can be utilized as well to create attack datasets which the researchers can employ to build detectors which are robust against evasion attacks. Furthermore, this thesis discusses the search space sizes for both 'black-box' and 'white-box' approaches. Search space analysis provides an upper limit on the number of candidate solutions (i.e. the number of attacks), given the problem definition.

6. **Multi-objective Optimization.** In mimicry attacks, besides the anomaly rate of the exploit, there are various characteristics which affect the success of the attack. For example, even if an exploit raises no alarms, the anomaly rate from the preamble should be included as well in order to calculate the anomaly rate of the attack. Furthermore, in the case of a long and anomalous preamble, the attacker can increase the length of the exploit to reduce the anomaly rate. Since the anomaly rate is calculated continuously as the attack progresses, appending a low or zero anomaly exploit will reduce the total anomaly rate in time. Furthermore, the experiments on pH indicated that delay is an effective method for stopping the attacks, therefore the attacker should aim to generate attacks which will keep the locality frame count low. Various other factors, such as the vulnerable buffer size, the target detector and the nature of the exploit may introduce new objectives/constraints which the attacker needs to consider. By employing Evolutionary Computation with multi-objective optimization support vulnerability testing efforts can investigate how different

factors affect the success of the evasion attacks. In turn, the detectors can be improved accordingly to be sensitive to attack attributes beyond the anomaly rate.

11.2 Discussion of Results

As discussed in Chapter 2, a typical stack buffer overflow attack consists of three components:

1. the NoOP sled;
2. the shellcode, which executes the system calls which achieve the attack goals;
3. the approximated return addresses which overwrite the EIP and divert the execution to the NoOP sled or the first instruction of the shellcode.

In order to investigate thoroughly the task of generating buffer overflow attacks automatically using Evolutionary Computation, the following characteristics of the stack buffer overflow attacks were considered for optimization:

1. the length of the NoOP sled and the accuracy of the approximated return addresses (Chapter 6);
2. shellcode design at the assembly level (Chapter 7);
3. shellcode design at the system call level (Chapters 8, 9 and 10).

To address the improvements of the above-mentioned characteristics separately, three frameworks were introduced. In the first framework, utilizing Grammatical Evolution in Chapter 6, the goal was to modify certain attack characteristics to minimize chances of detection by a misuse detector. Particularly, a block of NoOP instructions (i.e. the NoOP sled) presents a detectable pattern, therefore from an attacker's perspective, an attack with a shorter NoOP sled is desirable. However, in order to be able to use shorter NoOP sleds, the attack needs to make more accurate estimations of the return address. Therefore, the objective of the first framework was

to evolve malicious buffers which could deploy successfully with smaller NoOP blocks and more accurate return address approximations.

The results of Chapter 6 in optimizing buffer overflow characteristics to evade misuse detectors demonstrated that the framework can discover NoOP length and return address combinations which can evade the Snort misuse detector. A close look at the detector logs indicated that the attacks were detected by a signature which looks for long blocks of NoOP instructions, hence the best GE attack, which could deploy with only one NoOP instruction, could evade detection easily. The experiments in Chapter 7 utilized niching to encourage diversity in the population. In other words, the individuals were rewarded for crafting buffer overflows with different NoOP sled sizes and different numbers of approximated return addresses. In particular, three sets of experiments were conducted: (1) Basic GE, (2) GE with niching, and (3) GE with niching and NoOP minimization, where the successful individuals were awarded additional points for minimizing the NoOP sled length. In summary, the results indicated that GE can optimize the NoOP sled length and return address accuracy to bypass the Snort signature that monitors long sequences of NoOP instructions.

Having established the effectiveness of the proposed GE framework for evolving the NoOP sled and the return addresses, a second framework was developed in Chapter 7 which focused on evolving the payload (i.e. the shellcode) of the attack to evade misuse detectors and to investigate different ways for achieving the attack objectives. In this case, the goal of the framework was to design the payload at the assembly level in order to obfuscate the attack in such a way that it was undetectable by the misuse detectors. The second framework employed linear Genetic Programming because GE proved inefficient in modifying instructions due to its representation scheme. In particular, a single genome change near the beginning of the individual in GE causes the remaining genomes to be mapped differently, hence causing major changes in the program it produces. On the other hand, in linear GP, change in a genome causes that particular instruction to change, which provides the framework with a more efficient search method.

The results in evolving buffer overflow attacks against misuse detectors in Chapter 7, demonstrate that the code bloat property of GP provides a means of hiding the

true intent of the attack by mixing the exploit instructions with the introns which do not contribute explicitly to the functional properties of the attack. Furthermore, the results indicated that GP is effective at finding alternate ways of achieving attack objectives, hence implementing the core attack with different instructions. The experiments focused on utilizing different instruction sets which consisted of basic control instructions, arithmetic instructions and logic instructions. Moreover, this research investigated the use of an additional objective for improving the likelihood of the execution of the attack where the GP was encouraged to place the attack instructions as close to the end of the buffer as possible. In this way, even though the execution jumps into somewhere other than the first instruction of the shellcode, the attack can still be deployed successfully. The resulting attacks were passed through the Snort misuse detector where the detector contained the most recent shellcode signatures which could detect certain sub-goals of the attack, such as pushing `'/bin/sh'` to the stack. By utilizing different instructions and reordering the sub-goals, the attacks generated by GP evaded detection by Snort.

Sharing a similar objective with Chapter 7, Chapter 8 approaches the task of evolving the payload at a higher level. Having established that linear GP can evolve payload at the assembly level in Chapter 7, the aim of the third framework was to evolve the payload at the system call level. System calls facilitate the interaction of UNIX applications with the operating system kernel, therefore anomaly detectors monitor the system calls which the applications make to detect any deviations from established normal behaviour. Therefore, the objective of the framework was to discover sequences of system calls which could deploy the attack successfully while conforming to the normal behaviour definition of the detector. The resulting attacks are called mimicry attacks since they mimic normal behaviour. Most previous work on mimicry attacks [105] [99] [96] [98] [32] [56] [34] assumed a 'white-box' access to the detector, which implies that the attacker has access to the normal database and internal data structures of the detectors. This resulted in the employment of various exhaustive search methods on the normal database of the target detector. On the other hand, the proposed 'black-box' approach aimed to evolve mimicry attacks based only upon the anomaly rate returned by the detector. The proposed approach

demonstrated that evading detectors do not require an in-depth understanding of the target detector. A 'black-box' technique is feasible as long as the evasion attacks can be suitably parameterized.

The experiments on evolving mimicry attacks against anomaly detectors in Chapter 8 consisted of five different anomaly detectors, namely the Stide, pH, pH with a schema mask (pHsm), Markov Model and Neural Network detectors monitoring the traceroute, restore, samba and ftpd applications. The results demonstrated that linear GP succeeds in reducing the anomaly rate of the attacks by utilizing just the anomaly rate returned by the detector. This implies that although it represents a substantially more difficult problem, limiting the knowledge of the detector to the 'black-box' level does not prevent the identification of attacks which are equally effective as a 'white-box' model for the same detector. Assuming a 'black-box' access extends the application of vulnerability testing beyond the cases where internal knowledge of the detector is required. The results showed that standard EC methods can succeed in finding evasion attacks, which can largely circumvent the detection techniques that are either in wide use or have been studied extensively in the literature. Furthermore, the proposed approach should generalize to numerous other computer defenses. Furthermore, mimicry attacks have various characteristics which the attacker needs to consider, such as attack success, anomaly rate of the attack, attack length and delays which the detector imposes on the attack. Therefore, Pareto ranking was employed in the experiments detailed in Chapter 8 to facilitate multi-objective optimization.

Furthermore, it is apparent that evading anomaly detectors can be more difficult than previously believed [105] [99] [96] [98] [32] [56] [34] due to the attacker's lack of control over the system calls which execute before the attacker's shellcode is invoked. Previous work [105] [99] [96] [98] [32] [56] [34] reported anomaly rates on the exploit alone without considering the anomaly rate for the preamble. The results in Chapter 8 established that even if the attacker created an exploit with a 0% anomaly rate, the corresponding attacks, which included the anomalies from the preamble, would be more than 0%. The effects of the preamble are magnified especially if the preamble is long and anomalous. Moreover, within the scope of this framework, this thesis investigated the importance of training sets in anomaly detectors, where the results

indicated that the anomaly rates of the detectors are fairly sensitive to the training data employed for establishing the normal behaviour database.

Additionally, the experiment results in Chapter 8 demonstrated that a delay associated with locality frame counts is an effective way to stop an attack. Even if the attack achieved low anomaly rates, it can be frozen in effect if the anomalies were clustered together. In particular, mimicry attacks against samba have low anomaly rates yet the delays associated with them are billions of centuries in length. Therefore, by incorporating the system call parameters in detection and employing metrics to measure the dispersion of anomalies like the locality frame count in pH, the detectors can be more robust against mimicry attacks. Nevertheless, should the delays associated with the locality frame count be employed in the real-world, reducing false positives in the detector deserves further attention since legitimate behaviour which is unknown to the detector could cause substantial delays.

Furthermore, Sections 6.2.4, 7.2.4 and 8.2.4 provide discussions on search space sizes. The discussions indicated that evolving mimicry attacks at the system call level has the largest search space size for two reasons. First, the GP individuals representing mimicry attacks are longer than those representing malicious buffers and assembly level shellcode. Second, evolving mimicry attacks at the system call level requires more instructions, hence increasing the possible values which an instruction can take. Section 9.3.6 provides a discussion of the 'white-box' search space and compares it with the 'black-box' search space (Section 8.2.4). The comparison reveals that a common aspect of the previous 'white-box' approaches [105] [99] [96] [98] [32] [56] [34] was to employ the 'internal' detector knowledge to reduce the search space so that an exhaustive search could be performed. On the other hand, even though the proposed approach adopts a 'black-box' methodology, it succeeds in generating mimicry attacks which are comparable to 'white-box' mimicry attacks without employing any 'internal' detector knowledge available under a 'white-box' assumption.

In Chapter 9, the mimicry attacks generated in Chapter 8 were employed in an analysis which extends the experiments detailed in Chapter 8 in two ways. Firstly, as opposed to deploying the mimicry attacks only against the detector on which they were trained, the mimicry attacks were deployed against all five anomaly detectors.

Such an analysis aimed to investigate whether the mimicry attacks could generalize to the detectors for which they were not trained. The results indicated that, to a certain extent, mimicry attacks can produce low anomaly rates when they are deployed against other detectors, especially if the detection mechanism is similar and if the detector on which they were trained employed a longer sliding window length than the detector on which they were tested.

Secondly, Chapter 9 makes a comparison between the mimicry attacks which are generated by the proposed 'black-box' GP approach and the 'white-box' exhaustive search approaches, which the relevant work is based upon. The results indicate that although a 'black-box' access formulates a more difficult problem due to limited feedback from the detector, the resulting attacks provide anomaly rates comparable to the anomaly rates of the 'white-box' attacks. Moreover, it is important to note that separate 'white-box' exhaustive search approaches need to be developed for each detector since 'white-box' assumptions make use of the internal structures of the detector, which are very likely to be specific to the detector at hand. On the other hand, the proposed 'black-box' approach has the considerable advantage of being able to deploy against all five anomaly detectors without any changes.

Chapter 10 focuses on the analysis of the vulnerable applications which were employed in generating mimicry attacks in this thesis. The purpose of the analysis was to identify the characteristics of each application in terms of normal database lengths, number of system calls and the distribution of system calls which they utilize during normal operation. The results demonstrate that applications have different characteristics. For instance, applications providing service over a network utilize more system calls, which may provide the attacker with a wider normal behaviour within which to hide malicious intentions. Furthermore, the distribution of system calls varies since some applications use a smaller set of system calls very frequently. Consequently, the attacks generated by the proposed approach employ different techniques and system calls while hiding the true intent of the attack.

11.3 Guidelines for Detector Research

Mimicry attacks have various characteristics beyond anomaly rates which can affect their success. Even if an attack has a zero anomaly rate exploit, if the preamble is lengthy and produces several anomalies, it can be detected easily. Furthermore, even if both the preamble and the exploit raise very few alarms, if the anomalies are clustered together they can cause an increase in the locality frame count, hence effectively 'freezing' the attack. Another method which an attacker can employ to minimize the anomaly rate of an attack with a detectable preamble is to employ a longer exploit which raises fewer or no alarms, hence injecting more normal behaviour to obfuscate the anomalies. In the light of these observations, mimicry attack and vulnerability testing research should move from focusing on the anomaly rate alone to incorporating multiple characteristics such as the preamble and exploit length, locality frame counts and the associated delays

The experiments discussed in this thesis indicate that a better detection methodology is clearly necessary and several developments are possible. The current detectors have failed because they make the assumption that the sequence of system calls provides a suitable discriminator between normal and attack behaviours. This is clearly the wrong behavioural trait. Instead, the sequence of objectives which are used to establish a valid attack during the automation of mimicry attacks can be employed for the detection as well. That is to say, critical activities for detecting exploits are associated with attempts to access the password file or other critical system resources. A state machine capable of detecting this behaviour has a very simple representation and makes use of system call arguments to check the intention of the three critical commands (open-write-close). Consequently, the arguments of the system calls can be employed to distinguish between malicious and 'normal' behaviours. A goal for future research might be to provide a method for automating the design of such detectors and associating them with critical (that is privileged) computing system resources.

An alternative to the current state-of-the-art detection methodologies would be to move away from associating detectors with applications toward associating detectors with the objectives of an attack. In such a setting, mimicry attack generation would take the form of conducting searches for command sequences which provide

alternative means for establishing specific goals (the information hiding aspect not being as important because the detector is designed to ignore the vast majority of the instruction sequence, i.e. normal behaviour) and then a detector would take the form of modelling state spaces associated explicitly with compromising a target resource. Such a scheme would be more scalable than current approaches which concentrate on patching detecting in applications. Moreover, the detector would be able to play a more proactive role by predicting which step in the attack might appear next.

11.4 Future Research Directions

The proposed approach can be utilized in numerous ways to improve the current state-of-art in intrusion detection. First, it is a valuable vulnerability testing tool that can evolve attacks against a detector while employing a 'black-box' assumption. The detector developers can analyse the detection rate of the evolved attacks to identify any potential detector vulnerabilities. In a way, the proposed approach works to predict the possible variants of current vulnerabilities/attacks with a view to creating vaccines to the predicted attacks before they occur. The benefit would be that the defenders would no longer have to be one step behind the attackers.

The second and the more promising application of the proposed approach is to formulate an 'arms race' between attackers and detectors for different types of attacks as well. Moreover, adaptive detection methods will be employed to facilitate a co-evolutionary arms race where the attackers are rewarded as they defeat the detectors and, similarly, the detectors are rewarded as they adapt and 'learn' to detect the evasion attacks. Such an arms race will not only allow the defenders to identify the detector weaknesses but also enable the detectors to generalize beyond recognizing only a single instance of an attack hence freeing the detectors from working in a purely reactive manner.

Such an arms race can contribute to the security research mainly in three ways. First, by building robust detectors, which generalize beyond recognizing the specific instance of a single attack, variants of a core attack can be detected long before the variants are encountered. Second, detectors will be easier to update and maintain

since the arms race provides a modular detector design where a community of detectors provides coverage for numerous variants of a core attack. Third, the resulting evasion attacks can provide an attack database to the research community, which can be employed in detector vulnerability testing to determine detector blind spots.

An analogy can be drawn from flu vaccination research. Every year, researchers try to anticipate the possible strains of the flu virus which are most likely to affect people in the following flu season by analysing the flu viruses which have been seen up to that point in time. Based upon this analysis, predictions are made and flu vaccines are prepared for the following season. An arms race between attackers and detectors could enable researchers to follow a similar intervention against the attackers in order to strengthen the defense mechanisms for computer security.

Bibliography

[1] Julia Allen, Alan Christie, William Fithen, John McHugh, Jed Pickel, James Ellis, Eric Hayes, Jerome Marella, and Bradford Willke. State of the practice of intrusion detection technologies. Technical report, Carnegie Mellon Software Engineering Institute, 2000.

[2] SecurityFocus Vulnerability Archives. Lbnl traceroute heap corruption vulnerability. `http://www.securityfocus.com/bid/1739`, Last accessed June 2008.

[3] SecurityFocus Vulnerability Archives. Redhat linux restore insecure environment variables vulnerability. `http://www.securityfocus.com/bid/1914`, Last accessed June 2008.

[4] SecurityFocus Vulnerability Archives. Samba 'call_trans2open' remote buffer overflow vulnerability. `http://www.securityfocus.com/bid/7294`, Last accessed June 2008.

[5] SecurityFocus Vulnerability Archives. Wu-ftpd remote format string stack overwrite vulnerability. `http://www.securityfocus.com/bid/1387`, Last accessed June 2008.

[6] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[7] Leonard E. Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970.

[8] Konstantin Boldyshev. List of linux/i386 system calls. `http://asm.sourceforge.net/syscall.html`, 2000.

[9] Julien Budynek, Eric Bonabeau, and Ben Shargel. Evolving computer intrusion scripts for vulnerability assessment and log analysis. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1905–1912, New York, NY, USA, 2005. ACM.

[10] Bulba and Kil3r. Bypassing stackguard and stackshield. Phrack Online Magazine, Volume 0x05, Issue 0x10, January 2000.

[11] John M. Chambers, William S. Cleveland, and Paul A. Tukey. *Graphical methods for data analysis*. Wadsworth, 1983.

[12] Steve Christey and Robert A. Martin. Vulnerability type distributions in cve. MITRE Website `http://cwe.mitre.org/documents/vuln-trends/index.html`, May 2007.

[13] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[14] Carlos A. Coello Coello. Evolutionary multi-objective optimization: a historical view of the field. *Computational Intelligence Magazine, IEEE*, 1(1):28–36, Feb. 2006.

[15] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium, 1998*, pages 5 – 5, Berkeley, CA, USA, 1998. USENIX Association.

[16] Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 235–248, New York, NY, USA, 2005. ACM.

[17] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *Proceedings of the 17th Usenix Security Symposium, San Jose, CA*, July 2008.

[18] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[19] Kalyanmoy Deb and David E. Goldberg. An investigation of niche and species formation in genetic function optimization. In *Third international conference on Genetic algorithms*, pages 42–50, 1989.

[20] Dorothy E. Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[21] Maura A. Van der Linden. *Testing Code Security*. Auerbach Publications, Boston, MA, USA, 2007.

[22] Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer Superbus Von Underduk. Polymorphic shellcode engine using spectrum analysis. Phrack Online Magazine, Volume 0x0b, Issue 0x3d, August 2003.

[23] Gerry Dozier, Douglas Brown, Haiyu Hou, and John Hurley. Vulnerability analysis of immunity-based intrusion detection systems using genetic and evolutionary hackers. *Appl. Soft Comput.*, 7(2):547–553, 2007.

[24] Agoston E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[25] Jon Erickson. *Hacking: The Art of Exploitation*. No Starch Press, 2003.

[26] Henry Hanping Feng, Jonathon T. Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P. Miller. Formalizing sensitivity in static analysis for intrusion detection. *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 194–208, May 2004.

[27] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 62, Washington, DC, USA, 2003. IEEE Computer Society.

[28] Carlos M. Fonseca and Peter J. Fleming. Genetic algorithms for multiobjective optimization: Formulation discussion and generalization. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 416–423, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[29] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. The evolution of system-call monitoring. *Annual Computer Security Applications Conference, 2008. ACSAC 2008*, pages 418–430, Dec. 2008.

[30] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 120, Washington, DC, USA, 1996. IEEE Computer Society.

[31] James C. Foster, Vitaly Osipov, and Nish Bhalla. *Buffer Overflow Attacks*. Syngress Publishing, 2005.

[32] Debin Gao, Michael K. Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 318–329, New York, NY, USA, 2004. ACM.

[33] Debin Gao, Michael K. Reiter, and Dawn Song. Behavioral distance measurement using hidden markov models. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection - RAID, Lecture Notes in Computer Science*, LNCS 4219, pages 19–40, 2006.

[34] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Automated discovery of mimicry attacks. In *Recent Advances in Intrusion Detection, 9th International Symposium, RAID 2006*, volume 4219 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2006.

[35] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989.

[36] Sang-Jun Han and Kyung-Joong Kim Sung-Bae Cho. Evolutionary learning program's behavior in neural networks for anomaly detection. In *Neural Information Processing*, ICONIP 2004, LNCS 3316, pages 236–241. Springer Berlin / Heidelberg, 2004.

[37] Malcolm I. Heywood and A. Nur Zincir-Heywood. Dynamic page based crossover in linear genetic programming. *Systems, Man, and Cybernetics, Part B, IEEE Transactions on*, 32(3):380–388, Jun 2002.

[38] Hajime Inoue and Anil Somayaji. Lookahead pairs and full sequences: A tale of two anomaly detection methods. In *Proceedings of the 2nd Annual Symposium on Information Assurance (Academic track of the 10th NYS Cyber Security Conference)*, June 2007.

[39] Intel Corporation. *IA-32 Intel, Architecture Software Developer's Manual Volumes 2A, 2B: Instruction Set Reference, A-M, M-Z*, 2005.

[40] Nathalie Japkowicz, Catherine Myers, and Mark Gluck. A novelty detection approach to classification. In *In Proceedings of the Fourteenth Joint Conference on Artificial Intelligence*, pages 518–523, 1995.

[41] Josh Jones and Terry Soule. Comparing genetic robustness in generational vs. steady state evolutionary algorithms. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 143–150, New York, NY, USA, 2006. ACM.

[42] Cem Kaner. *Testing Computer Software*. TAB Books, Blue Ridge Summit, PA, USA, 1988.

[43] Dae-Ki Kang, Doug Fuller, and Vasant Honavar. Learning classifiers for misuse and anomaly detection using a bag of system calls representation. *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*, pages 118–125, June 2005.

[44] Hilmi Güneş Kayacık, Malcolm Heywood, and Nur Zincir-Heywood. On evolving buffer overflow attacks using genetic programming. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1667–1674, New York, NY, USA, 2006. ACM.

[45] Hilmi Güneş Kayacık, Malcolm I. Heywood, and A. Nur Zincir-Heywood. Evolving buffer overflow attacks with detector feedback. In *Applications of Evolutinary Computing, EvoWorkshops 2007: EvoCoMnet, EvoFIN, EvoIASP,EvoINTERACTION, EvoMUSART, EvoSTOC and EvoTransLog*, volume 4448 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2007.

[46] Hilmi Güneş Kayacık and A. Nur Zincir-Heywood. On the contribution of preamble to information hiding in mimicry attacks. In *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, pages 632–638, Washington, DC, USA, 2007. IEEE Computer Society.

[47] Hilmi Güneş Kayacık and A. Nur Zincir-Heywood. Mimicry attacks demystified: What can attackers do to evade detection? In *PST '08: Proceedings of the 2008 Sixth Annual Conference on Privacy, Security and Trust*, pages 213–223, Washington, DC, USA, 2008. IEEE Computer Society.

[48] Hilmi Güneş Kayacık, A. Nur Zincir-Heywood, and Malcolm Heywood. Evolving successful stack overflow attacks for vulnerability testing. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 225–234, Washington, DC, USA, 2005. IEEE Computer Society.

[49] Hilmi Güneş Kayacık, A. Nur Zincir-Heywood, and Malcolm I. Heywood. Automatically evading ids using gp authored attacks. *Computational Intelligence in Security and Defense Applications, 2007. CISDA 2007. IEEE Symposium on*, pages 153–160, April 2007.

[50] Hilmi Güneş Kayacık and Nur Zincir-Heywood. Generating representative traffic for intrusion detection system benchmarking. In *CNSR '05: Proceedings of the 3rd Annual Communication Networks and Services Research Conference*, pages 112–117, Washington, DC, USA, 2005. IEEE Computer Society.

[51] Richard Kemmerer and Giovanni Vigna. Intrusion detection: a brief history and overview. *Computer*, 35(4):27–30, Apr 2002.

[52] Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5):35–42, 1997.

[53] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[54] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley Publishing Inc., 2004.

[55] Mark A. Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE Journal*, pages 233–243, 1991.

[56] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 161–176, Berkeley, CA, USA, 2005. USENIX Association.

[57] Rajeev Kumar and Peter Rockett. Improved sampling of the pareto-front in multiobjective genetic optimizations by steady-state evolution: a pareto converging genetic algorithm. *Evol. Comput.*, 10(3):283–314, 2002.

[58] William B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Kluwer, 1998.

[59] William B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[60] Junmyung Lee, Sungzoon Cho, and Jinwoo Baek. Trend detection using autoassociative neural networks: Intraday kospi 200 futures. *Computational Intelligence for Financial Engineering, 2003. Proceedings. 2003 IEEE International Conference on*, pages 417–420, March 2003.

[61] David D. Lewis. Reuters-21578 text categorization test collection. `http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html`, September 1997.

[62] Peng Li, Hyundo Park, Debin Gao, and Jianming Fu. Bridging the gap between data-flow and control-flow analysis for anomaly detection. *Annual Computer Security Applications Conference, 2008. ACSAC 2008.*, pages 392–401, Dec. 2008.

[63] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. Analysis and results of the 1999 darpa off-line intrusion detection evaluation. In *RAID '00: Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pages 162–182, London, UK, 2000. Springer-Verlag.

[64] Larry Manevitz and Malik Yousef. One-class document classification via neural networks. *Neurocomput.*, 70(7-9):1466–1481, 2007.

[65] Markos Markou and Sameer Singh. Novelty detection: a review—part 1: statistical approaches. *Signal Process.*, 83(12):2481–2497, 2003.

[66] Raffael Marty. Thor: A tool to test intrusion detection systems by variations of attacks. Master's thesis, Swiss Federal Institute of Technology, 2002.

[67] Andrew R. McIntyre. *Novelty Detection + Coevolution = Automatic Problem Decomposition: A Framework For Scalable Genetic Programming Classifiers*. PhD thesis, Dalhousie University, November 2007.

[68] Brad L. Miller and Michael J. Shaw. Genetic algorithms with dynamic niche sharing for multimodal function optimization. Technical report, University Of Illinois at Urbana-Champaign, Dept. General Engineering, IlliGAL Report 95010, 1995.

[69] Melanie Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 1996.

[70] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[71] Robert T. Morris. A weakness in the 4.2bsd unix tcp/ip software.

[72] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, 2006.

[73] Darren Mutz, Giovanni Vigna, and Richard Kemmerer. An experience developing an ids stimulator for the black-box testing of network intrusion detection systems. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, page 374, Washington, DC, USA, 2003. IEEE Computer Society.

[74] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[75] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[76] Michael O'Neill and Conor Ryan. *Grammatical Evolution Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer, 2003.

[77] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *ASIACCS '08: Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 156–167, New York, NY, USA, 2008. ACM.

[78] Ron Patton. *Software Testing (2nd Edition)*. Sams, Indianapolis, IN, USA, 2005.

[79] Udo Payer, Peter Teufl, and Mario Lamberger. Hybrid engine for polymorphic shellcode detection. In *Intrusion and Malware Detection and Vulnerability Assessment*, LNCS-3548, pages 19–31. Springer Berlin / Heidelberg, 2005.

[80] Cyrus Peikari and Anton Chuvakin. *Security Warrior*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[81] Doron A. Peled, David Gries, and Fred B. Schneider, editors. *Software reliability methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[82] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks Inc., 1998.

[83] Quan Qian and Mingjun Xin. Research on hidden markov model for system call anomaly detection. In *Intelligence and Security Informatics, Pacific Asia Workshop, PAISI 2007, Chengdu, China, April 11-12, 2007, Proceedings*, volume 4430 of *Lecture Notes in Computer Science*, pages 152–159. Springer, 2007.

[84] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Readings in speech recognition*, pages 267–296, 1990.

[85] Michael Rash, Angela D. Orebaugh, Graham Clark, Becky Pinkard, and Jake Babbin. *Intrusion Prevention and Active Response: Deploying Network and Host IPS*. Syngress Publishing, 2005.

[86] Isidore Rigoutsos and Aris Floratos. Combinatorial pattern discovery in biological sequences: The teiresias algorithm. *Bioinformatics*, 14(1):55–67, 1998.

[87] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of Thirteenth Systems Administration Conference – LISA*, pages 229–238, 1999.

[88] Shai Rubin, Somesh Jha, and Barton P. Miller. Automatic generation and analysis of nids attacks. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, pages 28–38, Washington, DC, USA, 2004. IEEE Computer Society.

[89] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 144, Washington, DC, USA, 2001. IEEE Computer Society.

[90] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM.

[91] Anil Buntwal Somayaji. *Operating system stability and security through process homeostasis*. PhD thesis, The University of New Mexico, 2002. Chairperson: Stephanie Forrest.

[92] Sherri Sparks, Shawn Embleton, Ryan Cunningham, and Cliff Zou. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA*, pages 477–486. IEEE Computer Society, 2007.

[93] Richard M. Stallman and the GCC Developer Community. Gnu compiler collection internals. `http://gcc.gnu.org/onlinedocs/`, 2008.

[94] Sufatrio and Roland H.C. Yap. Improving host-based ids with argument abstraction to prevent mimicry attacks. In *Recent Advances in Intrusion Detection (RAID 2005)*, volume LNCS 3858, pages 146 – 164, 2006.

[95] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[96] Kymie M. C. Tan, Kevin S. Killourhy, and Roy A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection - RAID, Lecture Notes in Computer Science*, LNCS 2516, pages 54–73, 2002.

[97] Kymie M. C. Tan and Roy A. Maxion. "why 6?" defining the operational limits of stide, an anomaly-based intrusion detector. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 188, Washington, DC, USA, 2002. IEEE Computer Society.

[98] Kymie M. C. Tan and Roy A. Maxion. Determining the operational limits of an anomaly-based intrusion detector. *Selected Areas in Communications, IEEE Journal on*, 21(1):96–110, Jan 2003.

[99] Kymie M. C. Tan, John McHugh, and Kevin S. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *IH '02: Revised Papers from the 5th International Workshop on Information Hiding*, pages 1–17, London, UK, 2003. Springer-Verlag.

[100] Adam D. Todd, Richard A. Raines, Rusty O. Baldwin, Barry E. Mullins, and Steven K. Rogers. Alert verification evasion through server response forging. In *Recent Advances in Intrusion Detectio*, volume LCNS 4637, pages 256–275, 2007.

[101] Vangelis. Stack-based overflow exploit: Introduction to classical and advanced overflow technique. Wowhacker via Neworder, December 2004.

[102] Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 21–30, New York, NY, USA, 2004. ACM.

[103] David Wagner and Drew Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[104] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium*, pages 3–17, 2000.

[105] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM.

[106] GDB: The GNU Project Debugger Website. `http://www.gnu.org/software/gdb/`, Last accessed October 2008.

[107] Snort Website. `http://www.snort.org`, Last accessed June 2008.

[108] Stack Shield Website. `http://www.angelfire.com/sk/stackshield/`, Last accessed August 2008.

[109] Stide Website. Source code of stide and system call data sets. `http://www.cs.unm.edu/~immsec/systemcalls.htm`, Last accessed June 2008.

[110] Andreas Wespi, Marc Dacier, and Hervé Debar. Intrusion detection using variable-length audit trail patterns. In *RAID '00: Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pages 110–129, London, UK, 2000. Springer-Verlag.

[111] Wolfgang J.H. Wickler. Mimicry. In *In Encyclopaedia Britannica*. Encyclopaedia Britannica Online: `http://www.britannica.com/EBchecked/topic/383252/mimicry`, 2009.

[112] Ian Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, October 1999.

[113] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 243–257, Washington, DC, USA, 2006. IEEE Computer Society.

[114] Dit-Yan Yeung and Yuxin Ding. Host-based intrusion detection using dynamic and static behavioral models. *Pattern Recognition*, Volume 36:pages 229–243, 2002.

[115] Eckart Zitzler and Lothar Thiele. Multiobjective evolutionary algorithms: A comprehensive case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

Appendix A

Detector Training Set Analysis

A.1 Stide Training Set Analysis

The training set analysis for Stide is discussed in Section 8.4.

A.2 pH Training Set Analysis

Table A.1: Anomaly rates reported by pH with different training combinations for traceroute

	case1	**case2**	**case3**	**case4**	**case5**	**attack**
case1	0.00%	3.17%	8.97%	14.18%	18.75%	66.57%
case2	45.47%	0.00%	8.97%	11.19%	18.75%	66.57%
case3	81.04%	46.03%	0.00%	52.99%	18.75%	66.87%
case4	70.88%	40.48%	42.76%	0.00%	18.75%	73.43%
case5	96.70%	90.48%	83.45%	88.81%	0.00%	82.09%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	66.27%

Table A.2: Anomaly rates reported by pH with different training combinations for samba

	case1	**case2**	**case3**	**case4**	**case5**	**case6**	**attack**
case1	0.00%	4.76%	18.54%	100.00%	99.89%	36.67%	24.68%
case2	37.11%	0.00%	71.35%	73.58%	99.94%	52.80%	37.14%
case3	55.57%	56.99%	0.00%	100.00%	100.00%	73.80%	63.86%
case4	74.91%	59.38%	97.75%	0.00%	99.73%	81.63%	64.71%
case5	71.17%	53.87%	96.63%	23.05%	0.00%	77.35%	62.72%
case6	13.24%	4.46%	16.85%	71.14%	85.70%	0.00%	18.55%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	16.02%

Table A.3: Anomaly rates reported by pH with different training combinations for restore

	case1	**case2**	**case3**	**case4**	**case5**	**attack**
case1	0.00%	1.37%	0.00%	0.02%	8.89%	87.51%
case2	1.65%	0.00%	0.01%	0.01%	8.89%	88.53%
case3	1.02%	1.86%	0.00%	0.03%	8.89%	87.51%
case4	1.82%	0.59%	0.01%	0.00%	8.89%	88.53%
case5	98.18%	95.98%	99.98%	99.94%	0.00%	89.43%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	87.49%

Table A.4: Anomaly rates reported by pH with different training combinations for ftpd

	case1	**case2**	**case3**	**case4**	**case5**	**case6**	**case7**	**case8**	**case9**	**case10**	**attack**
case1	0.00%	0.02%	0.00%	2.41%	0.16%	0.02%	9.47%	0.25%	1.86%	0.25%	30.18%
case2	0.27%	0.00%	0.00%	2.18%	0.15%	0.01%	9.25%	0.00%	1.63%	0.00%	30.02%
case3	0.27%	0.00%	0.00%	2.18%	0.15%	0.01%	9.25%	0.00%	1.63%	0.00%	30.02%
case4	1.83%	0.11%	0.01%	0.00%	0.00%	0.00%	8.98%	0.00%	1.41%	0.00%	31.44%
case5	1.83%	0.11%	0.01%	0.00%	0.00%	0.00%	8.98%	0.00%	1.41%	0.00%	31.44%
case6	1.83%	0.11%	0.01%	0.00%	0.00%	0.00%	8.98%	0.00%	1.41%	0.00%	31.44%
case7	22.36%	94.70%	99.48%	22.95%	94.74%	99.48%	0.00%	15.18%	20.82%	15.18%	42.49%
case8	6.87%	70.34%	74.54%	7.80%	70.42%	74.55%	9.69%	0.00%	6.03%	0.00%	33.33%
case9	2.37%	46.74%	49.68%	2.54%	46.76%	49.68%	9.02%	0.00%	0.00%	0.00%	31.61%
case10	6.87%	70.34%	74.54%	7.80%	70.42%	74.55%	9.69%	0.00%	6.03%	0.00%	33.33%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	25.54%

A.3 pH with a Schema Mask Training Set Analysis

Table A.5: Anomaly rates reported by pH with a schema mask with different training combinations for traceroute

	case1	**case2**	**case3**	**case4**	**case5**	**attack**
case1	0.00%	6.22%	10.45%	26.83%	60.00%	82.10%
case2	61.65%	0.00%	10.45%	23.58%	60.00%	82.10%
case3	83.54%	51.04%	0.00%	57.72%	60.00%	82.41%
case4	83.54%	61.00%	58.96%	0.00%	60.00%	93.21%
case5	99.72%	99.17%	98.51%	98.37%	0.00%	98.15%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	81.79%

Table A.6: Anomaly rates reported by pH with a schema mask with different training combinations for samba

	case1	**case2**	**case3**	**case4**	**case5**	**case6**	**attack**
case1	0.00%	7.26%	23.35%	79.57%	99.88%	43.17%	27.50%
case2	43.80%	0.00%	90.42%	82.98%	99.96%	58.42%	40.25%
case3	61.04%	60.36%	0.00%	100.00%	100.00%	77.98%	66.08%
case4	99.65%	99.55%	100.00%	0.00%	99.88%	99.80%	99.18%
case5	76.61%	60.06%	100.00%	37.87%	0.00%	82.43%	66.61%
case6	17.85%	8.32%	25.75%	78.72%	85.75%	0.00%	19.14%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	16.67%

Table A.7: Anomaly rates reported by pH with a schema mask with different training combinations for restore

	case1	**case2**	**case3**	**case4**	**case5**	**attack**
case1	0.00%	2.08%	0.00%	0.04%	11.76%	96.91%
case2	2.24%	0.00%	0.02%	0.02%	11.76%	97.11%
case3	1.30%	2.68%	0.00%	0.04%	11.76%	97.11%
case4	2.37%	0.99%	0.02%	0.00%	11.76%	97.09%
case5	98.66%	97.02%	99.98%	99.96%	0.00%	98.06%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	96.77%

Table A.8: Anomaly rates reported by pH with a schema mask with different training combinations for ftpd

	case1	**case2**	**case3**	**case4**	**case5**	**case6**	**case7**	**case8**	**case9**	**case10**	**attack**
case1	0.00%	0.01%	0.00%	3.45%	0.23%	0.02%	12.67%	0.25%	2.60%	0.25%	30.46%
case2	0.13%	0.00%	0.00%	3.40%	0.22%	0.02%	12.63%	0.20%	2.51%	0.20%	30.43%
case3	0.13%	0.00%	0.00%	3.40%	0.22%	0.02%	12.63%	0.20%	2.51%	0.20%	30.43%
case4	2.78%	23.49%	24.85%	0.00%	23.31%	24.83%	12.63%	0.15%	2.37%	0.15%	29.39%
case5	2.65%	0.17%	0.02%	0.04%	0.00%	0.00%	12.63%	0.15%	2.32%	0.15%	29.33%
case6	2.65%	0.17%	0.02%	0.04%	0.00%	0.00%	12.63%	0.15%	2.32%	0.15%	29.33%
case7	23.09%	94.78%	99.49%	23.87%	94.83%	99.49%	0.00%	15.17%	22.15%	15.17%	45.14%
case8	8.83%	70.51%	74.56%	9.99%	70.59%	74.57%	13.04%	0.00%	7.89%	0.00%	36.48%
case9	3.18%	23.52%	24.85%	4.08%	23.58%	24.86%	12.67%	0.10%	0.00%	0.00%	26.86%
case10	8.83%	70.51%	74.56%	9.99%	70.59%	74.57%	13.04%	0.00%	7.89%	0.00%	36.48%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	20.27%

A.4 Markov Model Training Set Analysis

Table A.9: Anomaly rates reported by the Markov Model detector with different training combinations for traceroute

	case1	**case2**	**case3**	**case4**	**case5**	**attack**
case1	0.00%	0.38%	1.96%	8.33%	1.41%	39.07%
case2	11.28%	0.00%	1.96%	8.33%	0.70%	39.65%
case3	71.33%	26.15%	0.00%	8.33%	30.28%	43.15%
case4	82.61%	62.31%	43.14%	0.00%	66.90%	58.89%
case5	15.90%	10.00%	15.69%	8.33%	0.00%	40.52%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	38.78%

Table A.10: Anomaly rates reported by the Markov Model detector with different training combinations for samba

	case1	**case2**	**case3**	**case4**	**case5**	**case6**	**attack**
case1	0.00%	1.47%	4.84%	19.69%	14.42%	15.26%	11.30%
case2	13.06%	0.00%	18.28%	25.98%	28.60%	24.36%	16.37%
case3	29.15%	34.41%	0.00%	77.17%	71.37%	43.35%	34.68%
case4	55.80%	41.91%	73.66%	0.00%	42.83%	61.49%	42.35%
case5	46.80%	29.56%	68.82%	4.72%	0.00%	48.33%	35.77%
case6	5.36%	0.29%	5.38%	25.20%	14.37%	0.00%	10.21%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	9.03%

Table A.11: Anomaly rates reported by the Markov Model detector with different training combinations for restore

	case1	**case2**	**case3**	**case4**	**case5**	**attack**
case1	0.00%	0.29%	0.00%	0.00%	3.77%	45.05%
case2	0.09%	0.00%	0.00%	0.00%	3.77%	44.26%
case3	0.00%	0.29%	0.00%	0.00%	3.77%	45.05%
case4	0.09%	0.00%	0.00%	0.00%	3.77%	44.26%
case5	22.25%	20.93%	22.25%	22.24%	0.00%	69.55%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	44.26%

Table A.12: Anomaly rates reported by the Markov Model detector with different training combinations for ftpd

	case1	**case2**	**case3**	**case4**	**case5**	**case6**	**case7**	**case8**	**case9**	**case10**	**attack**
case1	0.00%	0.00%	0.00%	0.18%	0.01%	0.00%	0.58%	0.05%	0.41%	0.05%	7.74%
case2	0.04%	0.00%	0.00%	0.13%	0.01%	0.00%	0.54%	0.00%	0.36%	0.00%	7.71%
case3	0.04%	0.00%	0.00%	0.13%	0.01%	0.00%	0.54%	0.00%	0.36%	0.00%	7.71%
case4	0.22%	0.01%	0.00%	0.00%	0.00%	0.00%	0.76%	0.00%	0.27%	0.00%	7.48%
case5	0.22%	0.01%	0.00%	0.00%	0.00%	0.00%	0.76%	0.00%	0.27%	0.00%	7.48%
case6	0.22%	0.01%	0.00%	0.00%	0.00%	0.00%	0.76%	0.00%	0.27%	0.00%	7.48%
case7	13.56%	47.51%	49.75%	14.17%	47.55%	49.76%	0.00%	8.92%	12.16%	8.92%	14.22%
case8	2.62%	46.76%	49.68%	2.75%	46.77%	49.68%	1.03%	0.00%	2.03%	0.00%	7.91%
case9	0.22%	0.01%	0.00%	0.09%	0.01%	0.00%	0.76%	0.00%	0.00%	0.00%	7.24%
case10	2.62%	46.76%	49.68%	2.75%	46.77%	49.68%	1.03%	0.00%	2.03%	0.00%	7.91%
all normal	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	7.15%

A.5 Neural Network Training Set Analysis

Table A.13: Anomaly rates reported by the Neural Network detector with different training combinations for traceroute

	case1	**case2**	**case3**	**case4**	**case5**	**attack**
case1	0.01%	51.63%	65.26%	66.81%	51.68%	80.51%
case2	69.47%	0.01%	25.24%	38.22%	9.57%	23.99%
case3	70.86%	20.42%	0.02%	26.23%	21.90%	46.17%
case4	46.65%	7.61%	21.88%	0.01%	27.63%	28.61%
case5	100.00%	46.39%	35.71%	47.41%	0.03%	44.58%
all normal	1.87%	2.16%	2.29%	5.34%	2.89%	31.19%

Table A.14: Anomaly rates reported by the Neural Network detector with different training combinations for samba

	case1	**case2**	**case3**	**case4**	**case5**	**case6**	**attack**
case1	0.01%	25.46%	29.58%	57.53%	76.44%	13.56%	20.42%
case2	31.39%	0.01%	69.46%	71.34%	98.26%	46.98%	8.88%
case3	22.33%	40.64%	0.01%	28.73%	38.10%	27.37%	33.72%
case4	79.69%	94.59%	73.45%	0.01%	100.00%	80.84%	93.90%
case5	84.20%	93.88%	72.73%	75.26%	0.01%	69.84%	90.21%
case6	28.07%	45.88%	38.64%	54.93%	87.69%	0.02%	39.09%
all normal	4.00%	4.01%	4.25%	3.51%	2.38%	3.14%	5.73%

Table A.15: Anomaly rates reported by the Neural Network detector with different training combinations for restore

	case1	**case2**	**case3**	**case4**	**case5**	**attack**
case1	0.01%	6.66%	6.65%	6.53%	88.19%	100.00%
case2	2.84%	0.00%	5.62%	5.57%	47.99%	60.35%
case3	3.89%	7.63%	0.00%	0.07%	57.59%	62.53%
case4	4.84%	9.84%	0.09%	0.01%	76.06%	78.28%
case5	27.91%	26.45%	29.54%	29.52%	0.00%	15.50%
all normal	0.24%	0.40%	0.22%	0.22%	0.30%	14.00%

Table A.16: Anomaly rates reported by the Neural Network detector with different training combinations for ftpd

	case1	**case2**	**case3**	**case4**	**case5**	**case6**	**case7**	**case8**	**case9**	**case10**	**attack**
case1	0.01%	68.95%	73.51%	0.38%	68.94%	73.51%	5.95%	9.04%	1.06%	9.04%	24.00%
case2	81.75%	0.02%	5.80%	82.06%	0.04%	5.79%	83.55%	86.49%	82.04%	86.49%	66.51%
case3	74.30%	4.85%	0.01%	74.48%	4.87%	0.01%	72.96%	77.24%	74.52%	77.24%	62.52%
case4	0.30%	90.12%	95.80%	0.01%	90.10%	95.80%	5.12%	9.50%	1.10%	9.50%	25.68%
case5	39.22%	0.02%	2.68%	39.39%	0.01%	2.68%	40.75%	40.95%	39.28%	40.95%	34.55%
case6	76.92%	4.96%	0.02%	77.14%	4.98%	0.01%	79.32%	80.09%	76.95%	80.09%	65.92%
case7	4.93%	78.97%	84.02%	4.94%	78.95%	84.02%	0.01%	7.71%	4.72%	7.71%	21.90%
case8	5.85%	58.25%	61.60%	5.83%	58.24%	61.60%	9.50%	0.01%	5.18%	0.01%	20.55%
case9	0.91%	87.66%	93.11%	0.84%	87.64%	93.11%	5.16%	5.26%	0.01%	5.26%	17.39%
case10	9.88%	94.53%	100.00%	9.77%	94.51%	100.00%	12.63%	0.03%	8.62%	0.03%	36.02%
all normal	0.54%	0.54%	0.46%	0.61%	0.54%	0.46%	1.16%	0.54%	0.55%	0.54%	6.91%

Appendix B

Best Mimicry Attacks

B.1 Best Mimicry Attacks against Stide

open mmap fstat sendto gettimeofday select (2 x write) gettimeofday sendto gettimeofday select (2 x write) gettimeofday sendto gettimeofday select (2 x write) gettimeofday sendto gettimeofday select (2 x write) gettimeofday sendto gettimeofday select (2 x write) gettimeofday close

Figure B.1: The best mimicry attack against Stide for traceroute

stat open (4 x _llseek) read (2 x _llseek) read (16 x _llseek) read _llseek read (2 x _llseek) read (10 x _llseek) read _llseek read (6 x _llseek) read (17 x _llseek) read _llseek read (9 x _llseek) read (6 x _llseek) read _llseek read (7 x _llseek) read (9 x _llseek) read (4 x _llseek) read (4 x _llseek) read (4 x _llseek) read (22 x _llseek) read (2 x _llseek) read (4 x _llseek) read (8 x _llseek) read (6 x _llseek) read (2 x _llseek) read (7 x _llseek) read _llseek read (12 x _llseek) read _llseek read (4 x _llseek) read (12 x _llseek) read (26 x _llseek) read (2 x _llseek) read (4 x _llseek) read (6 x _llseek) read (2 x _llseek) read (7 x _llseek) read (6 x _llseek) read (12 x _llseek) read (8 x _llseek) read (10 x _llseek) read (5 x _llseek) read (6 x _llseek) read (6 x _llseek) read (2 x _llseek) read (4 x _llseek) read (6 x _llseek) read (5 x _llseek) read _llseek read (2 x _llseek) read (17 x _llseek) read (6 x _llseek) read (6 x _llseek) read (9 x _llseek) read (19 x _llseek) read (5 x _llseek) read (2 x _llseek) read (4 x _llseek) read (4 x _llseek) read (2 x _llseek) read (7 x _llseek) read (4 x _llseek) read (4 x _llseek) read (11 x _llseek) read (18 x _llseek) read (6 x _llseek) read (4 x _llseek) read (6 x _llseek) read (13 x _llseek) read (2 x _llseek) read (7 x _llseek) read (6 x _llseek) read (8 x _llseek) read (6 x _llseek) read _llseek read (5 x _llseek) read _llseek read (2 x _llseek) read (13 x _llseek) read (4 x _llseek) read (5 x _llseek) read (4 x _llseek) read (13 x _llseek) read (2 x _llseek) read (4 x _llseek) read (6 x _llseek) read (8 x _llseek) read (2 x _llseek) read (7 x _llseek) read _llseek read (4 x _llseek) read (5 x _llseek) read _llseek read _llseek read (2 x _llseek) read (12 x _llseek) read _llseek read (20 x _llseek) read (4 x _llseek) read (8 x _llseek) read (15 x _llseek) read (2 x _llseek) read (5 x _llseek) read (8 x _llseek) read (5 x _llseek) read (6 x _llseek) read (4 x _llseek) read (5 x _llseek) read _llseek read (4 x _llseek) read _llseek read (2 x _llseek) read (7 x _llseek) read (2 x _llseek) read (7 x _llseek) read (6 x _llseek) read (15 x _llseek) read _llseek read (11 x _llseek) read (6 x _llseek) read (6 x _llseek) read (14 x _llseek) read (6 x _llseek) read (5 x _llseek) read (4 x _llseek) read (2 x _llseek) read (5 x _llseek) read (4 x _llseek) read (2 x _llseek) read (6 x _llseek) read _llseek read (4 x _llseek) read (8 x _llseek) read (10 x _llseek) read (2 x _llseek) read (5 x _llseek) read (9 x _llseek) read (8 x _llseek) read (4 x _llseek) read (2 x _llseek) read (8 x _llseek) read (11 x _llseek) write close

Figure B.2: The best mimicry attack against Stide for samba

(18 x write) read (7 x write) read (27 x write) read (5 x write) read (13 x write) read (8 x write) read (5 x write) read (23 x write) read (11 x write) read (14 x write) read (12 x write) read (19 x write) read (5 x write) read (30 x write) read (12 x write) read (19 x write) read (5 x write) read (18 x write) read (8 x write) read (20 x write) read (35 x write) read (12 x write) read (19 x write) read (5 x write) read (13 x write) read (23 x write) read (7 x write) read (7 x write) read (27 x write) read (7 x write) read (5 x write) read (21 x write) read (5 x write) read (31 x write) read (7 x write) read (5 x write) read (14 x write) read (9 x write) read (5 x write) read (22 x write) read (8 x write) read (13 x write) read (14 x write) read (8 x write) read (11 x write) read (32 x write) read (8 x write) read (5 x write) read (18 x write) read (7 x write) read (23 x write) read (11 x write) read (23 x write) read (39 x write) read (21 x write) read (5 x write) read (21 x write) read (5 x write) read (31 x write) read (7 x write) read (5 x write) read (17 x write) read (19 x write) read (19 x write) read (5 x write) open write close

Figure B.3: The best mimicry attack against Stide for restore

connect getcwd fchdir close open fstat write close time open alarm

Figure B.4: The best mimicry attack against Stide for ftpd

B.2 Best Mimicry Attacks against pH

```
brk (3 x open) munmap mmap (2 x open) (2 x fstat) (3 x mmap) close munmap (3 x open) fstat
(3 x mmap) (2 x close) (2 x open) munmap (2 x close) open fstat (4 x mmap) close munmap
open fstat (4 x mmap) close munmap (3 x open) fstat (4 x mmap) close (2 x open) fstat (3 x
mmap) close munmap (3 x open) fstat (4 x mmap) close (3 x open) close open fstat (4 x mmap)
close munmap (3 x open) fstat (3 x mmap) close munmap open fstat (3 x mmap) close munmap (3
x open) fstat (4 x mmap) close (3 x open) mprotect mmap close open close write close
```

Figure B.5: The best mimicry attack against pH for traceroute

mmap2 (2 x fcntl64) open munmap fcntl64 write fcntl64 munmap (3 x fcntl64) munmap fcntl64 stat fcntl64 stat fcntl64 open (4 x fcntl64) stat (2 x fcntl64) stat (4 x fcntl64) munmap fcntl64 time fcntl64 stat (7 x fcntl64) munmap (2 x fcntl64) stat (2 x fcntl64) munmap (4 x fcntl64) munmap (2 x fcntl64) munmap (2 x fcntl64) munmap (4 x fcntl64) stat munmap (2 x fcntl64) stat fcntl64 stat (4 x fcntl64) stat munmap (6 x fcntl64) (2 x munmap) fcntl64 munmap stat (3 x fcntl64) stat (2 x fcntl64) stat mmap2 (11 x fcntl64) stat munmap (2 x fcntl64) munmap (4 x fcntl64) munmap fcntl64 munmap (2 x fcntl64) (2 x munmap) (8 x fcntl64) stat fcntl64 stat fcntl64 (2 x munmap) fcntl64 stat mmap2 (5 x fcntl64) munmap fcntl64 munmap (2 x fcntl64) munmap (4 x fcntl64) munmap (5 x fcntl64) read stat (3 x fcntl64) stat mmap2 (4 x fcntl64) stat (2 x fcntl64) munmap fcntl64 munmap (7 x fcntl64) munmap (2 x fcntl64) munmap (2 x fcntl64) munmap fcntl64 stat munmap (2 x fcntl64) stat (5 x fcntl64) time (8 x fcntl64) (2 x munmap) fcntl64 munmap stat (3 x fcntl64) stat fcntl64 (2 x stat) mmap2 (6 x fcntl64) munmap fcntl64 stat (2 x fcntl64) stat munmap (2 x fcntl64) munmap (4 x fcntl64) munmap fcntl64 munmap (3 x fcntl64) munmap (3 x fcntl64) munmap (3 x fcntl64) (2 x stat) munmap (2 x fcntl64) (2 x munmap) (2 x fcntl64) munmap fcntl64 stat (3 x fcntl64) munmap stat (3 x fcntl64) munmap (4 x fcntl64) stat (8 x fcntl64) munmap (2 x fcntl64) munmap (10 x fcntl64) stat (2 x fcntl64) munmap (2 x fcntl64) munmap fcntl64 stat (3 x fcntl64) stat (3 x fcntl64) munmap (2 x fcntl64) (2 x munmap) (5 x fcntl64) munmap (4 x fcntl64) munmap (6 x fcntl64) munmap fcntl64 stat munmap stat (8 x fcntl64) stat fcntl64 (3 x stat) munmap (4 x fcntl64) munmap (5 x fcntl64) munmap fcntl64 stat munmap fcntl64 stat munmap (2 x fcntl64) stat fcntl64 (2 x stat) (4 x fcntl64) munmap (6 x fcntl64) (2 x munmap) fcntl64 munmap stat (3 x fcntl64) (2 x stat) fcntl64 stat mmap2 (12 x fcntl64) munmap (2 x fcntl64) munmap (4 x fcntl64) munmap fcntl64 munmap (2 x fcntl64) (2 x munmap) (8 x fcntl64) stat fcntl64 stat fcntl64 (2 x munmap) (4 x fcntl64) stat (2 x fcntl64) (2 x munmap) stat (8 x fcntl64) stat mmap2 (7 x fcntl64) munmap (15 x fcntl64) stat (2 x munmap) stat fcntl64 stat munmap (6 x fcntl64) stat (5 x fcntl64) munmap (2 x fcntl64)

Figure B.6: The best mimicry attack against pH for samba (first part)

munmap fcntl64 stat munmap (2 x fcntl64) stat fcntl64 stat (5 x fcntl64) munmap fcntl64 munmap (3 x fcntl64) (2 x munmap) (2 x fcntl64) munmap stat fcntl64 munmap fcntl64 stat (2 x fcntl64) stat mmap2 (6 x fcntl64) munmap (2 x fcntl64) munmap (7 x fcntl64) stat (2 x fcntl64) stat fcntl64 stat fcntl64 stat fcntl64 munmap fcntl64 stat fcntl64 stat fcntl64 stat fcntl64 stat fcntl64 munmap (4 x fcntl64) (2 x munmap) (10 x fcntl64) munmap (3 x fcntl64) munmap fcntl64 stat munmap fcntl64 stat fcntl64 stat (6 x fcntl64) stat fcntl64 stat (2 x fcntl64) munmap fcntl64 munmap (4 x fcntl64) munmap (2 x fcntl64) munmap (4 x fcntl64) stat munmap fcntl64 stat munmap (3 x fcntl64) (2 x stat) (5 x fcntl64) munmap (6 x fcntl64) (2 x munmap) fcntl64 munmap stat (3 x fcntl64) (2 x stat) fcntl64 stat mmap2 (12 x fcntl64) (2 x munmap) stat fcntl64 munmap (3 x fcntl64) munmap fcntl64 munmap (2 x fcntl64) (2 x munmap) (8 x fcntl64) stat fcntl64 stat fcntl64 (3 x munmap) fcntl64 munmap fcntl64 stat (2 x fcntl64) (2 x munmap) stat (3 x fcntl64) munmap (4 x fcntl64) stat mmap2 (7 x fcntl64) munmap (2 x fcntl64) munmap (12 x fcntl64) stat (2 x munmap) stat fcntl64 stat munmap fcntl64 munmap (2 x fcntl64) stat fcntl64 munmap (3 x fcntl64) stat (2 x munmap) (2 x fcntl64) munmap (2 x fcntl64) munmap (2 x fcntl64) stat (7 x fcntl64) munmap (2 x fcntl64) munmap (2 x fcntl64) munmap (3 x fcntl64) munmap stat (6 x fcntl64) stat (11 x fcntl64) munmap stat munmap (2 x fcntl64) munmap stat (3 x fcntl64) munmap fcntl64 stat (3 x fcntl64) munmap (8 x fcntl64) stat fcntl64 stat fcntl64 (2 x munmap) fcntl64 stat munmap (4 x fcntl64) (2 x munmap) stat (2 x fcntl64) stat fcntl64 munmap fcntl64 (2 x munmap) stat munmap fcntl64 stat (2 x fcntl64) (2 x munmap) stat (2 x fcntl64) stat fcntl64 munmap (2 x fcntl64) munmap (15 x fcntl64) munmap (2 x fcntl64) (2 x munmap) (6 x fcntl64) munmap fcntl64 read stat (2 x fcntl64) (2 x munmap) (8 x fcntl64) stat open (3 x fcntl64) munmap stat (3 x fcntl64) munmap (9 x fcntl64) (2 x stat) mmap2 (5 x fcntl64) (2 x time) fcntl64 time (6 x fcntl64) munmap (10 x fcntl64) stat (4 x fcntl64) munmap (2 x fcntl64) munmap (3 x fcntl64) stat fcntl64 stat fcntl64 close

Figure B.7: The best mimicry attack against pH for samba (second part)

(2 x fcntl) open write read lseek (13 x write) lseek (3 x write) lseek (3 x write) lseek (4 x write) lseek (2 x write) (2 x lseek) (8 x write) lseek (5 x write) read write read (9 x write) lseek (16 x write) lseek (18 x write) lseek (6 x write) read (19 x write) lseek (4 x write) lseek write read (7 x write) lseek (10 x write) (2 x read) (2 x write) lseek (19 x write) lseek (7 x write) lseek (21 x write) read lseek (19 x write) lseek (5 x write) lseek (18 x write) read (20 x write) lseek (3 x write) (2 x lseek) (12 x write) lseek (16 x write) read (11 x write) (2 x lseek) (3 x write) lseek (26 x write) lseek (2 x write) read (7 x write) (2 x lseek) (15 x write) lseek (14 x write) lseek (8 x write) lseek (4 x write) lseek (11 x write) lseek (4 x write) read (4 x write) read (7 x write) lseek (7 x write) lseek (6 x write) read lseek (25 x write) lseek (8 x write) lseek (7 x write) read (18 x write) lseek (13 x write) lseek (14 x write) lseek (15 x write) lseek (2 x write) read lseek (13 x write) read (15 x write) lseek (7 x write) lseek (8 x write) lseek (18 x write) lseek (4 x write) lseek write read lseek (8 x write) read lseek (7 x write) read (2 x lseek) (12 x write) read (7 x write) lseek (3 x write) lseek (3 x write) lseek (8 x write) lseek (13 x write) read (4 x write) lseek (35 x write) lseek (7 x write) lseek (4 x write) lseek (2 x write) (2 x lseek) (8 x write) lseek (7 x write) read lseek (2 x write) lseek (9 x write) read (8 x write) lseek (7 x write) lseek (9 x write) lseek (11 x write) read (22 x write) lseek (21 x write) (2 x lseek) (7 x write) lseek (37 x write) lseek (6 x write) lseek (6 x write) lseek write lseek (15 x write) lseek (9 x write) lseek (3 x write) lseek (16 x write) read (3 x write) close

Figure B.8: The best mimicry attack against pH for restore

open close read write (23 x close) read (6 x close) read (21 x close) time (2 x open) read write (46 x close) read (11 x close) read (3 x close) time (2 x open) (2 x read) (11 x close) time (2 x open) read write (23 x close) read (11 x close) read (3 x close) time (2 x open) (2 x read) (4 x close) time (2 x open) read write (22 x close) read (14 x close) open (2 x read) (66 x close) read (3 x close) (2 x read) (88 x close) time (2 x open) read write (78 x close) time (2 x open) read write (58 x close) read (3 x close) time (2 x open) (2 x read) (4 x close) time (2 x open) read write (45 x close) time (2 x open) read write (21 x close) read (157 x close) read (21 x close) time (2 x open) read write (9 x close) read (36 x close) read (11 x close) read (3 x close) time (2 x open) (2 x read) (4 x close) time (2 x open) read write close read write (17 x close) read (34 x close) read (45 x close) time (2 x open) read write close

Figure B.9: The best mimicry attack against pH for ftpd

B.3 Best Mimicry Attacks against pH with a Schema Mask

gettimeofday (2 x open) write fstat gettimeofday (2 x mmap) open read open close (6 x open) mmap open close (2 x open) mmap open mmap (5 x open) mmap open (2 x mmap) (6 x open) mmap (7 x open) mmap (10 x open) mmap (13 x open) mmap (5 x open) mmap (4 x open) mmap open mmap (5 x open) mmap (25 x open) mmap (4 x open) mmap (4 x open) (4 x mmap) (5 x open) (2 x mmap) (4 x open) mmap (6 x open) mmap (6 x open) (2 x mmap) (6 x open) (2 x mmap) close mmap (2 x open) mmap (9 x open) mmap open (2 x mmap) (6 x open) mmap (18 x open) mmap (24 x open) mmap open mmap (22 x open) mmap open mmap (4 x open) mmap open mmap (4 x open) mmap (4 x open) mmap open mmap (4 x open) mmap open close mmap (2 x open) mmap (10 x open) mmap (6 x open) mmap (11 x open) mmap (2 x open) mmap (9 x open) mmap open mmap (4 x open) mmap open close mmap (2 x open) mmap (8 x open) mmap open mmap (18 x open) (4 x mmap) (13 x open) mmap open (2 x mmap) (5 x open) mmap open close (2 x open) (2 x mmap) (13 x open) mmap (6 x open) (2 x mmap) (6 x open) mmap open close mmap (22 x open) mmap (18 x open) (2 x mmap) open mmap (2 x open) mmap (5 x open) (3 x mmap) (6 x open) mmap open close mmap (12 x open) mmap open (2 x mmap) (6 x open) mmap (7 x open) mmap (10 x open) mmap (13 x open) mmap (10 x open) mmap open mmap (9 x open) mmap (10 x open) mmap open mmap (6 x open) mmap open mmap (4 x open) mmap (5 x open) (3 x mmap) (13 x open) mmap open (2 x mmap) (7 x open) mmap (13 x open) mmap (4 x open) mmap open mmap open close (2 x open) (2 x mmap) close mmap (2 x open) mmap open mmap (11 x open) mmap (9 x open) mmap open mmap (5 x open) mmap (4 x open) mmap (11 x open) mmap open mmap (4 x open) mmap open close mmap (2 x open) mmap (8 x open) mmap (7 x open) mmap open mmap (4 x open) mmap open close (3 x open) mmap (6 x open) mmap (5 x open) mmap (8 x open) mmap (28 x open) mmap (27 x open) mmap (8 x open) mmap open mmap (6 x open) mmap (8 x open) mmap (5 x open) mmap open mmap (4 x open) mmap open close (8 x open) (2 x mmap) open mmap (4 x open) mmap open close mmap (11 x open) mmap open mmap (18 x open) (2 x mmap) open mmap (4 x open) mmap (2 x open) mmap (5 x open) mmap (15 x open) mmap (18 x open) mmap munmap

Figure B.10: The best mimicry attack against pHsm for traceroute

open stat _llseek fcntl64 stat write fcntl64 munmap open close stat close fcntl64 read (3 x stat) read stat (4 x read) close mmap2 (2 x open) read (3 x stat) close read (3 x stat) fcntl64 (3 x stat) close read fcntl64 read stat read fcntl64 stat close (2 x read) mmap2 (4 x fcntl64) getegid32 stat _llseek read close (2 x read) open (2 x fcntl64) read stat (2 x close) stat read close read close fcntl64 stat mmap2 fcntl64 stat munmap fcntl64 (2 x stat) close stat close (2 x read) stat read stat (2 x read) stat open fcntl64 stat close stat fcntl64 read open read close mmap2 close open read (2 x stat) open (4 x fcntl64) (2 x stat) open stat close mmap2 stat open read stat (2 x fcntl64) close munmap read (3 x stat) (2 x fcntl64) read geteuid32 munmap _llseek read (2 x mmap2) read open read stat read stat close mmap2 (2 x read) open geteuid32 close fcntl64 (3 x read) fcntl64 stat (3 x fcntl64) geteuid32 stat _llseek open close (2 x read) stat (2 x read) (2 x stat) fcntl64 read fcntl64 open read (2 x stat) fcntl64 (2 x stat) read (4 x stat) read close (2 x read) mmap2 read fcntl64 open read stat munmap (2 x read) mmap2 read (3 x fcntl64) mmap2 stat close read close mmap2 read open read stat read stat read munmap (2 x stat) mmap2 stat open munmap read fcntl64 (2 x read) fcntl64 open read stat munmap (2 x read) open read (3 x fcntl64) mmap2 stat close (3 x stat) close open read stat read fcntl64 stat munmap close stat mmap2 stat close (2 x stat) mmap2 stat (3 x read) (3 x stat) munmap read (2 x stat) close open (2 x read) mmap2 read close stat close read stat read mmap2 stat read fcntl64 close munmap read stat open munmap (2 x fcntl64) stat _llseek read close (2 x mmap2) close (2 x read) stat read munmap (2 x read) fcntl64 stat close stat fcntl64 (2 x read) (2 x stat) open close open read (4 x stat) (3 x fcntl64) stat fcntl64 read open read (3 x fcntl64) geteuid32 read fcntl64 stat open read open close mmap2 close read open fcntl64 open (2 x stat) close read stat open stat (8 x fcntl64) getegid32 (2 x stat) fcntl64 mmap2 (2 x read) open (2 x read) fcntl64 close (2 x stat) close mmap2 read fcntl64 stat (2 x read) close fcntl64 stat read mmap2 read open read open read geteuid32 close open stat (3 x read) fcntl64 read stat close munmap read (2 x stat) (2 x close) (2 x fcntl64) mmap2 stat read (2 x stat) mmap2 read close read stat read stat read (4 x fcntl64) (3 x stat) close mmap2 (4 x read) stat mmap2 read open (3 x stat) fcntl64 read munmap (2 x read) stat read (2 x fcntl64) mmap2 geteuid32 stat mmap2 (2 x open) fcntl64 (3 x stat) read stat (2 x read) open read fcntl64 read stat close (2 x stat) read fcntl64 read munmap read (2 x stat) fcntl64 close (4 x stat) mmap2 fcntl64 close read open (2 x read)

Figure B.11: The best mimicry attack against pHsm for samba (first part)

```
mmap2 close open read (3 x stat) read (3 x fcntl64) (4 x stat) read mmap2 stat close stat
close mmap2 read open read (3 x stat) close (3 x read) mmap2 (4 x stat) mmap2 stat (2 x
read) (3 x stat) (2 x close) (4 x read) fcntl64 stat munmap (3 x stat) close stat (3 x
read) stat read (2 x stat) (2 x read) stat close (3 x stat) read fcntl64 mmap2 stat read
stat close mmap2 read open read open read stat close open read stat mmap2 stat open stat
close mmap2 stat (2 x read) stat fcntl64 stat close munmap read (2 x stat) close (2 x stat)
mmap2 (2 x open) read stat read fcntl64 read stat close munmap fcntl64 stat open mmap2
close (2 x fcntl64) geteuid32 (3 x stat) close mmap2 read open fcntl64 stat munmap stat
(4 x fcntl64) getegid32 (2 x stat) read mmap2 stat open read (5 x stat) (4 x read) close
mmap2 (2 x open) read fcntl64 read (4 x close) stat open stat close fcntl64 read stat (2
x read) (2 x fcntl64) read stat close (2 x stat) read fcntl64 munmap open read stat (2 x
fcntl64) close stat mmap2 stat open close fcntl64 close read open read close mmap2 close
open fcntl64 (2 x stat) (5 x fcntl64) (3 x stat) (2 x read) close stat close fcntl64 close
mmap2 read open read stat read stat close munmap (2 x stat) mmap2 stat close stat close
mmap2 stat (2 x read) (4 x stat) fcntl64 read open (2 x read) fcntl64 close read (3 x stat)
close stat (3 x read) (3 x stat) fcntl64 read munmap fcntl64 close (2 x stat) munmap read
fcntl64 mmap2 stat close stat close stat read open (3 x read) stat close munmap read stat
mmap2 open (3 x stat) mmap2 stat (2 x read) close fcntl64 stat close stat mmap2 close read
close (3 x stat) open (2 x read) fcntl64 read fcntl64 read stat close munmap fcntl64 stat
munmap mmap2 close mmap2 fcntl64 geteuid32 (2 x stat) fcntl64 stat mmap2 read open read (2
x stat) (5 x fcntl64) getegid32 (3 x stat) mmap2 stat open read stat close stat close stat
(3 x read) (2 x stat) close open read (2 x stat) fcntl64 (2 x stat) (2 x fcntl64) close
stat fcntl64 stat close fcntl64 open read close (2 x stat) mmap2 (2 x open) read fcntl64
read close stat mmap2 fcntl64 (3 x stat) open read close (2 x fcntl64) (2 x read) close
fcntl64 stat close mmap2 open read stat close stat (2 x read) fcntl64 stat close stat close
open read (3 x stat) munmap fcntl64 read mmap2 (4 x stat) mmap2 read open (3 x read) (2 x
stat) munmap read stat mmap2 stat fcntl64 stat close mmap2 stat (2 x read) (3 x stat) close
_llseek read getegid32 close munmap (2 x read) fcntl64 read (3 x fcntl64) (2 x stat) fcntl64
stat close mmap2 stat open read munmap fcntl64 mmap2 close (2 x read)
```

Figure B.12: The best mimicry attack against pHsm for samba (second part)

```
(4 x write) read write read (4 x write) read (2 x write) read (2 x write) (2 x read) write
read lseek (2 x write) lseek (3 x write) (2 x read) write read write (2 x read) write read
write read (6 x write) read (3 x write) read (2 x write) (2 x read) write (4 x read) (7 x
write) read write (2 x read) (8 x write) read (4 x write) lseek write read (3 x write) (2 x
read) write read lseek (4 x write) (2 x read) (3 x write) lseek write read (4 x write) read
write read (3 x write) read write read lseek (3 x write) lseek (2 x write) (3 x read) (2 x
write) (2 x read) (3 x write) read (3 x write) read write read mmap (7 x write) lseek write
read (2 x write) read (7 x write) lseek read (4 x write) read lseek (5 x write) read lseek
write (3 x read) (3 x write) lseek (2 x read) (2 x write) lseek write read (4 x write) (2 x
read) (2 x write) read (3 x write) (2 x read) (2 x write) (2 x read) (3 x write) (2 x read)
(2 x write) lseek write read write lseek write read (4 x write) (2 x read) (5 x write) read
(4 x write) read (4 x write) read (3 x write) (2 x read) write (2 x read) (5 x write) read
write read write read (7 x write) read lseek read (3 x write) read write (4 x read) (2 x
write) lseek read write read (3 x write) read lseek (7 x write) lseek write read write read
(3 x write) read (9 x write) read (8 x write) read (3 x write) read write (3 x read) write
read (3 x write) read lseek (2 x write) (2 x read) (2 x write) (3 x read) (2 x write) (3 x
read) mmap open write read (2 x write) read (9 x write) read (3 x write) lseek write read
write lseek write read (5 x write) (3 x read) (3 x write) read (6 x write) read (2 x write)
(3 x read) (5 x write) read (3 x write) read (10 x write) (2 x read) write lseek write
lseek (2 x read) (2 x write) lseek (2 x read) write lseek read (2 x write) lseek read (7
x write) lseek (2 x write) read (3 x write) read lseek (3 x read) (2 x write) lseek (7 x
read) (8 x write) read write (2 x read) write lseek (2 x read) write read (6 x write) read
write (2 x read)
```

Figure B.13: The best mimicry attack against pHsm for restore (first part)

```
lseek read (2 x write) read (6 x write) (2 x read) write read (2 x write) read (5 x write)
lseek write read (2 x write) (3 x read) (2 x lseek) write read write (4 x read) (2 x lseek)
read lseek write read (5 x write) lseek (2 x write) (2 x read) write read (8 x write) read
(3 x write) read (2 x write) (2 x read) (5 x write) read write read (5 x write) read write
read (2 x write) read write read (2 x write) lseek write read (5 x write) (2 x read) (2 x
write) read (2 x write) read write read lseek write (2 x read) lseek (2 x read) write read
(3 x write) (2 x read) lseek write read (2 x write) read write (2 x read) (4 x write) lseek
(2 x write) read (2 x write) read write read (13 x write) read (2 x write) read (4 x write)
read write read write read write read (10 x write) read (2 x write) lseek (4 x write) lseek
write (2 x read) (2 x write) read (2 x write) read (2 x write) read (3 x write) read (3 x
write) (2 x read) write read (6 x write) (4 x read) (2 x write) read (3 x write) (2 x read)
write (2 x read) lseek (2 x read) (2 x write) read lseek read mmap write (4 x read) (8 x
write) read (2 x write) read (4 x write) read write (3 x read) write (3 x read) (3 x write)
(6 x read) (4 x write) (2 x read) write (2 x read) write read write read (5 x write) (2 x
read) (2 x write) (2 x read) (3 x write) read write (2 x read) (2 x write) read write lseek
(3 x write) (5 x read) (2 x write) read (3 x write) read write (3 x read) (2 x write) (3 x
read) (5 x write) (2 x read) (5 x write) lseek read write read (2 x write) read (3 x write)
read (5 x write) (3 x read) (5 x write) read (5 x write) read (2 x write) (3 x read) write
lseek read write lseek (2 x read) (5 x write) read lseek (8 x write) read write (2 x read)
(3 x write) (2 x read) (5 x write) lseek read write read (3 x write) read (2 x write) lseek
read write read lseek (7 x write) read (3 x write) read (2 x write) read (5 x write) (2 x
read) write (2 x read) write (2 x read) write read (3 x write) (2 x read) write read close
read lseek open
```

Figure B.14: The best mimicry attack against pHsm for restore (second part)

chdir open rt_sigaction (3 x open) read time open (2 x read) close (2 x read) (2 x fstat) open read close open read close open read close rt_sigaction open read close read open (2 x read) open time write (2 x read) close fstat close open close open (2 x read) rt_sigaction (2 x munmap) close (2 x open) (2 x rt_sigaction) (2 x read) time open time fstat rt_sigaction time close open read open fstat read rt_sigaction (3 x close) read time (3 x close) read time rt_sigaction time read time (2 x open) rt_sigaction close read time close (2 x read) time fstat open close (2 x read) fstat time read time (2 x open) close open read close read time read (3 x close) fstat read (2 x time) close (5 x open) close open mmap open fstat open mmap open fstat close fstat time close read close open read fstat (3 x open) rt_sigaction (2 x open) time rt_sigaction munmap (2 x close) (2 x open) rt_sigaction (2 x read) time close fstat close mmap rt_sigaction read close time close (2 x read) (2 x open) (2 x close) read close (3 x read) munmap time close time rt_sigaction open read time (2 x close) open close open read (5 x open) fstat close fstat time (2 x close) rt_sigaction read munmap time read fstat read (2 x time) (2 x read) time (3 x close) fstat (2 x open) time open close (2 x read) close open close (2 x read) close read open mmap (3 x open) read close open read (3 x open) mmap open fstat mmap fstat read close (4 x read) (2 x open) (2 x close) read open (3 x read) munmap time close fstat read open read time (2 x close) open close open read open close open close open fstat close fstat (3 x close) read rt_sigaction (2 x close) mmap (2 x open) read (2 x close) time (2 x read) rt_sigaction time close mmap (3 x open) (3 x read) time close (2 x read) time (2 x open) time close open read close rt_sigaction fstat read fstat read (2 x open) mmap open read close rt_sigaction munmap (3 x close) open time (3 x close) read time rt_sigaction time close time (2 x open) rt_sigaction open read time close (2 x read) time fstat (2 x open) close (2 x read) rt_sigaction time close mmap (2 x open) (4 x read) time close (2 x read) time (3 x open) close open read close rt_sigaction fstat read fstat read (4 x open) read close rt_sigaction munmap (2 x close) open (2 x close) rt_sigaction read open fstat read (2 x open) close open close open (2 x read) open time close open read (3 x open) mmap close (3 x fstat) time mmap (2 x open) (4 x close) open (2 x time) fstat (2 x open) close (3 x read) (2 x open) read (3 x close) open (2 x time) read close read time (2 x read) time fstat (2 x open) close open (2 x read) rt_sigaction read open rt_sigaction read close (3 x read) open time fstat read time open time fstat open time close open read time

Figure B.15: The best mimicry attack against pHsm for ftpd (first part)

(2 x open) close open read close read time read (4 x close) read time (2 x close) open close (2 x open) (2 x close) open mmap open fstat close fstat time (2 x close) rt_sigaction read munmap time fstat (2 x read) (2 x time) read rt_sigaction time close fstat time fstat open fstat rt_sigaction read close (2 x read) open (2 x time) fstat (2 x open) close open (2 x read) (2 x open) (3 x close) read open (2 x time) read close (2 x time) (2 x read) open read mmap rt_sigaction fstat close (4 x read) (2 x time) close time close open read fstat (2 x open) close open read rt_sigaction read munmap close (2 x read) open (2 x time) fstat (2 x open) close read open read fstat open (4 x close) open close time read close (2 x time) (2 x read) open read mmap rt_sigaction fstat close read open close open mmap open fstat close fstat time (2 x close) rt_sigaction read munmap time (2 x fstat) read (2 x time) read rt_sigaction time close time close fstat (2 x open) time read close (2 x read) close open close (2 x read) close (2 x open) mmap (4 x open) close open read (3 x open) mmap open fstat open fstat time close open rt_sigaction read munmap (2 x read) mmap read (3 x close) fstat read time (2 x close) open close read (2 x open) (2 x close) read close open read time fstat (2 x open) close open (2 x read) close (2 x read) time fstat (2 x open) read open read close rt_sigaction munmap close rt_sigaction (2 x read) write read open (2 x read) open time write (2 x read) open fstat (2 x open) close open (2 x read) rt_sigaction munmap (2 x close) (2 x open) time rt_sigaction (2 x read) time open time fstat rt_sigaction time close open read open fstat read rt_sigaction (3 x close) read time (3 x close) read time rt_sigaction open close time (2 x open) rt_sigaction (2 x read) time close (2 x read) open fstat open rt_sigaction (2 x read) fstat time read time (2 x open) close open read close read time read close open time fstat read time (4 x close) (3 x open) close open mmap open fstat open mmap open fstat close fstat time close read close open read fstat (3 x open) rt_sigaction open read time rt_sigaction munmap open close time open rt_sigaction read rt_sigaction time close fstat close mmap rt_sigaction read close (5 x read) open (3 x close) open (3 x read) munmap time close fstat rt_sigaction open read time (2 x close) open close open read open close open mmap time fstat close fstat (2 x close) (2 x read) rt_sigaction (2 x close) mmap (2 x open) close (3 x read) time close (2 x read) time fstat (2 x open) close read fstat open (2 x close) open (3 x read) munmap fstat close open read open rt_sigaction (2 x open) close read close read (2 x open) (3 x close) (2 x open) read open close read rt_sigaction (3 x close) read open close open rt_sigaction (2 x open) write close

Figure B.16: The best mimicry attack against pHsm for ftpd (second part)

B.4 Best Mimicry Attacks against the Markov Model

brk (2 x mmap) close (2 x write) gettimeofday (6 x write) munmap open fstat mmap close (3 x write) gettimeofday write gettimeofday (3 x write) gettimeofday (9 x write) gettimeofday (2 x write) gettimeofday write gettimeofday (4 x write) gettimeofday write gettimeofday write gettimeofday (3 x write) munmap brk (8 x write) gettimeofday write gettimeofday select write gettimeofday (3 x write) munmap write munmap (3 x write) gettimeofday write gettimeofday select (2 x write) gettimeofday write gettimeofday (6 x write) gettimeofday (5 x write) gettimeofday write munmap (5 x write) gettimeofday write gettimeofday write munmap write gettimeofday write gettimeofday (3 x write) gettimeofday (7 x write) gettimeofday select (4 x write) gettimeofday (4 x write) munmap open fstat mmap close (6 x write) gettimeofday (3 x write) munmap (2 x brk) (2 x write) gettimeofday (5 x write) gettimeofday (6 x write) munmap (3 x write) munmap open fstat mmap close (3 x write) gettimeofday write gettimeofday write gettimeofday (3 x write) gettimeofday (4 x write) gettimeofday (4 x write) gettimeofday (2 x write) gettimeofday (6 x write) gettimeofday write gettimeofday write gettimeofday (3 x write) munmap brk (8 x write) gettimeofday write gettimeofday select write gettimeofday (3 x write) munmap write munmap (9 x write) gettimeofday (4 x write) gettimeofday (2 x write) gettimeofday (6 x write) gettimeofday write gettimeofday write gettimeofday (3 x write) munmap brk (8 x write) gettimeofday write gettimeofday select write gettimeofday (3 x write) munmap write munmap (5 x write) gettimeofday select (2 x write) gettimeofday write gettimeofday write munmap write gettimeofday write gettimeofday (3 x write) gettimeofday (3 x write) munmap write fstat (2 x mmap) close (3 x write) gettimeofday write gettimeofday select (3 x write) munmap (8 x write) gettimeofday write gettimeofday (3 x write) munmap (3 x write) gettimeofday write gettimeofday select (2 x write) gettimeofday write gettimeofday (12 x write) gettimeofday write munmap (3 x write) gettimeofday write gettimeofday (3 x write) munmap (3 x write) gettimeofday write gettimeofday (3 x write) gettimeofday (9 x write) munmap open fstat mmap close munmap (2 x write) gettimeofday (5 x write) gettimeofday write gettimeofday write gettimeofday (3 x write) munmap brk (8 x write) gettimeofday write gettimeofday select write gettimeofday (3 x write) munmap write munmap (3 x write) gettimeofday write gettimeofday select (2 x write) gettimeofday write gettimeofday (6 x write) gettimeofday (5 x write)

Figure B.17: The best mimicry attack against the Markov Model for traceroute (first part)

gettimeofday write munmap (5 x write) gettimeofday write gettimeofday (3 x write) gettimeofday write gettimeofday (3 x write) gettimeofday (7 x write) gettimeofday select (2 x write) gettimeofday write gettimeofday (4 x write) munmap write fstat mmap close (3 x write) gettimeofday (6 x write) munmap brk gettimeofday (7 x write) gettimeofday (3 x write) gettimeofday write gettimeofday (3 x write) munmap (3 x write) gettimeofday write gettimeofday select (2 x write) gettimeofday write gettimeofday (6 x write) gettimeofday (5 x write) gettimeofday write munmap (3 x write) gettimeofday (5 x write) munmap write gettimeofday write gettimeofday (3 x write) gettimeofday (7 x write) gettimeofday select (2 x write) gettimeofday write gettimeofday (16 x write) gettimeofday write munmap (3 x write) gettimeofday write gettimeofday (3 x write) munmap (3 x write) gettimeofday (3 x write) gettimeofday (2 x write) gettimeofday (4 x write) gettimeofday select (2 x write) gettimeofday write gettimeofday (5 x write) munmap open fstat mmap close (3 x write) gettimeofday write gettimeofday write gettimeofday (2 x write) gettimeofday (3 x write) munmap brk (3 x write) gettimeofday (5 x write) gettimeofday (3 x write) gettimeofday (2 x write) munmap (3 x write) munmap open fstat mmap close (3 x write) gettimeofday write gettimeofday (3 x write) gettimeofday (2 x write) gettimeofday (2 x write) munmap (3 x write) munmap open fstat mmap close munmap (2 x write) gettimeofday write gettimeofday (3 x write) gettimeofday (4 x write) gettimeofday (11 x write) gettimeofday (3 x write) gettimeofday write gettimeofday (3 x write) munmap (3 x write) gettimeofday write gettimeofday select (2 x write) gettimeofday write gettimeofday (6 x write) gettimeofday (5 x write) gettimeofday write munmap write gettimeofday select (9 x write) munmap write fstat (2 x mmap) close (3 x write) gettimeofday write gettimeofday select (3 x write) munmap (8 x write) gettimeofday write gettimeofday (3 x write) munmap (3 x write) gettimeofday write gettimeofday select (2 x write) gettimeofday write gettimeofday (7 x write) munmap open fstat mmap close (3 x write) gettimeofday write gettimeofday (3 x write) gettimeofday (2 x write) gettimeofday (2 x write) munmap (3 x write) munmap open fstat mmap close munmap (2 x write) gettimeofday write gettimeofday (3 x write) gettimeofday (4 x write) gettimeofday (6 x write) gettimeofday (7 x write) gettimeofday (3 x write) gettimeofday (2 x write) munmap (3 x write) munmap open fstat mmap close (6 x write) gettimeofday (3 x write) munmap brk write fstat (2 x mmap) close write

Figure B.18: The best mimicry attack against the Markov Model for traceroute (second part)

fcntl64 stat munmap _llseek munmap (2 x stat) geteuid32 stat _llseek munmap time (2 x fcntl64) read stat fcntl64 munmap stat (2 x fcntl64) (2 x stat) time munmap (4 x stat) _llseek munmap stat (3 x munmap) (2 x time) fcntl64 close munmap (3 x stat) fcntl64 stat geteuid32 stat munmap _llseek munmap fcntl64 stat geteuid32 (2 x stat) munmap time munmap fcntl64 munmap (2 x stat) geteuid32 stat munmap _llseek munmap (2 x stat) time stat _llseek munmap time mmap2 (2 x fcntl64) (2 x stat) fcntl64 munmap (2 x fcntl64) (2 x stat) fcntl64 (2 x stat) munmap time stat fcntl64 munmap fcntl64 stat munmap stat geteuid32 stat (2 x munmap) time fcntl64 read stat (2 x munmap) (2 x stat) fcntl64 stat munmap time munmap (2 x stat) geteuid32 stat _llseek munmap (3 x stat) munmap _llseek munmap (2 x stat) munmap time fcntl64 stat open fcntl64 open stat fcntl64 (2 x stat) geteuid32 stat _llseek munmap time fcntl64 (2 x munmap) _llseek munmap fcntl64 read (2 x stat) mmap2 _llseek read stat munmap (2 x stat) mmap2 time munmap close stat fcntl64 munmap (2 x fcntl64) (2 x stat) munmap time fcntl64 munmap time stat (2 x fcntl64) munmap (2 x stat) fcntl64 geteuid32 (2 x stat) (2 x munmap) (3 x stat) geteuid32 stat _llseek munmap time fcntl64 read stat munmap fcntl64 stat munmap fcntl64 stat munmap time munmap stat (2 x geteuid32) stat fcntl64 munmap (2 x stat) fcntl64 (2 x munmap) (3 x stat) geteuid32 (2 x stat) munmap open fcntl64 (2 x stat) (2 x fcntl64) stat time _llseek munmap (2 x stat) fcntl64 munmap stat _llseek write (3 x stat) mmap2 time read close select munmap write close stat _llseek munmap time (2 x fcntl64) (2 x munmap) stat geteuid32 stat munmap _llseek munmap stat time geteuid32 stat fcntl64 munmap time mmap2 (2 x fcntl64) close stat fcntl64 munmap (2 x fcntl64) (3 x stat) time fcntl64 stat fcntl64 time (2 x stat) _llseek munmap time munmap (2 x stat) geteuid32 stat _llseek munmap time (2 x fcntl64) read stat fcntl64 munmap stat (2 x fcntl64) (2 x stat) time munmap (2 x stat) geteuid32 stat _llseek (5 x munmap) time stat fcntl64 close (2 x munmap) (2 x stat) munmap stat geteuid32 stat munmap _llseek munmap (2 x stat) geteuid32 (2 x stat) munmap time munmap fcntl64 munmap (2 x stat) geteuid32 time munmap stat munmap (2 x stat) time stat _llseek munmap (2 x time) fcntl64 (3 x stat) fcntl64 munmap (2 x fcntl64) stat mmap2 fcntl64 (2 x stat) munmap time stat fcntl64 munmap fcntl64 stat munmap (2 x geteuid32) stat (2 x munmap) time fcntl64 read stat munmap fcntl64 stat (2 x fcntl64) stat munmap time munmap (2 x stat) geteuid32 time _llseek munmap (3 x stat) munmap _llseek munmap (2 x stat) munmap time fcntl64 stat open fcntl64 open fcntl64 munmap (2 x

Figure B.19: The best mimicry attack against the Markov Model for samba (first part)

stat) geteuid32 stat _llseek munmap time (2 x fcntl64) munmap fcntl64 munmap fcntl64 read stat time mmap2 read stat (2 x fcntl64) stat munmap fcntl64 stat munmap time (3 x munmap) geteuid32 stat _llseek munmap (3 x stat) munmap _llseek munmap (3 x stat) fcntl64 geteuid32 stat munmap time (2 x stat) (2 x munmap) (3 x stat) geteuid32 (2 x stat) time stat _llseek munmap (2 x time) munmap (3 x stat) fcntl64 munmap (2 x fcntl64) (2 x stat) fcntl64 (2 x stat) munmap time stat fcntl64 munmap fcntl64 (2 x stat) (2 x geteuid32) stat (2 x munmap) time fcntl64 read stat munmap fcntl64 stat (2 x fcntl64) stat munmap time munmap (2 x stat) geteuid32 time _llseek munmap (3 x stat) mmap2 time read close select munmap write close stat _llseek munmap time (2 x fcntl64) (2 x munmap) stat geteuid32 stat munmap _llseek munmap (2 x stat) geteuid32 stat fcntl64 munmap time mmap2 (2 x fcntl64) close stat fcntl64 munmap (2 x fcntl64) (3 x stat) time fcntl64 stat fcntl64 time (2 x stat) _llseek munmap time munmap (2 x stat) geteuid32 stat (2 x munmap) time (2 x fcntl64) read stat fcntl64 munmap stat (2 x fcntl64) (2 x stat) time munmap (2 x stat) geteuid32 stat _llseek munmap fcntl64 (3 x munmap) time stat fcntl64 close (2 x munmap) (2 x stat) munmap stat geteuid32 stat munmap _llseek munmap (2 x stat) geteuid32 (2 x stat) munmap time munmap fcntl64 munmap (2 x stat) geteuid32 time munmap stat munmap (2 x stat) time stat _llseek munmap (2 x time) fcntl64 (3 x stat) fcntl64 munmap (2 x fcntl64) (2 x stat) fcntl64 (2 x stat) munmap time stat fcntl64 munmap fcntl64 stat munmap (2 x geteuid32) time (2 x munmap) time fcntl64 read (2 x stat) fcntl64 stat (2 x fcntl64) stat munmap time munmap (2 x stat) geteuid32 time _llseek munmap (3 x stat) munmap _llseek munmap (2 x stat) munmap time fcntl64 stat open fcntl64 open fcntl64 munmap (2 x stat) geteuid32 stat _llseek munmap time (2 x fcntl64) munmap fcntl64 munmap fcntl64 read stat time mmap2 read stat (2 x fcntl64) stat munmap fcntl64 stat munmap time (3 x munmap) geteuid32 stat _llseek munmap (3 x stat) munmap _llseek munmap (3 x stat) fcntl64 geteuid32 stat munmap time (2 x stat) (2 x munmap) (3 x stat) geteuid32 (2 x stat) time stat _llseek munmap (2 x time) munmap (3 x stat) fcntl64 munmap (2 x fcntl64) (2 x stat) fcntl64 (2 x stat) munmap time stat fcntl64 munmap fcntl64 (2 x stat) (2 x geteuid32) stat (2 x munmap) time fcntl64 read stat munmap fcntl64 stat (2 x fcntl64) stat munmap time munmap (2 x stat) geteuid32 time _llseek munmap (4 x stat) _llseek munmap (2 x stat) munmap time (2 x fcntl64) open fcntl64 open stat munmap (2 x stat) geteuid32 stat _llseek munmap time (2 x fcntl64) munmap fcntl64 munmap fcntl64 read stat time mmap2 read stat munmap (2 x stat) munmap fcntl64 stat munmap time (3 x munmap) geteuid32 stat _llseek munmap (3 x stat) munmap _llseek munmap stat (2 x munmap) stat geteuid32 stat fcntl64 stat (3 x munmap) time stat fcntl64 munmap (4 x stat) _llseek munmap time mmap2 (2 x fcntl64) close stat fcntl64 munmap fcntl64 (2 x stat) fcntl64 (2 x stat) _llseek munmap time mmap2 (2 x fcntl64) stat _llseek munmap time fcntl64 read stat munmap fcntl64 stat fcntl64 (2 x stat) _llseek munmap time (2 x fcntl64) munmap stat munmap geteuid32 stat munmap _llseek munmap (4 x stat) _llseek munmap time (2 x fcntl64) munmap (2 x stat) geteuid32 stat munmap stat (2 x munmap) time stat munmap write close select

Figure B.20: The best mimicry attack against the Markov Model for samba (second part)

lseek (2 x read) (3 x write) lseek read lseek (2 x open) (3 x write) (2 x lseek) open (2 x write) read open read open read open write lseek (3 x read) lseek fstat mmap read (2 x lseek) open write (2 x read) open (2 x read) lseek open read lseek write read (2 x write) lseek read (3 x write) read (3 x lseek) open write (2 x read) write lseek read (2 x write) lseek (2 x write) (2 x read) (3 x lseek) fstat mmap read open write read write lseek (2 x open) (2 x read) lseek read (3 x open) (4 x read) lseek open read (2 x write) (2 x read) lseek (2 x open) write (2 x read) (2 x lseek) open read write lseek (2 x write) read write lseek (4 x open) (2 x read) lseek open (2 x write) read (2 x open) write (3 x read) lseek open read mmap open fstat mmap read (2 x lseek) open write read (2 x lseek) read write (2 x read) lseek write (2 x read) open read lseek (2 x read) open read lseek open read (3 x lseek) open write (2 x read) (2 x lseek) read (2 x write) (2 x read) lseek (3 x open) read write lseek open (3 x write) lseek open (2 x write) read (2 x write) (2 x read) lseek read open write (2 x read) lseek (2 x open) write read open write (4 x read) lseek (2 x read) mmap open fstat mmap read (4 x lseek) (2 x read) write (5 x read) lseek open read (2 x lseek) (2 x read) (2 x open) read open (2 x read) lseek open read (2 x write) (2 x read) (2 x lseek) open write read lseek open read mmap read lseek write read lseek open fstat mmap rt_sigprocmask read (2 x open) write read (2 x write) read (3 x lseek) open write (2 x read) write lseek read (2 x write) lseek (2 x write) (2 x read) (3 x lseek) fstat mmap read open write read write lseek (2 x open) (2 x read) lseek (2 x read) (2 x open) read open (2 x read) lseek open read (2 x write) (2 x read) lseek (2 x open) write (2 x read) (2 x lseek) open read write lseek (2 x write) read write lseek (4 x open) (2 x read) lseek open (2 x write) read (2 x open) write (3 x read) lseek open read mmap open fstat mmap read (2 x lseek) open write read (2 x lseek) read write (2 x read) lseek write (2 x read) open read lseek (2 x read) open read lseek open read (3 x lseek) open write (2 x read) (2 x lseek) read (2 x write) (2 x read) lseek (3 x open) read write lseek open (3 x write) lseek open (2 x write) read open write (2 x read) lseek read open write (2 x read) lseek (2 x open) write read open write (4 x read) lseek (2 x read) mmap open fstat mmap read (4 x lseek) (2 x read) write (5 x read) lseek open read (2 x lseek) (2 x read) (2 x open) read open (2 x read) lseek open read (2 x write) (2 x read) (2 x lseek) open write read lseek open read mmap read lseek write (2 x

Figure B.21: The best mimicry attack against the Markov Model for restore (first part)

lseek) open fstat mmap rt_sigprocmask read (2 x open) write read (2 x lseek) (2 x open) write read write (2 x lseek) open write read (2 x open) read open (2 x read) lseek open read lseek (3 x read) lseek (2 x open) (2 x write) read (2 x lseek) open (2 x write) lseek (2 x write) (2 x read) lseek (2 x open) (3 x write) read (2 x write) read lseek write (2 x read) (2 x lseek) read open write (3 x read) lseek open write (3 x read) lseek open write lseek read (2 x write) read write lseek (3 x open) read (3 x open) fstat read write (2 x lseek) open write (2 x read) (2 x lseek) open read mmap read lseek write (2 x lseek) open fstat mmap rt_sigprocmask read (2 x open) write (2 x read) lseek read open write read (2 x write) lseek open write read (2 x open) read open (2 x read) (2 x open) read write (3 x read) lseek (2 x open) write (2 x read) (2 x lseek) open (2 x write) lseek (2 x write) (2 x read) lseek open read (3 x write) read write read (2 x write) lseek read (3 x write) read (4 x lseek) write (2 x read) (2 x lseek) read (2 x write) lseek (2 x write) (2 x read) (3 x lseek) fstat mmap (2 x read) write read write lseek (2 x open) read (2 x lseek) write read (2 x open) read open (2 x read) lseek open read (2 x write) (2 x read) lseek (2 x open) write (2 x read) (2 x lseek) open read write lseek (2 x write) read write lseek (3 x open) write (2 x read) (2 x lseek) (2 x write) read (2 x open) write (3 x read) lseek open read mmap open fstat mmap read lseek (2 x open) write read (2 x lseek) read (2 x write) read lseek write (2 x read) lseek read lseek (2 x read) open read lseek open read open write (2 x read) lseek (2 x open) write read write (2 x lseek) open (2 x write) lseek (2 x write) (2 x read) lseek read lseek fstat mmap read lseek read lseek (2 x write) read lseek open write read (3 x lseek) (3 x read) lseek open read write (3 x lseek) (2 x open) (2 x write) read write (2 x read) (2 x write) lseek read (2 x lseek) read open write (3 x read) lseek open write (3 x read) lseek open (2 x write) read (2 x write) read write lseek (2 x open) (2 x read) (3 x open) fstat read write (2 x lseek) open write (2 x read) (2 x lseek) open read mmap read lseek write (2 x lseek) open fstat mmap rt_sigprocmask read open read write (2 x read) lseek read open write read lseek write lseek open write read (2 x open) read open (2 x read) lseek open read write (3 x read) (3 x open) write (2 x read) lseek read open (2 x write) lseek (2 x write) (2 x read) lseek (2 x open) (3 x write) (2 x read) lseek open (4 x read) write read open read (2 x mmap) (4 x read) (3 x open) (2 x write) (2 x read) open write close

Figure B.22: The best mimicry attack against the Markov Model for restore (second part)

time rt_sigaction open write open (4 x rt_sigaction) open (2 x write) close munmap close read write rt_sigaction close write (2 x close) read close read close read (2 x write) close (2 x read) alarm read close open (2 x write) (4 x close) (2 x write) rt_sigaction close open write rt_sigaction alarm close read alarm read close read write read close read close write open write open write open rt_sigaction alarm open rt_sigaction open read close open read rt_sigaction open write (3 x open) read (2 x close) open read close read close getcwd open read (2 x close) open (2 x write) open read write close open read close read close rt_sigaction open write rt_sigaction alarm write rt_sigaction open read (2 x close) rt_sigaction open read (3 x close) open read close write close getcwd open read close read write close write (2 x open) read rt_sigaction open rt_sigaction alarm read (2 x rt_sigaction) close read (3 x close) write open read close (3 x write) rt_sigaction close read mmap write close open rt_sigaction open (2 x rt_sigaction) close munmap write rt_sigaction alarm read close read open read (2 x close) open write rt_sigaction open (2 x read) close open read close write close rt_sigaction open write close (2 x write) rt_sigaction open read (3 x close) open read (2 x close) (2 x read) open write (2 x close) open write alarm close munmap close read write (3 x close) write close read close read close read write rt_sigaction (2 x close) munmap close open read alarm read close read (2 x write) close rt_sigaction close write open write rt_sigaction (2 x close) write rt_sigaction alarm close write alarm read open read write read (2 x close) read write (4 x close) (3 x write) close read mmap write close read rt_sigaction open (2 x rt_sigaction) close munmap write rt_sigaction alarm read close read write read (2 x close) open (2 x write) open (2 x read) (2 x close) read close read close getcwd open write close (2 x write) rt_sigaction open read (3 x close) open read (3 x close) write open write close write close rt_sigaction close write open write read close open write rt_sigaction alarm close read alarm read close read write read close read close write (3 x open) write open rt_sigaction alarm read (2 x write) read close open read rt_sigaction open write (2 x open) (2 x read) (2 x close) open read close read close getcwd open read (2 x close) open (2 x write) open read write close open read close read close getcwd open write rt_sigaction alarm write rt_sigaction open read (2 x close) getcwd open read rt_sigaction alarm write rt_sigaction open read close read (2 x open) (2 x read) close (2 x read) close open read close read close getcwd open write rt_sigaction open write rt_sigaction open read (2 x close) rt_sigaction open read close read close open write close write (2 x open) (2 x read) close write open write rt_sigaction alarm (2 x write) open read (2 x close) rt_sigaction open read (3 x close) open rt_sigaction close (2 x read) alarm read close read write read close read open

Figure B.23: The best mimicry attack against the Markov Model for ftpd (first part)

write open write (2 x close) open write rt_sigaction close read close open read close read (2 x write) open (2 x read) (2 x close) read close read close getcwd open read rt_sigaction alarm write rt_sigaction open rt_sigaction (2 x close) read rt_sigaction alarm write rt_sigaction open read (2 x close) rt_sigaction open read (3 x close) open write (2 x close) (2 x write) (2 x close) write close rt_sigaction open write open read rt_sigaction open write (3 x open) read write close open read close read close getcwd open read close read write close read (2 x close) read close read write read (3 x close) (2 x open) write (3 x close) write (2 x open) write close read mmap read close open rt_sigaction open rt_sigaction alarm read rt_sigaction open read write open read close open write (2 x open) (2 x read) close open write close read (2 x close) read write read close read close write open write close read open read rt_sigaction close read write open read close read write read close read close write open write (2 x close) open write rt_sigaction close read close open read close open (2 x write) open (2 x read) close open read close read close getcwd open rt_sigaction close write open write (2 x close) open write read close (2 x read) open read close open write rt_sigaction close (2 x read) write open read rt_sigaction alarm write rt_sigaction open read close read (2 x open) (2 x read) close (2 x read) close open read close read close getcwd open write rt_sigaction open write rt_sigaction open read (2 x close) rt_sigaction open read (3 x close) open write close write (2 x open) (2 x read) close write open write rt_sigaction alarm write rt_sigaction open read (2 x close) rt_sigaction open read (3 x close) open rt_sigaction close (2 x read) alarm read close read write read close read open write open write (2 x close) (2 x open) rt_sigaction close read close open read close read (2 x write) open (2 x read) close open read close read close getcwd open read rt_sigaction alarm write rt_sigaction open rt_sigaction (2 x close) read rt_sigaction alarm write rt_sigaction open read (2 x close) rt_sigaction open read (3 x close) open write close (3 x write) (2 x close) write close rt_sigaction open write open read rt_sigaction open write (3 x open) read close rt_sigaction open read close read close getcwd open read close read write close rt_sigaction (2 x close) read close read write read (3 x close) (2 x open) write (3 x close) write (2 x open) write close read mmap read close open rt_sigaction open rt_sigaction alarm read rt_sigaction open read write open read rt_sigaction open write (2 x open) (2 x read) close open write close read (2 x close) read write read close read close write close write close read open read rt_sigaction close read close open read close open (2 x write) open (2 x read) close open read alarm read close getcwd open write rt_sigaction alarm write rt_sigaction open read (2 x close) rt_sigaction open (2 x read) (2 x close) open write close (3 x write) (2 x close) (2 x write) rt_sigaction open write open read write close open

Figure B.24: The best mimicry attack against the Markov Model for ftpd (second part)

B.5 Best Mimicry Attacks against the Neural Network

mmap uname write connect (2 x gettimeofday) sendto gettimeofday mprotect mmap gettimeofday
ioctl connect fstat write read close uname connect open uname munmap mmap open (2 x mmap)
close uname mmap open fstat close recvfrom (2 x gettimeofday) select gettimeofday ioctl
mprotect sendto brk write gettimeofday (2 x mmap) uname gettimeofday fstat close mmap
(2 x open) mmap write gettimeofday sendto close open fstat select poll open close select
uname gettimeofday fstat close mmap write mmap open gettimeofday close mmap brk mmap send
read fstat read gettimeofday (2 x open) gettimeofday brk (2 x mmap) (2 x brk) select send
close read mmap close mmap write send connect (2 x read) select gettimeofday recvfrom
(2 x mmap) close read gettimeofday connect socket close sendto gettimeofday open uname
close poll mmap mprotect open gettimeofday munmap gettimeofday fstat sendto recvfrom
write fstat mmap select read brk mmap ioctl munmap (2 x mmap) read sendto fstat sendto
gettimeofday ioctl select ioctl socket (2 x mprotect) fstat gettimeofday send gettimeofday
write (2 x mmap) write close sendto poll write gettimeofday munmap mmap fcntl munmap
mmap read mprotect write (3 x mmap) close mmap select mprotect close select gettimeofday
mprotect write mprotect read close open send recvfrom mmap close select close mprotect
write socket gettimeofday read close read socket open mmap gettimeofday uname close read
fstat write fcntl open socket fcntl gettimeofday poll fstat read uname brk close read
sendto munmap gettimeofday write close uname recvfrom mprotect (2 x gettimeofday) poll
open munmap open mmap fstat (2 x mmap) recvfrom gettimeofday open (2 x close) write mmap
(2 x gettimeofday) send fcntl gettimeofday write (3 x gettimeofday) ioctl gettimeofday
sendto write read close mmap close sendto fstat sendto munmap read close gettimeofday close
read uname gettimeofday mmap poll gettimeofday mmap close (2 x mmap) close uname write
fcntl select fcntl gettimeofday munmap mmap munmap mprotect fstat mprotect fstat open mmap
gettimeofday write sendto mprotect brk (2 x close) poll (2 x close) brk mmap open fstat
close munmap close connect recvfrom mmap fstat poll gettimeofday (2 x mmap) gettimeofday
write ioctl close (2 x open) recvfrom mmap (2 x sendto) send munmap gettimeofday sendto
close gettimeofday read mmap gettimeofday brk sendto open fstat send mmap close write mmap
sendto connect munmap mmap

Figure B.25: The best mimicry attack against the Neural Network for traceroute (first part)

ioctl munmap mmap read brk poll gettimeofday (2 x write) read close gettimeofday write gettimeofday read uname gettimeofday close mmap uname fcntl connect (3 x write) select mmap gettimeofday read ioctl brk gettimeofday open read connect open gettimeofday munmap mmap (2 x open) write socket read open write poll mmap read sendto write fstat mmap socket sendto (2 x gettimeofday) select mmap socket (2 x fcntl) send read write mmap (2 x munmap) fstat brk (2 x open) write munmap read close open fcntl mmap (2 x gettimeofday) select mmap open mmap write fstat select mprotect select recvfrom close poll recvfrom close connect read gettimeofday fstat mmap fstat recvfrom write gettimeofday poll sendto gettimeofday mmap gettimeofday mprotect brk fstat socket poll open munmap send (2 x munmap) open recvfrom (4 x write) fcntl mmap munmap read connect gettimeofday mprotect fcntl read sendto close mmap read close munmap gettimeofday select sendto mmap munmap close mmap open read socket gettimeofday brk read select fstat (2 x mmap) fstat mmap open mprotect read mmap munmap close socket open write uname read select close uname read gettimeofday fstat close sendto read brk read close mprotect select gettimeofday mmap open munmap select recvfrom (2 x write) close write read gettimeofday mprotect munmap open (2 x read) munmap fstat open (2 x read) close mmap sendto brk gettimeofday fstat close munmap (2 x open) fstat open fstat gettimeofday mmap (2 x recvfrom) gettimeofday write (2 x mmap) uname gettimeofday socket gettimeofday munmap recvfrom select open fcntl select munmap read (2 x fstat) open recvfrom sendto fcntl (2 x close) mprotect sendto munmap close fstat read fstat close fstat (2 x mmap) gettimeofday socket poll mmap open fstat (2 x gettimeofday) open select open mmap gettimeofday fcntl close read write socket close (2 x mprotect) select fstat mmap gettimeofday ioctl mmap gettimeofday fstat mprotect close brk write fcntl mmap read gettimeofday open fstat mprotect write open select open read fstat read mmap brk mmap open mmap fstat (2 x sendto) (2 x close) gettimeofday fstat close munmap fstat munmap gettimeofday read fcntl mprotect poll sendto fstat mmap gettimeofday open brk select read select uname sendto gettimeofday open mprotect open munmap (2 x write) fcntl close fcntl write sendto select read (2 x munmap) mprotect socket close uname mprotect socket close mmap gettimeofday socket open mmap poll mprotect gettimeofday open write sendto close brk ioctl select mmap brk gettimeofday uname

Figure B.26: The best mimicry attack against the Neural Network for traceroute (second part)

```
gettimeofday close read open recvfrom munmap close sendto mmap (2 x fstat) munmap mmap
(2 x gettimeofday) fcntl read gettimeofday connect munmap read munmap fstat open mprotect
send open write mmap brk mprotect gettimeofday open mmap open close recvfrom gettimeofday
sendto close fcntl fstat write (2 x gettimeofday) connect write read fstat mprotect close
mmap read sendto read munmap fcntl gettimeofday munmap open mmap brk select mmap close
read munmap select close recvfrom fcntl write uname mmap ioctl read close (2 x write)
gettimeofday read fstat gettimeofday read close brk gettimeofday write recvfrom mmap munmap
mprotect close write (3 x fstat) read gettimeofday mmap sendto fstat connect mmap (2 x
gettimeofday) open gettimeofday munmap brk sendto recvfrom sendto brk sendto open mmap
recvfrom select recvfrom (2 x mmap) munmap brk socket fstat mmap close open fstat close
connect close read open ioctl write fcntl connect fstat open select mmap close write socket
close munmap select gettimeofday (2 x open) gettimeofday munmap mprotect (3 x gettimeofday)
mmap socket gettimeofday open read mmap gettimeofday open close gettimeofday open write
fcntl write close connect mmap select mmap gettimeofday write mmap fstat uname close write
close mmap read gettimeofday close sendto select gettimeofday ioctl send mmap read select
mprotect recvfrom munmap close mprotect close ioctl open connect socket open sendto close
open uname connect read (2 x fstat) write close gettimeofday sendto fcntl close write
uname mmap open select send sendto send mmap select write close recvfrom fstat open mmap
fstat send gettimeofday write sendto gettimeofday close munmap close ioctl close sendto
gettimeofday fstat close gettimeofday (2 x close) read mmap gettimeofday open recvfrom
fcntl write fstat connect open uname
```

Figure B.27: The best mimicry attack against the Neural Network for traceroute (third part)

read open stat munmap geteuid32 close open munmap fcntl64 close _llseek (2 x munmap) stat open write open gettimeofday fcntl64 _llseek select munmap getegid32 close getsockopt mmap gettimeofday mmap2 (2 x gettimeofday) open (2 x stat) select gettimeofday munmap open munmap mmap2 select write geteuid32 umask fcntl64 gettimeofday mmap _llseek select (2 x close) write _llseek munmap stat open gettimeofday _llseek stat write close mmap2 getsockopt select fcntl64 _llseek write munmap (3 x read) geteuid32 stat close munmap select munmap geteuid32 munmap close mprotect read (2 x munmap) open stat select open fcntl64 getegid32 stat fcntl64 read write close mmap send select _llseek gettimeofday fcntl64 _llseek mmap _llseek close munmap time close mprotect close umask write mprotect munmap stat time mmap close stat fcntl64 (2 x stat) munmap getsockopt stat select _llseek close read stat fcntl64 read close gettimeofday read mmap fcntl64 munmap mmap getsockopt write open getsockopt read select mprotect send getegid32 open send geteuid32 stat fcntl64 stat close fcntl64 (2 x gettimeofday) read close fcntl64 _llseek mprotect munmap _llseek (2 x munmap) fcntl64 mmap stat close mprotect munmap (2 x read) _llseek (2 x fcntl64) umask write read munmap getegid32 (2 x select) munmap select umask (2 x close) getegid32 close (3 x read) select _llseek mmap _llseek geteuid32 close fcntl64 umask stat munmap stat read stat (2 x mmap) _llseek select time (2 x _llseek) write time mmap2 munmap fcntl64 stat close gettimeofday read fcntl64 stat fcntl64 _llseek send stat read mprotect mmap2 stat munmap (2 x _llseek) geteuid32 select mmap stat fcntl64 stat select (2 x read) mprotect mmap2 munmap read select (3 x munmap) umask stat read _llseek (2 x munmap) getsockopt munmap close time stat mmap2 _llseek fcntl64 read stat mmap geteuid32 stat open mmap _llseek fcntl64 close munmap mmap geteuid32 mmap stat getsockopt read stat munmap (2 x stat) munmap (2 x read) mmap getegid32 (2 x _llseek) mmap read munmap mmap2 stat munmap send _llseek mmap2 open select munmap mmap _llseek open fcntl64 close stat (3 x munmap) fcntl64 _llseek mprotect _llseek umask stat mmap stat (2 x _llseek) munmap open close stat fcntl64 close (2 x select) geteuid32 open read (2 x munmap) read close stat read stat read stat _llseek stat fcntl64 close open stat open stat read _llseek munmap fcntl64 close stat _llseek select _llseek (2 x open) select time gettimeofday close stat read umask mprotect select mmap read munmap getsockopt mprotect close time _llseek read close munmap read _llseek munmap open read open close select mmap2 time close stat read mmap close stat gettimeofday select munmap getsockopt munmap fcntl64 munmap read stat fcntl64 mmap _llseek umask _llseek read stat open stat munmap close mprotect time _llseek munmap _llseek read geteuid32 umask close read getsockopt geteuid32 close select geteuid32 read _llseek mmap _llseek mmap geteuid32 select close setresgid32 _llseek fcntl64 (2 x munmap) (2 x fcntl64) getegid32 setresgid32 write gettimeofday (2 x read) fcntl64 _llseek open mprotect send mmap (2 x munmap) select time _llseek open getegid32 open stat gettimeofday getegid32 write stat read open munmap time umask read fcntl64 read _llseek geteuid32 munmap write _llseek gettimeofday munmap stat select geteuid32 mmap open read select open munmap mmap2 getegid32 select stat fcntl64 geteuid32

Figure B.28: The best mimicry attack against the Neural Network for samba (first part)

(2 x getsockopt) read (2 x munmap) geteuid32 gettimeofday open (2 x stat) munmap send _llseek select close umask munmap open stat mprotect munmap fcntl64 mmap2 _llseek gettimeofday _llseek time mmap2 _llseek (2 x stat) fcntl64 gettimeofday umask fcntl64 open close fcntl64 mmap open read gettimeofday read close munmap close stat select stat write munmap (2 x fcntl64) gettimeofday (2 x _llseek) read time read _llseek fcntl64 stat (2 x munmap) mmap open mmap2 read select close fcntl64 munmap setresgid32 select fcntl64 (2 x read) mmap munmap read send munmap fcntl64 open close write read write mmap _llseek select _llseek stat mmap _llseek stat open getegid32 stat fcntl64 mprotect stat gettimeofday munmap mprotect geteuid32 mmap close munmap time munmap getsockopt (2 x munmap) stat _llseek read fcntl64 select mmap2 (2 x getsockopt) setresgid32 stat mmap fcntl64 munmap close mmap stat send stat mmap mmap2 select munmap _llseek read munmap stat fcntl64 getegid32 write close munmap mprotect open munmap read gettimeofday umask open _llseek time close select stat fcntl64 stat write gettimeofday _llseek time select (2 x munmap) stat (2 x mmap) write munmap open close _llseek getegid32 _llseek stat time mmap fcntl64 close mmap2 setresgid32 close fcntl64 getsockopt stat munmap read fcntl64 mmap stat read munmap mmap2 select mprotect close (2 x stat) mmap read munmap geteuid32 _llseek (2 x fcntl64) getegid32 munmap open read _llseek mmap fcntl64 select mprotect open stat close gettimeofday stat close munmap stat munmap umask close select close mmap close munmap umask close select send (2 x munmap) getsockopt (2 x getegid32) close fcntl64 munmap open stat fcntl64 _llseek getsockopt stat select mmap geteuid32 select mmap stat munmap mprotect open (2 x munmap) _llseek (2 x stat) close _llseek geteuid32 munmap _llseek setresgid32 (2 x mmap2) munmap time _llseek mmap2 munmap read mprotect mmap2 close select read geteuid32 _llseek read fcntl64 geteuid32 read close getegid32 setresgid32 munmap select open (2 x fcntl64) munmap open read stat read mprotect mmap (2 x munmap) write mmap2 (2 x _llseek) send geteuid32 _llseek open fcntl64 close munmap setresgid32 munmap getegid32 (2 x munmap) close time mmap2 munmap getsockopt mmap gettimeofday close gettimeofday getsockopt munmap open getsockopt stat munmap geteuid32 _llseek read mprotect mmap close (2 x stat) open stat open (2 x stat) mmap read close munmap fcntl64 mmap2 mmap read gettimeofday mmap fcntl64 read munmap select stat gettimeofday open fcntl64 _llseek gettimeofday munmap _llseek fcntl64 select read munmap time close fcntl64 munmap mmap close fcntl64 close stat mmap2 munmap open (2 x munmap) fcntl64 read (2 x close) fcntl64 open time gettimeofday mmap fcntl64 (2 x munmap) close munmap write setresgid32 write mmap stat mmap2 stat write fcntl64 read (2 x munmap) fcntl64 stat select close munmap fcntl64 getegid32 fcntl64 geteuid32 gettimeofday mmap read stat open munmap fcntl64 munmap mmap fcntl64 stat munmap _llseek close fcntl64 send stat fcntl64 close umask (2 x write) munmap fcntl64 time stat (2 x getegid32) fcntl64 mmap write open fcntl64 select munmap mprotect fcntl64 open mprotect read stat munmap (3 x read) open select send _llseek mmap open stat fcntl64 read open read mmap open munmap fcntl64 _llseek fcntl64 munmap close _llseek munmap read (2 x munmap)

Figure B.29: The best mimicry attack against the Neural Network for samba (second part)

fcntl fstat brk fcntl munmap open mprotect mmap write open fcntl write open mmap close (2 x brk) mprotect fstat mprotect mmap mprotect open mmap open (3 x mmap) mprotect fcntl open chmod open brk write fstat mmap read mprotect open close write open read brk mmap close read open (2 x mprotect) brk fstat chmod lseek write mmap fstat open read (2 x close) (2 x mprotect) (2 x fstat) mprotect fstat chmod fcntl (3 x mmap) mprotect read brk write brk (2 x write) (2 x mprotect) open mmap write fstat write open close open ioctl open chmod mmap write mmap close chmod open fstat write fstat lseek open fcntl mmap mprotect write (2 x mmap) munmap open close write open write ioctl chmod (2 x close) chmod close mmap mprotect (2 x open) mmap (2 x fstat) (3 x write) close lseek mmap chown (2 x open) mmap close lseek fstat open mmap open utime write mmap write fstat mmap open mmap open lseek mmap mprotect mmap write mmap fstat utime mprotect brk open write close open write mmap open brk open brk write fstat mmap (2 x open) fcntl close unlink open fstat (2 x write) (2 x lseek) (2 x close) mmap write mmap write open write (2 x open) lseek (2 x mmap) write fcntl chmod mmap close chmod utime open write fstat close fstat open (2 x mmap) (2 x open) mmap mprotect (2 x fstat) fcntl write close fcntl mmap _llseek fstat close write mmap close (2 x open) mprotect unlink open close lseek open mmap close mprotect write fstat mmap write mprotect write stat fcntl open lseek write brk open fstat open stat mmap fstat mmap open fstat write open mprotect (2 x fstat) mmap write close open close fstat close fstat mmap brk mprotect mmap mprotect write open write open (2 x write) mmap read fstat close fstat write close brk mmap fstat mmap mprotect (2 x write) read mmap fstat mmap (2 x fstat) mmap close mmap fstat close open mprotect mmap stat close brk (2 x fstat) lseek fstat write fcntl open fstat brk fstat mmap open write fstat chown write mmap write open fstat open close mprotect mmap open mmap close mmap write mprotect mmap write mprotect chown lseek write chmod (2 x fstat) lseek mmap close fstat stat open fstat (2 x write) chown write mmap mprotect close rt_sigaction mprotect fstat open fstat open fcntl stat brk open (2 x write) close write mmap open (2 x mmap) (3 x fstat) mmap (2 x open) fcntl close (2 x mprotect) close fstat write mmap fcntl brk mmap fstat (2 x write) ioctl (2 x mmap) unlink write read lseek chmod open write stat write (3 x mmap) open (2 x chmod) brk fstat open (2 x fstat) write mmap brk mmap open read mmap open (3 x mmap) open (2 x mmap) fstat (2 x mmap) brk mmap mprotect (2 x mmap) write mmap write open write mmap open mmap lseek fstat (2 x brk) chown (2 x open) write lseek fcntl brk fstat utime (2 x mmap) (2 x open) fstat fcntl open mmap fcntl lseek fstat brk mmap mprotect close unlink (2 x write)

Figure B.30: The best mimicry attack against the Neural Network for restore (first part)

(2 x close) lseek (2 x open) write open mmap write munmap open close (2 x open) (2 x mmap) open write mmap open close (2 x write) mprotect brk open mmap write (2 x open) fcntl open chown chmod write mprotect mmap close mmap mprotect open (3 x mmap) write mprotect lseek (2 x open) (2 x fstat) (2 x write) (2 x chown) open mmap open fstat mmap fstat ioctl close mmap write close mmap write brk mmap open mmap fstat brk close write mmap open mmap chmod (2 x fstat) open write (2 x fstat) ioctl mprotect open mmap write mmap close brk ioctl write fstat chown read mprotect chmod write mmap open write stat write close (2 x brk) open close write mmap open write fcntl mmap fstat unlink close (2 x mmap) write close brk write lseek open fcntl mmap close utime open mmap chown mmap (2 x write) lseek open mprotect mmap fstat chown mmap open ioctl (2 x mmap) (2 x open) write fcntl brk (3 x mmap) open fstat mmap write close mmap fcntl fstat close fstat open mmap chown (2 x fstat) mprotect (2 x mmap) fcntl mmap chown lseek ioctl mmap fstat brk (2 x mmap) (4 x open) write fstat mprotect lseek mmap open mprotect open (2 x fstat) mmap brk write open write mmap mprotect write open mprotect write close open write read mprotect write fstat lseek mprotect mmap mprotect open write close (2 x mmap) chmod fcntl open fstat mmap open write open mmap utime mprotect write fstat open mmap open chown (2 x open) mmap fstat write mmap fstat mmap (2 x write) stat write close (2 x fstat) lseek write fstat write brk open mmap open mprotect (2 x write) close (2 x mmap) close (2 x open) fstat brk chown (2 x write) fcntl close mmap chmod mmap open write mmap chown close mprotect close fstat chmod write open fstat open write mmap mprotect lseek (2 x write) chown mmap chown (2 x mmap) chmod fstat write open (2 x mmap) open mprotect mmap mprotect close open chmod write mprotect brk (2 x fstat) mmap open mprotect mmap write fstat mmap chmod write fstat mmap open close mmap ioctl (2 x fstat) mprotect (2 x write) lseek close write mmap brk fstat mmap write mmap chown fstat write mmap fstat chown mmap fstat mmap open (2 x write) fstat mmap open mmap lseek mmap open (2 x mmap) (3 x fstat) stat unlink mmap write open write fstat fcntl mmap (2 x open) mprotect (2 x fstat) munmap chmod mmap fstat mmap mprotect fstat write open fstat open brk mmap mprotect mmap lseek write lseek write (2 x fstat) mprotect fstat mprotect mmap open mmap open fcntl open (2 x chmod) chown brk mprotect brk mmap brk chmod mmap fstat write (3 x open) write mmap open mmap open lseek mmap write fcntl mmap open close fstat mprotect chown open mprotect mmap (4 x open) (2 x mmap) fstat (2 x mprotect) close (2 x write) read (2 x mmap) mprotect fstat (2 x write) fstat (2 x open) write mmap open stat close mmap fstat (2 x mmap) mprotect close open (2 x write) open mmap write fstat (2 x mmap) write brk mmap write lseek

Figure B.31: The best mimicry attack against the Neural Network for restore (second part)

close mmap write mmap socket mprotect rt_sigaction (3 x close) alarm mmap (4 x close) write connect (2 x close) chdir rt_sigaction write close read close rt_sigaction mprotect (3 x close) open socket (4 x close) read write read (2 x close) read brk fcntl chdir (2 x close) open (2 x close) getcwd fstat close fstat close socket close read socket close time close munmap read open mmap (2 x close) rt_sigaction (2 x close) read (2 x close) fcntl close chdir close open (2 x close) read open (4 x close) socket close read fcntl rt_sigaction close send close read alarm (2 x close) socket close socket close chdir open rt_sigaction brk read close time brk (3 x close) read open write open close read (4 x close) brk open fstat close rt_sigaction close chdir close write (2 x close) fstat (2 x close) read chdir close getcwd read mmap munmap (3 x close) rt_sigaction (2 x close) (2 x rt_sigaction) mmap (2 x close) read close getcwd close munmap close alarm fstat (2 x close) socket read write send rt_sigaction close read (4 x close) rt_sigaction fcntl rt_sigaction mmap open close write (2 x read) close alarm read (6 x close) mmap (2 x close) mmap read (3 x close) send (2 x close) rt_sigaction (5 x close) rt_sigaction close read brk fstat rt_sigaction mmap close rt_sigaction write close mmap socket (5 x close) rt_sigaction close write close rt_sigaction (2 x close) socket rt_sigaction read (2 x close) rt_sigaction (2 x close) open close chdir brk close send open send time write close write close open close socket close fchdir (2 x close) mmap close socket close write send (2 x close) mprotect (2 x close) mmap open close open close brk fstat (2 x close) read close (2 x open) (2 x close) brk close read rt_sigaction getcwd close brk read close fstat close rt_sigaction close read close socket write close mmap close open close munmap (3 x close) (2 x read) socket (2 x read) close (2 x rt_sigaction) close write close socket mmap read (3 x close) rt_sigaction write munmap (5 x close) chdir (2 x mprotect) (3 x read) mmap close read close rt_sigaction getcwd rt_sigaction munmap send (3 x close) fcntl fstat mmap (2 x close) open close write open close mmap (2 x munmap) close rt_sigaction (2 x close) chdir rt_sigaction close fstat write (2 x close) rt_sigaction fstat (2 x close) read mprotect close (3 x open) mmap close write (3 x close) connect munmap write read rt_sigaction close send brk (5 x close) socket close munmap close mmap read munmap close munmap close read (2 x close) mmap brk (3 x close) brk close write fstat read close rt_sigaction read (3 x close) read

Figure B.32: The best mimicry attack against the Neural Network for ftpd (first part)

(8 x close) alarm read close mmap (2 x close) brk rt_sigaction (2 x close) chdir (3 x close) getcwd (2 x close) rt_sigaction close rt_sigaction read socket (2 x close) read send (2 x close) (2 x open) (6 x close) (2 x write) brk (3 x close) read (2 x mmap) mprotect close socket send (4 x close) open mmap (5 x close) read (2 x close) send close read (2 x close) fcntl (3 x close) mmap write close munmap fcntl mprotect mmap (2 x close) read rt_sigaction socket close fcntl close brk munmap brk read rt_sigaction (2 x close) alarm (3 x close) open fcntl read brk close read rt_sigaction (5 x close) rt_sigaction close mmap close write close send close read (4 x close) rt_sigaction close read close getcwd (2 x close) mmap (2 x close) read (3 x close) munmap close write close rt_sigaction (8 x close) rt_sigaction (2 x close) alarm chdir (2 x close) mprotect brk (2 x close) rt_sigaction close mmap close open (2 x close) mmap (2 x close) brk fstat (2 x close) socket (2 x read) socket close write (2 x close) send getcwd close open socket (3 x close) read fcntl write close mmap close read socket fstat close fstat close read (2 x close) read close brk write close (2 x open) close alarm (3 x close) send socket (2 x close) connect rt_sigaction chdir (4 x close) open mprotect (4 x close) read mprotect write rt_sigaction munmap mmap close write time close socket close read (3 x close) brk (4 x close) socket (3 x close) rt_sigaction read brk (4 x close) read close rt_sigaction mmap close write alarm (2 x close) read brk close (2 x munmap) alarm (8 x close) send socket close getcwd (4 x close) rt_sigaction (2 x close) send (2 x close) (2 x read) close munmap open close read close connect open fcntl (2 x read) close mprotect close read open socket rt_sigaction close open brk (2 x close) write close socket (2 x close) open brk (2 x read) rt_sigaction read (3 x close) read fstat close rt_sigaction close mmap (3 x close) mmap mprotect (3 x close) brk connect rt_sigaction read (2 x close) mmap read alarm rt_sigaction close getcwd close brk (3 x close) munmap (2 x close) munmap (2 x close) fstat chdir rt_sigaction (2 x close) connect write (3 x close) brk (3 x close) open close mmap (6 x close) send (5 x close) read open read close brk send (4 x close) mmap (2 x close) rt_sigaction (3 x close) send (3 x close) brk close read close read write (4 x close) fstat (4 x close) socket open (10 x close) open brk rt_sigaction close rt_sigaction open mmap open read (2 x close) rt_sigaction send munmap chdir (2 x close) mprotect close brk close send open read close rt_sigaction (2 x close) fcntl (2 x close) fstat (2 x close) mmap rt_sigaction close chdir close fcntl close send rt_sigaction (3 x close) chdir (8 x close) rt_sigaction read close chdir (5 x close) rt_sigaction (4 x close) read (2 x close) read rt_sigaction write (4 x close) brk fstat (3 x close) read close (2 x open) close fcntl munmap

Figure B.33: The best mimicry attack against the Neural Network for ftpd (second part)

Appendix C

Linux i386 System Calls

In Chapter 10, the mimicry attacks are analysed in terms of the system calls which they execute to hide the true intent of the attack. Such an analysis includes categorization of system calls where each category of system calls specialize in facilitating a particular operating system function.

The categorization and description of system calls which are employed in Section 10.2.6, are detailed further in this appendix. The list is compiled from various resources such as the Linux Assembly website[1] (in particular the documentation prepared by Boldyshev [8]) and the Kernel development source codes[2]. An installation of Red Hat 9.0 with Kernel Development Packages installs the relevant source codes at `/usr/src/linux-2.4.20-8` where 2.4.20-8 is the kernel version number and may change depending upon the operating system version. New system calls are added and the existing system calls are updated and rearranged as new Linux distributions emerge, therefore this appendix should be considered as a guideline on what each system call does and where it is located as opposed to a definitive and final source of information. The reader should visit the Linux Assembly website for the updated version of system call definitions.

According to Boldyshev [8], system calls can be categorized according to where they are defined in the Kernel source. To this end, Boldyshev [8] provides six kernel sources where system calls are defined, namely arch/i386/, fs/, ipc/, kernel/, mm/, net/. Each category is discussed separately in the following subsections. For more information on kernel source codes and system call definitions, the reader is encouraged to see the Linux Assembly website.

[1]The URL for the Linux Assembly website, as of January 2009, is `http://asm.sourceforge.net/`.

[2]The Linux Kernel Archives website contains the kernel source codes for current and past Linux kernels at `http://www.kernel.org/`, as of January 2009.

C.1 Architecture-Dependent System Calls (arch)

Architecture-dependent system calls are defined in this category. i386 is known as IA-32 as well, which comprise the 32 bit processor architectures. The system calls in this category include numerous system calls which have a non-standard calling sequence on the Linux/i386 platform and other architecture-dependent system calls which involve process handing and signalling.

- clone
- execve
- fork
- idle
- ioperm
- iopl
- ipc
- modify_ldt
- old_mmap
- olduname
- pause
- pipe
- ptrace
- rt_sigreturn
- rt_sigsuspend
- sigaction

- sigaltstack
- sigreturn
- sigsuspend
- uname
- vfork
- vm86
- vm86old

C.2 File System-Related System Calls (file)

This category focuses on the system calls related to file system operations such as opening, closing, reading and modifying a file. Furthermore, this category contains system calls which support the random access to files and system calls which modify the ownership and permissions of a file.

- access
- bdflush
- chdir
- chmod
- chown
- chroot
- close
- creat
- dup
- dup2

- fchdir
- fchmod
- fchown
- fcntl
- fdatasync
- flock
- fstat
- fstatfs
- fsync
- ftruncate
- getcwd
- getdents
- ioctl
- lchown
- link
- llseek
- lseek
- lstat
- mkdir
- mknod
- mount

- newfstat
- newlstat
- newstat
- nfsservctl
- oldumount
- open
- poll
- pread
- pwrite
- quotactl
- read
- readlink
- readv
- rename
- rmdir
- select
- stat
- statfs
- symlink
- sync
- sysfs

- truncate
- umount
- unlink
- uselib
- ustat
- utime
- utimes
- vhangup
- write
- writev

C.3 System Calls Related to Inter-Process Communication (ipc)

Inter-process communication enables communication between the threads of a process or different processes. System calls which facilitate inter-process communications are defined in this category.

- msgctl
- msgget
- msgrcv
- msgsnd
- semctl
- semget
- semop
- shmat

- shmctl
- shmdt
- shmget

C.4 System Calls Related to Kernel Functions (kernel)

The Linux kernel is responsible for managing hardware resources and providing communication between the applications and the hardware resources. The system calls relevant to Kernel functions are defined in this category. To name a few, the system calls in this category handles process management, timing and the priority scheduling of the processes.

- acct
- adjtimex
- adjtimex
- alarm
- capget
- capset
- create_module
- create_module
- delete_module
- exit
- get_kernel_syms
- getegid
- geteuid

- getgid
- getgroups
- gethostname
- getitimer
- getpgid
- getpgrp
- getpid
- getppid
- getpriority
- getresgid
- getresuid
- getrlimit
- getrusage
- getsid
- gettimeofday
- getuid
- init_module
- kill
- nanosleep
- newuname
- ni_syscall

- nice
- prctl
- query_module
- reboot
- rt_sigaction
- rt_sigpending
- rt_sigprocmask
- rt_sigqueueinfo
- rt_sigtimedwait
- sched_get_priority_max
- sched_get_priority_min
- sched_getparam
- sched_getscheduler
- sched_rr_get_interval
- sched_setparam
- sched_setscheduler
- sched_yield
- setdomainname
- setfsuid
- setgid
- setgroups

- sethostname
- setitimer
- setpgid
- setpriority
- setregid
- setresgid
- setresuid
- setreuid
- setrlimit
- setsid
- settimeofday
- setuid
- sgetmask
- signal
- sigpending
- sigprocmask
- ssetmask
- stime
- sysctl
- sysinfo
- syslog

- time
- times
- umask
- wait4
- waitpid

C.5 System Calls Related to Memory Management (memory)

System calls related to memory management operations such as allocating, deallocating and locking memory resources are defined in this category.

- brk
- mlock
- mlockall
- mprotect
- mremap
- msync
- munlock
- munlockall
- munmap
- sendfile
- swapoff
- swapon

C.6 System Calls Related to Network Communications (network)

System calls which facilitate network communication are defined in this category, such as establishing a socket connection and transmitting data over the socket.

- accept
- bind
- connect
- getpeername
- getsockname
- getsockopt
- listen
- recv
- recvfrom
- recvmsg
- send
- sendmsg
- sendto
- setsockopt
- shutdown
- socket
- socketcall
- socketpair