Applied Soft Computing xxx (2010) xxx–xxx

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

# Can a good offense be a good defense? Vulnerability testing of anomaly detectors through an artificial arms race

Hilmi Güneş Kayacık[a,*], A. Nur Zincir-Heywood[b], Malcolm I. Heywood[b]

[a] Carleton University, School of Computer Science, 1125 Colonel By Drive, Ottawa, ON K1S 5B6, Canada
[b] Dalhousie University, Faculty of Computer Science, 6050 University Avenue, Halifax, NS B3H 1W5, Canada

ARTICLE INFO

*Article history:*
Received 28 April 2010
Received in revised form 3 August 2010
Accepted 28 September 2010
Available online xxx

*Keywords:*
Computer security
Intrusion detection
Evasion attacks
Genetic Programming
Arms race

ABSTRACT

Intrusion detection systems, which aim to protect our IT infrastructure are not infallible. Attackers take advantage of detector vulnerabilities and weaknesses to evade detection, hence hindering the effectiveness of the detectors. To do so, attackers generate evasion attacks which can eliminate or minimize the detection while successfully achieving the attacker's goals. This work proposes an artificial arms race between an automated 'white-hat' attacker and various anomaly detectors for the purpose of identifying detector weaknesses. The proposed arms race aims to automate the vulnerability testing of the anomaly detectors so that the security experts can be more proactive in eliminating detector vulnerabilities.

## 1. Introduction

An intrusion detection system (IDS) is a combination of software and hardware that collects and analyses data from networks and hosts to determine if there is an attack [1] and possibly react to it as in the case of intrusion prevention systems [2]. 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. 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 [3]. On the other hand, anomaly detectors adopt the opposite approach, which is, to know what is normal, and then find the deviations from normal behavior. These deviations are considered as anomalies or possible intrusions. Anomaly detection systems rely on knowledge of normal behavior to detect attacks.

Naturally, intrusion detection systems are by no means infallible. Software vulnerabilities and hardware faults can cause them to misclassify or 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 may make use of detector vulnerabilities to alter their actions thus evading detection, rendering the detector ineffective. Although the evasion methodologies vary, the main objective of the attacker is to alter the attack so that it does not trigger signatures or generate anomalous behavior, while carrying out the attacker's goals.

In this work, we propose an arms race between artificial 'white-hat' attackers and candidate detectors. By 'white-hat' we imply an automated process for vulnerability testing (attack generation) without access to private information regarding the detector architecture. Feedback from the detector is limited to public information that a legitimate user might expect to receive; such as alarms. Such a scenario implies that we could deploy the same generic white-hat approach to any number of different detector architectures. Relative to the previous approaches discussed in Section 2, this means that we make extensive use of heuristic search – in this case Genetic Programming – to conduct the search for vulnerabilities under the guise of general objectives of an attack. Conversely, previous researchers made strong assumptions regarding the operation of specific detector architectures and face limitations on the subset of detectors they may evaluate (see [4] for a performance comparison between the two general approaches). We note that this work provides an empirical evaluation of an automated technique for vulnerability testing. Developing a theoretical model for testing detectors is beyond the scope of this work.

The proposed Evolutionary Exploit Generator (EEG) makes two assumptions (Section 3). There is a common goal of an attack – in this case adding the user to the root login file – and the system calls of the target application may be profiled. Given that the target

* Corresponding author.
*E-mail addresses:* kayacik@ccsl.carleton.ca (H.G. Kayacık), zincir@cs.dal.ca (A.N. Zincir-Heywood), mheywood@cs.dal.ca (M.I. Heywood).

1568-4946/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.asoc.2010.09.005

applications take the form of generic Linux applications, such information can be collected without accessing privileged information from the application itself. We use the set of most common application system calls to define the instruction set for GP. The fitness function is expressed as a multi-criteria objective rewarding alarm rate minimization, correct formulation of an exploit and – in the case of the more sophisticated detectors – delay minimization.

The basic EEG framework represents a generic framework for designing buffer overflow style attacks, where this represents a major class of intrusions [5]. However, in order to be effective, users need to be clear regarding what the concept of an attack represents, particularly with respect to the relative contributions of preamble and exploit to the construction of an attack (Section 3). With the nature of buffer overflow attacks established, we then introduce the vulnerable applications and the corresponding set of candidate detectors on which benchmarking will later be performed (Section 4).

The following benchmarking study emphasizes three themes (Section 5). Firstly it is important to evolve attacks with the preamble included. Earlier works have tended to ignore the contribution of this entirely and make the mistaken claim that detectors can be avoided with zero anomaly rate. Secondly, the delay mechanisms embedded in some detectors (i.e., the IDS is able to react to potential intrusions by delaying the running process) has the capacity to render all the original attack variants, and all but one EEG attack, ineffective; again as a consequence of the contribution of the preamble. Finally, we are able to characterize the strategies used by EEG to obfuscate the true intent of an attack, with different strategies clearly being adopted depending on the detector against which EEG is deployed.

Section 6 brings these findings together and in doing so recognizes that a major factor in the most successful attack is the corresponding succinctness of the preamble. That is to say, the finite size of the vulnerable buffers may enforce system call quotas in which the attack must be expressed. Consequently, if the preamble is short, then the corresponding relative contribution of the exploit component of an attack (the part the attacker has most control over) has much more impact on the overall alarm rate. Thus the relative count of instructions that do not conform to the normal application behavior profile is much lower. Naturally attacks with lengthy and anomalous preambles will be next to impossible to obfuscate.

## 2. Related work

Earlier works in vulnerability analysis make extensive use of knowledge regarding the internal design of the detector, with the emphasis being directed purely at the exploit. Wagner and Soto [6] investigated an approach to alter the system call sequences of an attack in order to render it undetectable to a specific IDS, namely Stide. Given a minimum sequence of malicious system calls to support execution of a successful attack – the *core* attack – their goal was to find other sequences of system calls that avoid detection by the target IDS yet still achieve the objective of the core attack. This was achieved by manually adding system calls that have no effect on the success of the attack. Similarly, Tan et al. [7] aimed to undermine the anomaly based IDS Stide [8] by identifying weaknesses and modifying the malicious system call sequences to exploit these limitations. To do so, they first modified the attack by hand to change the ownership of a critical file. Secondly, they inserted system calls from data characterizing normal behavior into the malicious system call sequence. Vigna et al. [9] described a methodology to generate variations of an attack to test the quality of detection signatures of Snort. Stochastic modification of attack code was employed to generate variants of attacks to render the attack undetectable. Techniques such as packet splitting, evasion and polymorphic shellcode were discussed. Kruegel et al. [10] developed a static analysis tool for Intel x86 binaries in order to automatically identify instructions that can be used to redirect control flow. They use symbolic execution to achieve this. Giffin et al. generated mimicry attacks against Stide by applying automatic model checking to prove that no reachable operating system configuration corresponds to the effect of an attack [11]. However, in their approach, the operating system model, application (program) model and system call specifications as well as the attack configuration are still generated manually.

On the other hand, our work contributes to the existing work on evasion attacks in two ways. Firstly, our approach represents an arms race between various anomaly detectors and artificial ‘white-hat’ attackers i.e., the Evolutionary Exploit Generator (EEG) framework. 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 attacks by providing feedback from the detector in the form of anomaly rates or other detection information such as the nature of dynamic measures deployed against an attack (delays). 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 result of the arms race is a set of evasion attacks, which can evade the target detector. The resulting attacks provide the defenders with crucial information that can be utilized to eliminate the weaknesses of the target detector. Needless to say, the exploits produced are entirely a result of the Evolutionary Exploit Generator with no hand crafting of the exploits.

Second, the previous work [6,7] assumed that the attacker can take control of the vulnerable application silently i.e., no consideration was given to the contribution of the preamble (Section 3.1) to attack detection. By contrast, in this work, we acknowledge that evasion attacks against anomaly detectors may not be as easy to perform in practice due to the attacker’s lack of control over the system calls executed before the attacker’s shellcode is invoked. Indeed, it readily becomes apparent that only when the preamble component of an attack contributes a significantly lower proportion of the attack code is it possible to evade the more sophisticated detectors (Section 5). We are also able to demonstrate that as the target detector changes, the composition of the attack controlled by EEG undergoes a significant change, implying that different detectors do have rather different implicit weaknesses.

## 3. Evolving buffer overflow attacks

In this section, the EEG framework is introduced to evolve attacks for analyzing vulnerabilities of detectors along with a brief discussion of buffer overflow attacks. Furthermore, we discuss the relevant work employing similar arms race methodologies for other attack types, although this work focuses on buffer overflow attacks.

### 3.1. Generic design decisions of a buffer overflow attack

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 against wu-ftpd server [12], 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 behavior.

After the attacker gains control of the application, the second step is to execute code to carry out a malicious action such as spawning a root shell or creating a super-user account. Commonly, this is referred to as the exploit and is achieved by injecting a shellcode (further discussed in Section 4.3). 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 Linux 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 with the objective of evading detection while simultaneously satisfying the goals of the exploit. On the other hand, modifying the preamble requires finding an alternative way to take advantage of the vulnerability or finding another vulnerability, and might therefore be more sensitive to modification. In short, the preamble sets up the exploit code to make use of an application vulnerability whereas the exploit performs a malicious action. Under an anomaly detection setting, both aim to minimize the likelihood of detection by using instructions that in some way also appear to be statistically no different from 'normal' behavior. In practice, we maintain that it is more difficult to achieve this for the preamble than the exploit, whereas earlier research has ignored the contribution of the preamble entirely when evaluating detector performance.

When analyzing system call traces, the boundary between the preamble and the exploit can be determined by locating the first action of the shellcode [13]. The four original attacks [14–16,12] employed in the analysis execute an *execve*('/*bin*/*sh*') system call to spawn a Linux shell with super-user privileges. Any system call including and after *execve*('/*bin*/*sh*') is a result of the spawned Linux 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. Although the previous works [17,7,18] were aware of the functional contribution of preambles, they assumed that the attacker could gain control of the vulnerable application silently. Specifically, Wagner and Soto [17] 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."

Our results indicate that the attacker may not always be able to take control of the application silently. Furthermore, our analysis of the preamble reveals that the actual penetration *does* leave anomalous system calls in the trace, at least in the cases of traceroute, restore, samba and ftpd applications benchmarked here. The experiments of Section 5 therefore compare the original and the evasion attacks generated by the EEG framework. Furthermore, results are reported for each component separately (i.e. preamble, exploit) as well as for the entire attack (i.e. preamble + exploit).

### 3.2. *Framework for Evolutionary Exploit Generation*

The process at the center of the proposed arms race involving the Evolutionary Exploit Generation (EEG) framework is 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 [19]. Each candidate solution, or individual, takes the form of a program i.e., a sequence of system calls in the case of this work. Although parameters for the system calls are specified, there is no need to support the specification of the internal state i.e., register values. This makes the resulting attacks *real exploits* as opposed to just system call sequences. The evolved system call sequences are transformed into executable shellcodes, as discussed in Section 4.3.

Aside from being able to conduct a stochastic search over a code based representation, GP provides several properties that make it a particularly attractive model for evolving exploits [20]. We consider the top three properties to take the form of:

**Representation**: Machine learning paradigms generally impose an *a priori* representation on the nature of a solution e.g., neurons in artificial neural networks, kernels in support vector machines or rules in decision tree induction. The representation required by the exploits imposes a system call based representation (although solutions based on the assembly language of the target platform would also be appropriate [21]). This precludes the utility of most machine learning algorithms. Thus, the automated variants of earlier research in this area have relied on exhaustive search, an option made possible by making extensive use of privileged information from the target detector e.g., the content of the detector's behavioral database post configuration (training) [17,18,22]. By assuming the representation of the actual application, the credit assignment process becomes more direct and support for including the preamble is straightforward (merely a question of concatenating preamble and exploit).

**Multi-criteria fitness**: Expressing exploit generation as a single objective – for example constructing code representing a valid exploit – would not sufficiently encompass the breadth of the task at hand. In particular, exploits need to achieve a malicious objective while simultaneously minimizing the alarm rate and reducing any delay imposed by dynamic/reactive detectors. One approach to doing this might be to merely linearly combine three performance functions into a single scalar value of fitness. However, Evolutionary Computation provides a much better mechanism in the form of Pareto multi-criteria formulations [23]. In particular, assuming a Pareto formulation implies that objectives are ranked relative to the number of other individuals they dominate in the Pareto sense. This avoids any need to impose arbitrary scaling of different objectives and encourages solutions to take the form of a front of non-dominated individuals as opposed to converging to a single possibly suboptimal solution.

**Obfuscation**: Code bloat or introns are a well known by-product of search in GP [19]. Although generally removed post-training, in this application, they are fundamental for providing the ability to obfuscate the resulting solutions. In short, it is the properties of the intron code that are used to model the statistical properties of normal behavior as measured by the alarm rate of the detector.

The principle EEG design decisions are now limited to defining the instruction set (representation) and search and selection operators (establishes basis for credit assignment), and establishing the appropriate feedback (goals/objectives) used to guide the process of evolution.

Fig. 1 details the components associated with the EEG and the arms race. Representation of the attacks and the application of the search and selection operators are handled by the Evolutionary Exploit Generator. Code injection component takes an exploit generated by EEG such as a system call sequence and converts it into an executable format such as a shellcode. The resulting executable is then injected to the vulnerable application using the injection method provided *a priori*, generally in the form of vulnerability reports [14,16,15,12]. In this framework, the detector is considered as a 'black-box' component, which implies the attacker (i.e. EEG) has only access to detection feedback which is produced as a normal operation of the detector. Attack validation component takes the detector feedback and converts it into a scalar value which supports the credit assignment process.

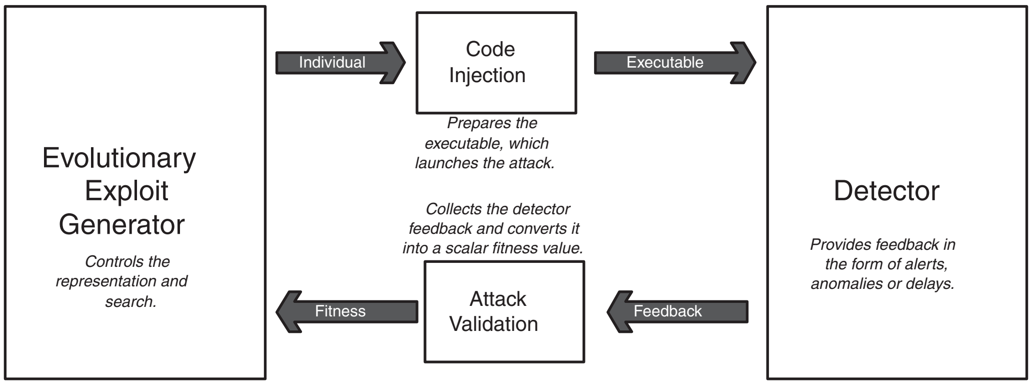
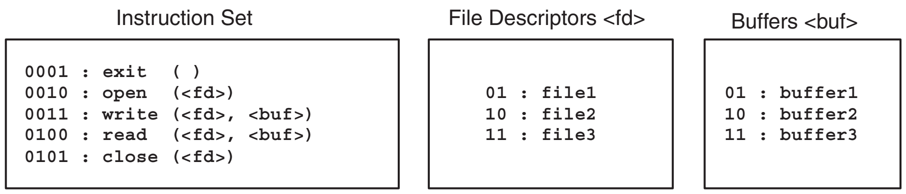

Evolutionary Exploit Generator
Controls the representation and search.
Individual
Code Injection
Prepares the executable, which launches the attack.
Executable
Detector
Provides feedback in the form of alerts, anomalies or delays.
Feedback
Attack Validation
Collects the detector feedback and converts it into a scalar fitness value.
Fitness

**Fig. 1.** EEG framework, which formulates the arms race.


Instruction Set
File Descriptors <fd>
Buffers <buf>

```
0001 : exit   ( )
0010 : open   (<fd>)
0011 : write  (<fd>, <buf>)
0100 : read   (<fd>, <buf>)
0101 : close  (<fd>)
```

```
01 : file1
10 : file2
11 : file3
```

```
01 : buffer1
10 : buffer2
11 : buffer3
```

**Fig. 2.** Sample EEG instruction set and parameters.

The representation that EEG employs is discussed in Section 3.2.1. Section 3.2.2 establishes the objectives for the attack validation component and Section 3.2.3 details the search and selection operators that EEG implements. The code injection methods are briefly discussed in Section 3.3, whereas the detectors employed in our framework are detailed in Section 4.2.

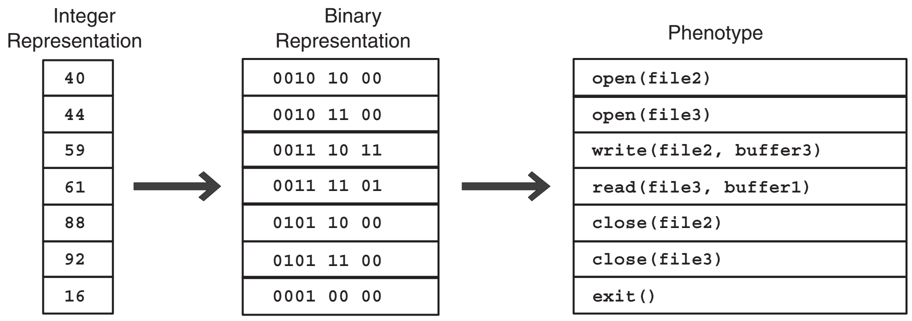
### *3.2.1. Representation*

We use knowledge of the top 20 system calls as utilized by each vulnerable application to define the appropriate subset of instructions (established in Section 4.1). From an attacker's perspective such an analysis involves recording the system calls on a local copy of the application and naturally does not require internal knowledge of the detector. The underlying assumption behind this is that instructions that are not in the application's top 20 will be more likely to raise alarms than instructions in the top 20.

The specific representation that EEG utilizes 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 (i.e. the system calls) to be used whereas the last two bytes identifies which terminal(s) (i.e. the system call parameters) the function uses.

A sample instruction set for EEG is provided in Fig. 2. The sample instruction set consists of system calls which can take file descriptors or memory locations (i.e. variables) as parameters, which are provided in Fig. 2 as well. Although EEG employs 4 bytes to define an instruction, the example provided in Figs. 2 and 3 employs 1 byte to define an instruction for the sake of simplicity.

Fig. 3 presents an EEG individual the genotype of which is represented in integer format. Based upon the mapping defined by the instruction set and parameters in Fig. 2, 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 third instruction from the top in Fig. 3 contains the integer value 59, which is equivalent to 00111011 in binary. The first four bits, 0011, map to the instruction `write (<fd>, <buf>)`. In the list of file descriptors in Fig. 2, the 10 maps to the file descriptor `file2` and the 11 maps to the buffer `buffer3`. If the instruction has one parameter (e.g. in case of integer value 44, which maps to `open(file3)`), the last two bits (i.e. 00) are ignored. As new instructions are introduced to the individuals – during the initialization of the population and the application of the mutation operator – the validity of the individuals is checked to ensure that the instruction and parameter fields produce a valid instruction.



| Integer Representation | Binary Representation | Phenotype |
|---|---|---|
| 40 | 0010 10 00 | open(file2) |
| 44 | 0010 11 00 | open(file3) |
| 59 | 0011 10 11 | write(file2, buffer3) |
| 61 | 0011 11 01 | read(file3, buffer1) |
| 88 | 0101 10 00 | close(file2) |
| 92 | 0101 11 00 | close(file3) |
| 16 | 0001 00 00 | exit() |

**Fig. 3.** An example of a genotype–phenotype mapping for an EEG individual.

(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

**Fig. 4.** Fitness function for establishing the objectives of modifying the Linux password file.

### 3.2.2. Fitness calculation and Pareto Ranking

Pareto Ranking is a method for combining multiple objectives under the concept of dominance [23]. 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 rank of an individual in our experiments is equal to the number of individuals which it dominates [24]. In terms of evasion attack characteristics to be optimized, the following objectives are considered:

1. *Attack success*. The original attack contains a standard shellcode, which uses the *execve* system call to spawn a Linux 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 will be detected easily. To this end, a different strategy is employed for defining the exploit such that the need to spawn a Linux shell is eliminated [21]. Typically, most programs perform I/O operations – in particular to open, write to/read from and close files. Therefore the goal of the attack is altered to involve the following three steps, which aim to gain super-user privileges:
   (a) open the Linux 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 EEG 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 behavioral success function rewarding the above behavior awards a total of 5 'points' for establishing the behavioral steps for the 'core' attack in Fig. 4.
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.
3. *Delay*. In addition to reporting anomaly rates, various anomaly detectors respond to anomalies by enforcing delays, as discussed in Section 4.2. Therefore, the attacker aims to minimize the delays associated with the attacks.

### 3.2.3. Search and selection operators

The search process progresses through the iterative application of selection and search operators. The selection operator is applied in two stages; in part 1, two individuals are identified from the population (the parents) with probability of selection proportional to the rank of the individual. Search operators are then applied to the individuals, resulting in two children. 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, or part 2 of the selection operator's role. Moreover, by pursuing a fitness function based on Pareto ranks, we are able to seamlessly incorporate multiple criteria into the performance evaluation without resorting to arbitrary combinations of unrelated objectives [24]. The training parameters, which remain the same over all applications, are detailed in Table 1. Individuals are defined using a variable length format, and population initialization creates individuals with varying program lengths.

**Table 1**
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 |

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 1. The specific details of each operator are detailed below.

**Cut and splice crossover**. The crossover operator provides a scheme for investigating instruction sequences that 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. Fig. 5 shows an example of cut and splice crossover.

**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, an example of which is shown in Fig. 6.

**Instruction-wise mutation**. The mutation operator provides a way to introduce new sequences to the individual. Mutation is

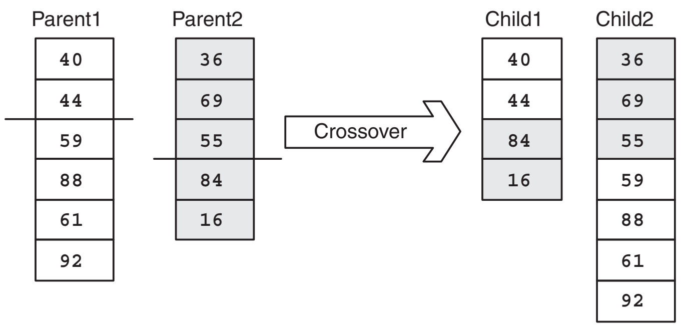


**Fig. 5.** An example of the cut and splice crossover operator.

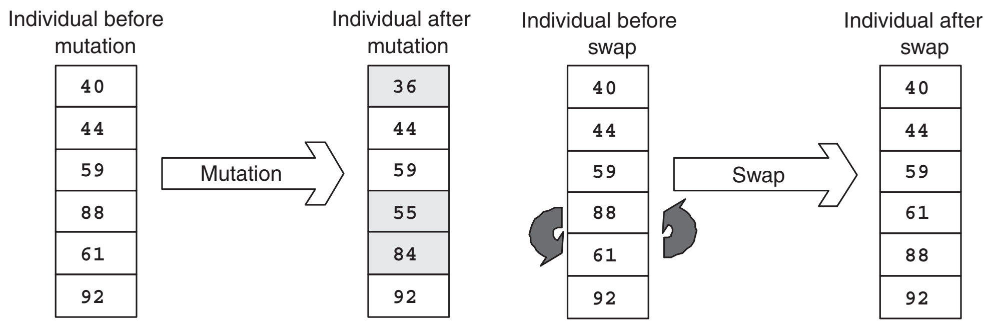


**Fig. 6.** Examples of the mutation and swap operators.

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 pre-defined list of instructions, as shown in Fig. 6. 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.

## 3.3. *Similar applications of the arms race*

While this work focuses on evolving buffer overflow attacks to evade anomaly detectors, various related work exist which formulate an arms race for generating different types of attacks. In order to apply the arms race on different types of attacks, the following design decisions need to be made:

**Representation of the exploits**. Depending on the vulnerability being exploited and the detector being evaded, a suitable language for evolving attacks should be defined, which constitutes the EEG instruction set. Examples of instruction sets are assembly instructions, system calls or network packet descriptions.
**Code injection method**. The attacker should be able to produce the attack executable to deploy the attack. Typically, this can be considered as a code injection task. Thus, the method in which the evolved attack is converted into an executable should be defined prior to the arms race. This implies that the attacker knows how to take advantage of the vulnerability. In the case of buffer overflow attacks, the exploits that the EEG generates in this work are embedded in C programs (provided in the vulnerability reports [14,16,15,12]) which exploits the vulnerability and facilitates the code injection.
**Interpretation of the detector feedback**. The feedback from the detector should be converted into a scalar value, upon which the fitness is calculated. This generally involves collecting the output that the detector generates as a part of its normal operation (such as the alerts, anomaly rates and other detection feedback) and calculating a fitness value. In case of evolving buffer overflow attacks, this involves collecting the anomaly rates and the delays associated with the attacks and combining them into a scalar value using Pareto Ranking.

Below, we overview various applications of similarly formulated arms race techniques.

Kayacık et al. [25] employed a similar arms race methodology to evolve shellcode on assembly level using Genetic Programming. The main goal is to evolve executable code that can deploy the attack successfully while evading the Snort misuse detector. The instruction set is a subset of the Intel 32-bit instruction set. The fitness function described the core objectives of the executable, namely: to deploy the attack successfully while remaining undetected. The detector feedback is obtained from Snort. The success of the attack is determined by a runtime environment, which simulates the execution of the assembly code. The results indicated that the arms race could discover different ways to deploy an attack by changing the composition and the ordering of the instructions. Evolving shellcode on assembly level complements the arms race proposed in this work since it provides means to transform the system call sequences into executable assembly code. In order to identify the weaknesses of network based detectors, Dozier et al. [26] proposed an arms race between ‘evolutionary hackers’ and artificial immune system-based detectors, in which a host was protected by a population of detectors. A Genetic Algorithm and various particle swarm search methods were employed to craft malicious network packets, which were then sent to the detectors. The fitness was the percentage of the detectors that failed to detect the attack. Their results indicated that the arms race identified numerous weaknesses by discovering malicious packet sequences, which evaded the detectors. Similarly, LaRoche et al. [27] proposed a GP-based approach to evolve TCP/IP packets for port scanning to evaded Snort misuse detector. Their attacks take the form of instruction sequences, which set the relevant fields of TCP and IP headers. Consequently, sequences of packets are generated which correspond to a port scan. The fitness is assigned based on the attack’s ability to discover open ports as well as the number of alarms that it raises. Their results demonstrated that GP succeeded in evolving port scan attacks, which evade Snort by using additional instructions to break the packets apart in time to remain below the detection thresholds. Utilizing a similar arms race, LaRoche and Zincir-Heywood [28] also evolved data link header fields to generate Denial of Service attacks on 802.11 networks against the Snort wireless misuse detector. While not formulated as an arms race, the work of Vigna et al. [9] provides an interaction between an automated attacker, which takes a set of attack templates and applies various mutation techniques such as packet splitting or using alternate encoding to evade network-based misuse detectors. The attack templates are generally buffer overflow attacks and the mutation techniques aim to preserve the validity of the attack while introducing additional control sequences or changing the composition. In addition to the feedback from Snort and ISS RealSecure detectors, the success of the mutant exploit was determined by checking if it writes to a pre-determined file. Their results demonstrated that all the baseline attacks (i.e. core attacks before mutation) are detected whereas majority of the mutated attacks evaded detection.

## 4. Vulnerable applications and candidate detectors

In this section, we introduce the set of vulnerable applications used in the later benchmarking study and therefore establish the

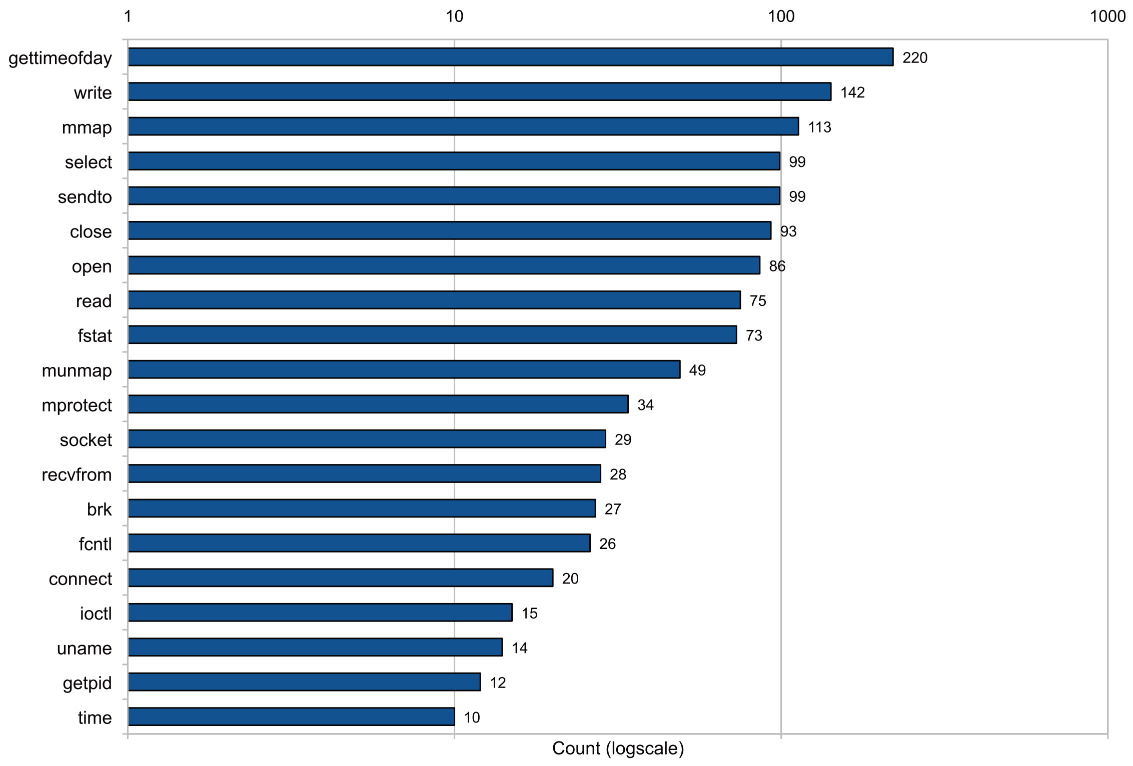


**Fig. 7.** Most frequent 20 system calls for traceroute.

application specific instruction set employed by EEG for evolving exploits.

The set of detectors, against which vulnerability testing is carried out, is then established where we provide a cross-section of detector complexity as well as covering different families of detector architecture. We then provide a discussion of how the evolved system call sequences are converted into executable shellcodes.

### 4.1. Vulnerabilities

In our arms race experiments, four Linux vulnerabilities are included: traceroute, samba, restore and ftpd, all of which have known and documented vulnerabilities.[1] These are also the vulnerable applications most frequently used in the vulnerability testing literature [6,7,18]. The traceroute and restore vulnerabilities can be exploited locally whereas ftpd and samba vulnerabilities can be exploited remotely. The versions of the operating system and the vulnerable application are provided along with references to the exploits. All vulnerable applications and the corresponding exploits are deployed with default settings. As established by the previous work, vulnerable applications are executed under different scenarios to establish the normal behavior for each application [6,7,18,20].

The attacks against the four Linux applications are buffer overflow attacks (Section 3.1), where the attackers take advantage of the vulnerabilities to inject their malicious code. In a typical buffer overflow attack, the attacker injects more data than the vulnerable variable can hold, hence causing the excess data to spill into the unallocated memory space or into other allocated variables. The main goal of a buffer overflow attack is to overwrite the system state information stored in the memory and consequently divert the execution to attacker's code. Although the experiments detailed in this paper employ buffer overflow attacks, the proposed arms race is applicable to a wider scope of attacks – as discussed in Section 3.3 – where the objective of the attacker is to design an attack, which can perform the objectives of a core attack while remaining undetected.

We note that, while the vulnerable applications employed in this work are for Linux, our approach is applicable to buffer vulnerabilities that exist on other operating systems. The exploits that the arms race generates are OS independent as long as the vulnerability – which can be considered as a method for injecting the evolved exploit – exists on multiple platforms. For example, an attacker can take advantage of the LibTIFF buffer underflow vulnerability [29] on numerous operating systems such as Solaris, Mac OS X, iPhone OS as well as Linux by crafting a malicious TIFF image (containing a shellcode) as long as the operating system makes use of the vulnerable TIFF library.

#### 4.1.1. Traceroute configuration

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 6.2 is shipped with traceroute version 1.4a5, where this is susceptible to a local buffer overflow exploit that provides a local user with super-user access [14]. The attack takes advantage of vulnerability in malloc chunk, and then uses a debugger to determine the correct return address to take control of the program. In order to establish traceroute behavior under normal conditions traceroute is executed by supplying different targets, as established by the previous work [13]. As per the discussion of EEG (Section 3.2) the most frequent 20 system calls employed by the application (Fig. 7) are used to define the instruction set.

#### 4.1.2. Restore configuration

Restore is a component of Linux backup functionality, which restores the file system image taken by the *dump* command.

[1] See Security Focus Vulnerability archives http://www.securityfocus.com.

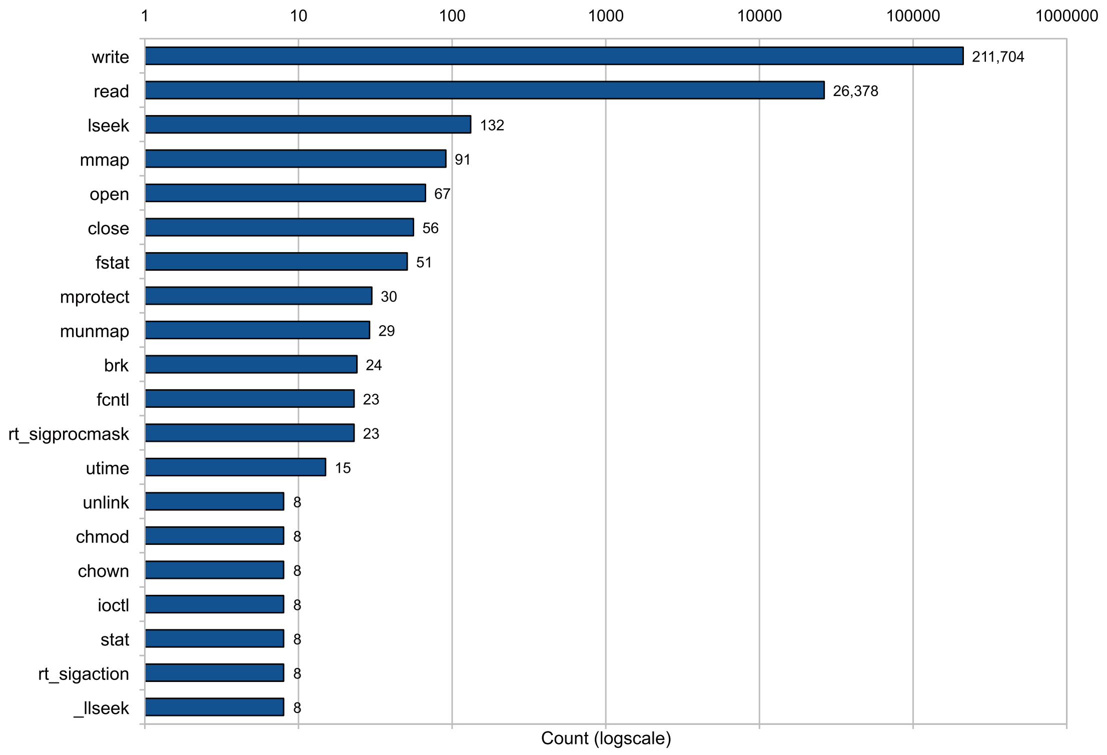


**Fig. 8.** Most frequent 20 system calls for restore.

Files or directories can be restored from full or incremental backups. Restore version 0.4b15 on Red Hat 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 [16], attacker spawns a root shell. In order to establish normal behavior for restore, restore is executed numerous times to process backup files with different sizes [13]. The most frequent 20 system calls for restore application is shown in Fig. 8. The instruction set for the restore attacks contain the system calls in Fig. 8.

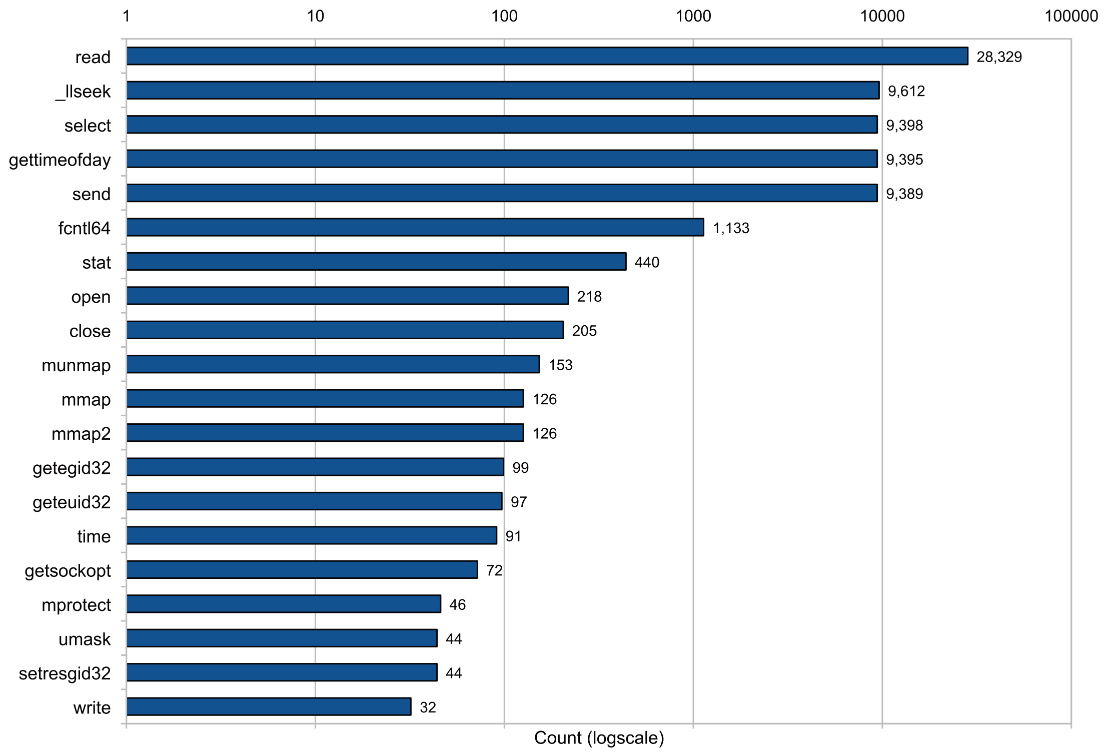


**Fig. 9.** Most frequent 20 system calls for samba.

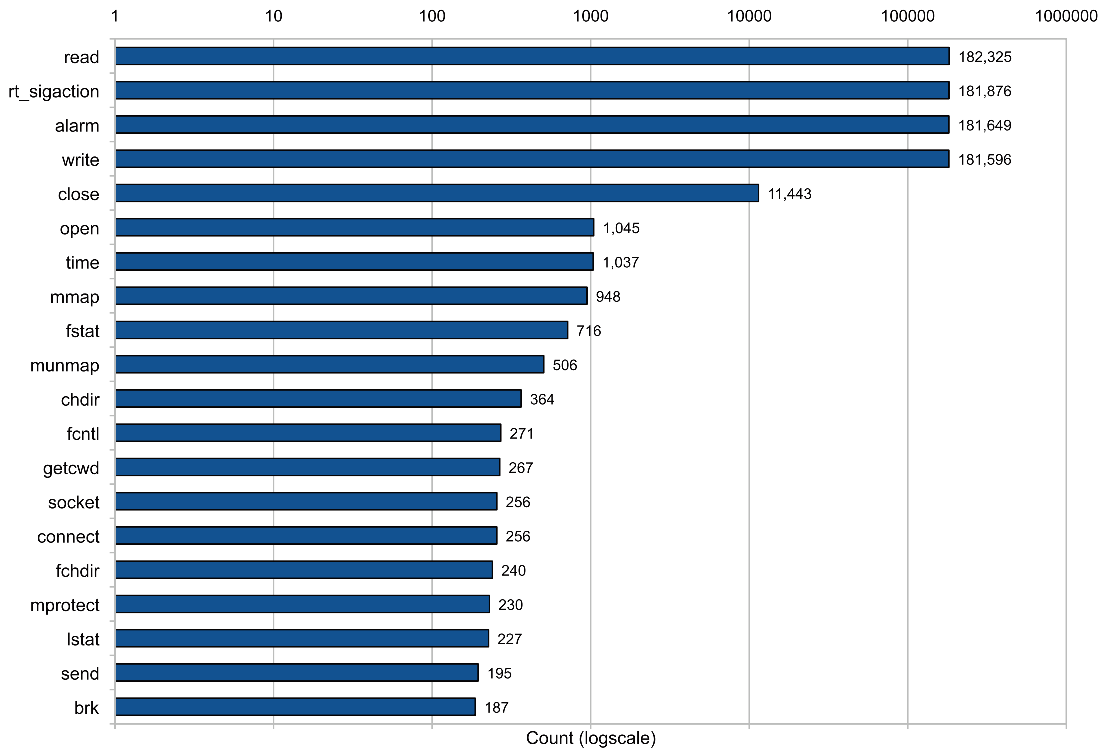


**Fig. 10.** Most frequent 20 system calls for ftpd.

### 4.1.3. *Samba configuration*

The Samba suite provides printer and file sharing for Windows clients and can run on most Linux variants. Samba sets up printer and network shares that appear as disks and printers under a Windows operating system. Red Hat 9.0 is shipped with Samba suite version 2.2.7a, which has a vulnerability [15] that can be exploited over the network to gain super-user privileges. The buffer overflow occurs when a 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. To establish normal behavior, samba is deployed on a host and activity was generated involving the mounting and unmounting of a samba share and various file operations on the samba share including file edit and copy operations [13]. The most frequent 20 system calls that samba application makes is detailed in Fig. 9.

### 4.1.4. *Ftpd configuration*

Red Hat 6.2 is shipped with Washington University Ftp Server version 2.6.0(1), which provides FTP access to remote users. WuFtpD 2.6.0(1) is susceptible to an input validation attack where the attacker can corrupt the process memory by sending malformed commands and overwrite the return address to execute a shellcode. Although the attack is an input validation attack [12], the deployment is similar to a buffer overflow attack. In order to establish a normal behavior, numerous FTP sessions are formulated where each session involves a login followed by a series of file upload and download operations [13]. The most frequent 20 system calls that ftpd application makes is detailed in Fig. 10.

## 4.2. *Anomaly detectors*

The anomaly detectors employed in this work 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.

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 application system calls, with sliding window, Markov and frequency based pattern recognition detectors all being considered. Second, depending on the complexity of the detector, they provide feedback in the form of anomaly rates and delays, which can be utilized to guide the search for the evasion attacks. The detectors discussed in this section are employed with their default parameters.

### 4.2.1. *Stide*

Forrest et al. [8] employed a methodology motivated by immune systems. This characterizes the problem as distinguishing 'self' from 'non-self' (normal and abnormal behaviors, 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 behavior model. During the deployment (detection) phase, if the pattern from the sliding window is not in the normal behavior database, it is considered a mismatch. Stide is employed with the parameterization assumed in earlier studies (Section 2) and are listed in Table 2.

### 4.2.2. *Process Homeostasis (pH)*

Process Homeostasis (pH) [30] represents a second generation anomaly detector and is therefore designed to address specific

**Table 2**
Stide configuration parameters.

| Parameter | Setting |
|---|---|
| Sliding window length | 6 |

**Table 3**
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 |

drawbacks of 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[2] 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 [30] 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 behavior. Sensitization prevents abnormal behavior from leaking into the normal behavior database [30].

In addition, pH responds to attacks by slowing down the offending process. The delay is an exponential function of the locality frame (*LF*) count where the locality frame count aims to identify the clusters of anomalies. To this end, pH simply maintains a count of how many of the past system calls within the locality frame were anomalous. Consequently, even though the attack might minimize anomaly rate, it can still be detected if the remaining anomalies are clustered together. pH was employed with the training parameters, which are listed in Table 3.

### 4.2.3. *Process Homeostasis with a schema mask (pHsm)*

Inoue and Somayaji [31] discussed the differences between look-ahead pairs and sequences. In their paper, the authors also proposed an improvement to pH based upon the concept of a random schema mask. Their main motivation was the observation that longer windows improve detection rates hence there exists a potential to increase the difficulty of generating evasion attacks against pH (and indirectly Stide and variants). We call the extended pH as pH with a schema mask (pHsm) in this work. In pHsm, a longer sliding window is maintained and a number of taps 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 configuration parameters for pHsm are detailed in Table 4.

### 4.2.4. *The Markov Model-based detector*

The Markov Model is a statistical modeling technique, which is useful for building probabilistic models of event sequences evolving in time. Markov Models have been utilized within the context of intrusion detection systems [32,33] as in the related case of a Finite State Automata representation [34]. The Markov Model was selected as an anomaly detector in this work because: (1) it can build probabilistic models using exemplars from only one class (i.e. normal behavior) and (2) it can capture temporal (i.e. sequence) information without employing a sliding window.

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

**Table 4**
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 |

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 work employs a first order Markov Model. In a first order Markov Model, the next state is only dependent upon the current state, where such an assumption is widely employed in these systems to reduce the number of 'free parameters', which require estimation. In order to establish values for the above model parameters, the Baum–Welch model is assumed [35]. The detection decision is based upon a characterization of state transition behavior, which was employed in the previous Markov Model detector approaches [32] and in Stide [36]. 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, the probability transition is zero), a mismatch flag is set. A count of the mismatch flag is maintained, and the anomaly rate is defined by the number of mismatch flags divided by the total number of transitions encountered. Such an anomaly rate implies that, if the test sequence follows the training model (i.e. normal behavior), 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 5.

### 4.2.5. *Auto-associative neural network*

The auto-associative neural network is a multi-layer perceptron configured in a 'bottleneck' topology – i.e., the hidden layer is configured with a reduced neuron count relative to the input and output spaces (which are presented with the same exemplar during training). Such a bottleneck forces the network to identify the most appropriate encoding to correctly reconstruct the input at the output post training [37–40]. 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 that the auto-associative neural network encapsulates. In our experiments, an auto-associative neural network was employed as an anomaly detector. As opposed to the other detectors discussed in 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. [41], which uses a bag of words representation as the detector input. As such, the resulting frequency distribution constitutes the normal behavior characteristics. Given the frequency distribution vector for the test trace, the detection is therefore based upon the

**Table 5**
Markov Model parameters.

| Parameter | Setting |
|---|---|
| Order | First order |
| Number of states | 223 (number of system calls) |
| Training algorithm | Baum–Welch |

divergence between the frequency distribution vectors of the test trace and the 'normal behavior'. Since the frequency distribution of a trace is calculated after the trace is complete, the auto-associative neural network can be considered as an 'off-line' detector, which provides post-mortem analysis of the system call traces after they are executed.

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. 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. 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 input and the output that the neural network produces are compared using Euclidean distance. The calculated distance varies between 0 and 100, where larger numbers indicate anomalous behavior. The training parameters for the auto-associative Neural Network are detailed in Table 6. The credit assignment takes the form of the very efficient second order conjugate gradient optimization algorithm, making training much more efficient and more accurate than the regular back propagation algorithm [42].

**Table 6**
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 | $10^{-6}$ |

### 4.3. Executing a system call sequence in practice

The anomaly detectors, which are introduced in Section 4.2 monitor system call sequences therefore the arms race generates exploits in the form of system call sequences. When the attacks are deployed in practice, it is necessary to create the appropriate shellcode for a given system call sequence. Fig. 11 provides the shellcode for system call sequence `open write close`, which was automatically generated by the converter software that we developed. The converter software contains a library of system call templates. Each template is an assembly code segment which ensures that the registers are set properly to represent the system call parameters before calling the `int 0x80` instruction. The converter software takes a

```
jmp short variables
syscalls:
   pop esi
   xor eax, eax
   mov [esi + 11], al  ; put \0 in file name (replaces #)
   mov [esi + 47], al  ; put \0 in malicious text (replaces #)
   mov byte [esi + 46], 0xa  ; put \n in malicious text (replaces *)
   lea eax, [esi + 16]  ; load a pointer to malicious text in eax
   mov long [esi + 48], eax  ; move the pointer value, replacing values XXXX
   ;;;;;;;;;;;;; open ;;;;;;;;;;;;;
   xor eax, eax    ; eax is set to zero
   mov al, 5       ; system call number for open
   lea ebx, [esi]  ; ebx contains a pointer to the file name
   mov cx, 1090    ; file permissions, 1090 denotes append, create if not exists
   mov dx, 744q    ; if newly created, these are the file permissions
   int 0x80        ; execute the open system call
   mov [esi + 12] , eax  ; save file descriptor, replacing values YYYY
   ;;;;;;;;;;;;; write ;;;;;;;;;;;;;
   xor eax, eax     ; eax is set to zero
   mov al, 4        ; system call number for write
   mov long ebx, [esi + 12]  ; move file descriptor pointer to ebx
   mov ecx, [esi + 63]  ; move malicious text pointer to ecx
   xor edx, edx  ; edx is set to zero
   mov dx, 9999  ; move length of the malicious text (i.e. 31)
   sub dx, 9968  ; into edx without creating null characters in shellcode
   int 0x80      ; execute the write system call
   ;;;;;;;;;;;;; close ;;;;;;;;;;;;;
   xor eax, eax  ; eax is set to zero
   mov al, 6     ; system call number for close
   mov long ebx, [esi + 12]  ; move file descriptor pointer to ebx
   int 0x80  ; execute the close system call
   ;;;;;;;;;;;;; exit ;;;;;;;;;;;;;
   xor eax, eax  ; eax is set to zero
   mov al, 0x01  ; system call number for exit
   xor ebx, ebx  ; ebx is set to zero
   int 0x80  ; execute the close system call
variables:
   call syscalls
   ;   0123456789012345      byte index provided for readability
   db '/etc/passwd#YYYY'                      ; file name
   ;   6789012345678901234567890123456789 01      byte index
   db 'toor::0:0:root:/root:/bin/bash*#XXXX'   ; malicious text
```

**Fig. 11.** Assembly code of the system call sequence open() write() close(), which opens the password file and adds a super-user account.

**Table 7**
Anomaly rates of the preamble and exploit components of the traceroute attacks (evolved attacks compared to the original).

| | | Preamble | Exploit | Attack |
|---|---|---|---|---|
| Stide | Original | 6.98% | 71.48% | 61.26% |
| | Evolved | | 16.67% | 10.96% |
| pH | Original | 36.49% | 73.91% | 66.27% |
| | Evolved | | 11.71% | 18.29% |
| pHsm | Original | 77.78% | 83.06% | 81.79% |
| | Evolved | | 27.60% | 29.28% |
| MM | Original | 8.54% | 47.89% | 38.78% |
| | Evolved | | 0.10% | 0.20% |
| NN | Original | 22.04% | 70.21% | 31.19% |
| | Evolved | | 2.47% | 1.63% |

**Table 8**
Anomaly rates of the preamble and exploit components of the restore attacks (evolved attacks compared to the original).

| | | Preamble | Exploit | Attack |
|---|---|---|---|---|
| Stide | Original | 77.82% | 88.13% | 84.69% |
| | Evolved | | 0.40% | 46.25% |
| pH | Original | 81.01% | 90.70% | 87.49% |
| | Evolved | | 0.10% | 48.57% |
| pHsm | Original | 93.67% | 98.30% | 96.77% |
| | Evolved | | 0.31% | 57.92% |
| MM | Original | 35.08% | 48.84% | 44.26% |
| | Evolved | | 0.10% | 21.05% |
| NN | Original | 13.29% | 15.53% | 14.00% |
| | Evolved | | 2.90% | 5.60% |

system call sequence and automatically creates the shellcode. The resulting shellcode can be embedded into the attacks [14,16,15,12] after being encoded in a C-type string format using *s-proc*.[3]

There are two main differences between an assembly program and a shellcode. First, shellcode cannot contain any null characters since memory copy functions such as *strcpy* will only copy until the null byte is reached thus failing to copy the entire shellcode. However, most system calls require arguments in the form of C-style strings, which should be terminated by a null byte. Thus, the shellcode needs to prepare the null terminated strings without causing a null byte in the shellcode encoding. A well-known method for obtaining a null byte is to apply the `XOR` instruction on a register and moving its contents to where the null byte is needed. For example, step 4 in Fig. 11 sets the EAX register to zero since the result of the `XOR` instruction, which is stored back in EAX register, is always zero regardless of the value of the EAX. Consequently, step 5 moves the contents of the 1-byte AL register (which is a null byte) to the memory location (denoted by # in line 40), where the string needs to be terminated.

Second, all parameters and variables required to execute the system calls properly should be self-contained in the shellcode. Since the address of the vulnerable variable changes at runtime, a shellcode needs to a way to address the variables irrespective of the actual memory address. To this end, jmp/call method [43] can be utilized, in which the shellcode jumps to a call instruction first (step 1 in Fig. 11). The call instruction (step 38, Fig. 11) directs the execution to the actual shellcode. However, the use of the call instruction results in the address of the parameters to be stored in ESI. Thus, in Fig. 11, the parameter `'/etc/passwd'` can be accessed at [ESI] and the parameter `'toor::0:0:root:/root:/bin/bash'` can be accessed at [ESI +16].

The assembly code in Fig. 11 starts by jumping to the call instruction (step 38), which directs the execution back to step 3 after the address of the parameters are stored in ESI register. Step 4 sets the EAX register to zero, creating a null byte. Using the null byte, steps 5 and 6 copy the null byte to the end of the strings, locations of which are denoted by `#` (steps 40 and 42). A next line character is inserted (step 7), the location of which is shown by `*`, step 42. Using the `LEA` instruction, the address of the malicious text is obtained and stored in its allocated 4-byte space denoted by `XXXX` (step 42).

Each system call segment in Fig. 11, sets the register values to system call parameters as defined in Linux manual pages.[4] For example, to execute an `open` system call, the system call number is stored in EAX, the pointer to the file name is stored in EBX, the flags and permissions are stored in ECX and EDX, respectively. After the system call is executed (step 17) `open` system call returns a file descriptor, which is then stored in its allocated 4-byte space denoted by `YYYY` (step 40).

## 5. Results

The following empirical evaluation is developed through three themes: Anomaly rate minimization; Significance of detector counter measures; and Exploit analysis. Specifically, earlier approaches to detector vulnerability testing concentrated on the Stide detector alone and focused on minimizing the exploit anomaly rate as opposed to the attack anomaly rate.

### 5.1. Anomaly rate minimization

In order to determine the anomaly rates of the original attacks, original traceroute, restore, samba, ftpd attacks [14,16,15,12] are downloaded from the SecurityFocus website and deployed against the detector configurations detailed in Section 4.2 on the four vulnerable applications detailed in Section 4.1. Needless to say, depending on the detection methodologies and configuration parameters, the original attacks produce different anomaly rates for different anomaly detectors [44].

In the proposed arms race, multiple solutions are maintained; hence multiple attacks are generated. For the comparisons, the attacks that produced the least attack anomaly rates are selected. It is important to emphasize that evasion attacks are generated against each detector separately, hence there are five evasion attacks, one against each detector. Tables 7–10 provide a comparison of anomaly rates between the original attacks and the evolved attacks, for traceroute, restore, samba, and ftpd applications, respectively.

Anomaly rates of the preambles in Tables 7–10 show that an attacker is generally not able to take control of an application 'silently' in every case. For example, the traceroute preamble produces between 6.98% and 77.78% anomaly rates depending on the anomaly detector. Even though the attacker manages to generate exploits with very low anomaly rates, the attack (i.e. pream-

**Table 9**
Anomaly rates of the preamble and exploit components of the samba attacks (evolved attacks compared to the original).

| | | Preamble | Exploit | Attack |
|---|---|---|---|---|
| Stide | Original | 3.57% | 60.04% | 10.16% |
| | Evolved | | 0.50% | 3.00% |
| pH | Original | 9.97% | 60.51% | 16.02% |
| | Evolved | | 0.10% | 8.11% |
| pHsm | Original | 12.07% | 99.60% | 99.95% |
| | Evolved | | 29.23% | 15.84% |
| MM | Original | 6.78% | 25.53% | 9.03% |
| | Evolved | | 0.10% | 5.45% |
| NN | Original | 6.34% | 21.15% | 5.73% |
| | Evolved | | 16.68% | 5.77% |

[3] Erickson [43] provides a detailed discussion of shellcode encoding and the *s-proc* application.

[4] Section 2 of the Linux manual pages, which can be accessed by typing `man 2 syscalls` in a Linux terminal, provides the system call definitions.

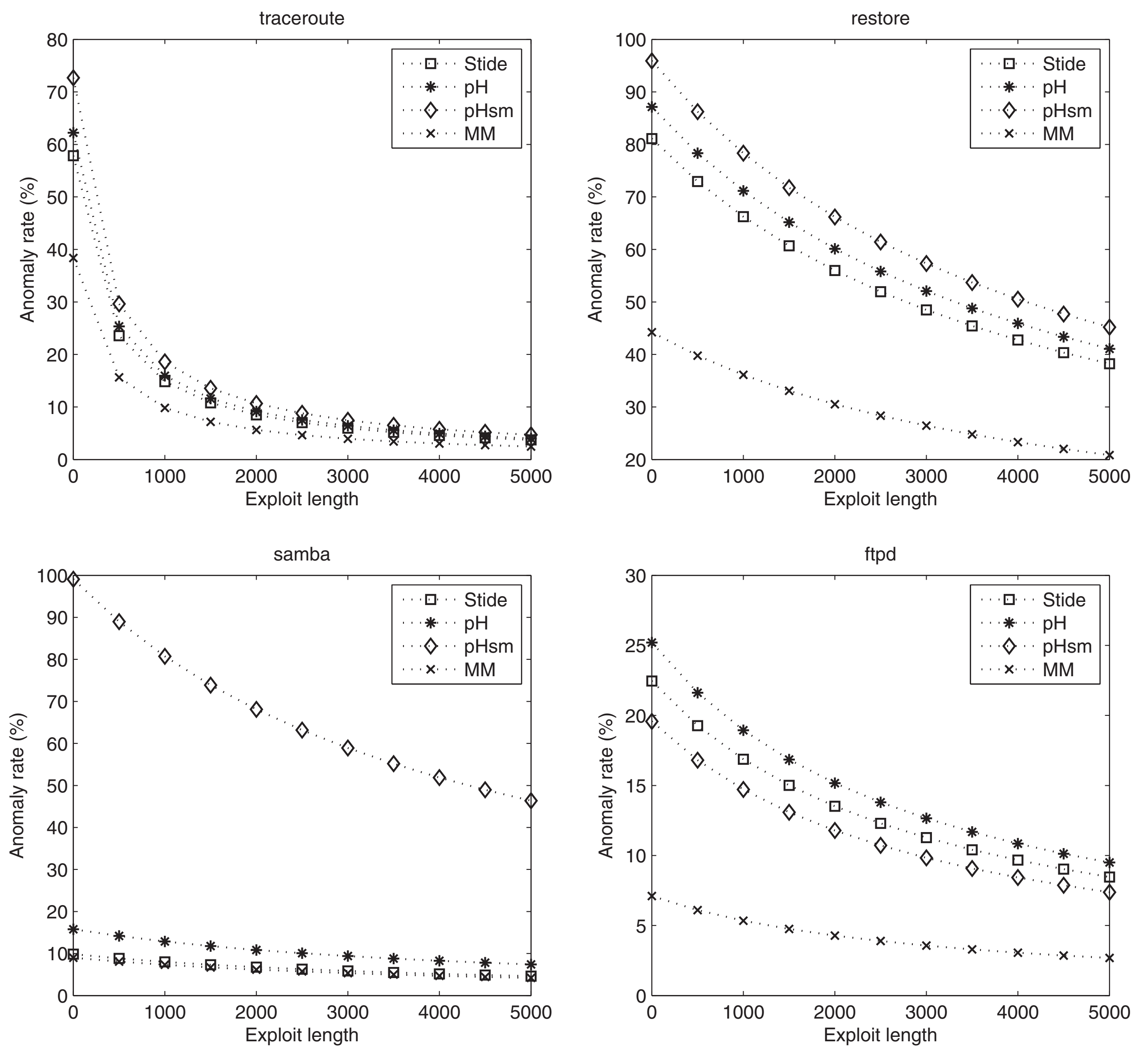


**Fig. 12.** Anomaly rate of the attacks with different exploit lengths for traceroute, restore, samba and ftpd applications.

ble + exploit) will still contain anomalies. Indeed distinct trends emerge with respect to evolved versus original exploits. Thus under the original exploit, the anomaly rate from the exploit alone is generally higher than that of the contribution of the preamble alone. Moreover, under the original configuration, the resulting attack anomaly rate is generally lower than the exploit alone. Conversely under EEG, anomaly rates of the exploit are generally lower than that of the preamble, whereas the overall attack anomaly rate is generally higher than that of the exploit alone. Needless to say, only very rarely does the attack anomaly rate of the original attack approach is as low as that of the evolved attacks.

While a stealthy exploit with 0% anomaly rate can reduce the overall attack anomaly rate, the anomaly rate of the preamble along with its length play an important role in determining the anomaly rate of the resulting attack. Fig. 12 shows the anomaly rate of the attacks, when a stealthy exploit with varying lengths are utilized for traceroute, restore, samba and ftpd. In case of traceroute, employing a long and stealthy exploit reduces the anomaly rate of the attack from approximately 70% to 5%. On the other hand, in case of restore and samba, where the preamble is long, the benefit of employing a longer exploit is considerably less. For example, a restore attack against Stide can reduce the attack anomaly rate from approximately 80% to 40%. However, a samba attack against Stide only reduces the anomaly rate from approximately 20% to 10%. Thus, the anomaly rate of the overall attack (i.e. preamble + exploit) depends on the length and the anomaly rate of the preamble.

**Table 10**
Anomaly rates of the preamble and exploit components of the ftpd attacks (evolved attacks compared to the original).

| | | Preamble | Exploit | Attack |
|---|---|---|---|---|
| Stide | Original | 19.04% | 47.52% | 22.78% |
| | Evolved | | 57.14% | 19.30% |
| pH | Original | 21.94% | 47.85% | 25.54% |
| | Evolved | | 0.10% | 16.11% |
| pHsm | Original | 14.30% | 57.29% | 20.27% |
| | Evolved | | 35.55% | 20.19% |
| MM | Original | 6.11% | 13.65% | 7.15% |
| | Evolved | | 0.10% | 4.47% |
| NN | Original | 6.88% | 18.86% | 6.91% |
| | Evolved | | 3.46% | 1.26% |

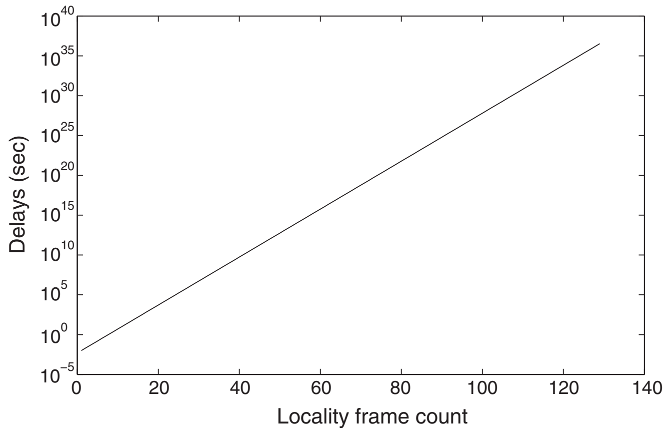


**Fig. 13.** Locality frame counts and the associated delays (*y*-axis is in logarithmic scale).

## 5.2. *Effectiveness of counter measures in pH and pHsm*

The impact of preambles to the attack gains importance especially when the anomaly detectors employ locality frame count to delay the attacks. Although Stide can keep track of the locality frame counts, pH and pHsm employ it to enforce delays on the attacks. The locality frame count keeps track of the mismatches over a given time period (by default, the previous 128 system calls). If the anomalies are clustered together, this condition will produce high locality frame counts and pH and pHsm treat this condition as attacks. On the other hand, if the anomalies are scattered, it will produce lower locality frame counts [30]. Based upon the observed locality frame counts, pH and pHsm respond to attacks by slowing down the process. The delay that pH and pHsm associate with a system call can be expressed as:

$$delay\_factor \times 0.01 \times 2^{LFC} \quad (1)$$

Higher *delay _factor* values produce longer process delays and the *LFC* signifies how many of the past 128 system calls were anomalous. Utilizing the Eq. (1), Fig. 13 shows the relationship between the locality frame count values and the resulting delays in seconds. The delays shown in Fig. 13 depends on the *LFC* values hence are applicable to both pH and pHsm. If the observed locality frame count increases to 120, the resulting delay will be close to $10^{35}$ s, which can be expressed in centuries. Sustaining high LFC values throughout the attack will enforce high delays multiple times hence increasing the overall delay above $10^{35}$ s.

Table 11 provides a comparison of delays between the original attacks and the evolved attacks, for traceroute, restore, samba, ftpd applications, respectively. For example, even though the ftpd attack evolved against pHsm achieves low delays, the anomalous preamble causes a $5.26 \times 10^{30}$ second delay, which can be expressed in centuries. This issue has not been thoroughly investigated in previous work because the preamble is assumed to be ‘silent’, it follows that reducing or eliminating the anomalies of the exploit suffices in evading anomaly detectors. Clearly, in case of anomaly detectors this is not the case.

The analysis of the delays reveals that, 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.

Here we emphasize that our observations are independent of the evasion methodology. That is to say, even though one can utilize various other evasion attack techniques [7,6,10,11] to reduce the anomaly rate of the exploits (or reduce it to 0%), the anomalies from the preamble will suffice in the detection of the attack. Our analysis indicates that evading sequence based anomaly detectors may not be as easy as suggested by the previous work. However, an anomalous preamble followed by an evasion exploit will exhibit itself as an anomalous behavior followed by an abrupt switch to normal behavior. Therefore, even though the exploits can be improved to evade detection, such an abrupt transition provides an opportunity for detecting evasion attacks. Moreover, application–attack combinations with short preambles – in this case traceroute – provide most opportunity for the attacker to evade the counter measure. In effect, the larger the contribution of the preamble, the less freedom there is to bias the behavioral properties of the attack towards a normal behavioral profile.

Finally, one could argue that the preamble can also be improved. However, it should be noted that the attacker does not have full control of the application during the preamble, therefore an evasion methodology for improving exploits cannot be applied since the evasion methodologies generally boil down to finding system call sequences with 0% anomaly rates. One should keep in mind that the system calls executed during the preamble stage are the result

**Table 11**
Delays of the preamble and exploit components of attacks in seconds (evolved attacks compared to the original).

| | | Preamble | Exploit | Attack |
|---|---|---|---|---|
| Traceroute application | | | | |
| pH | Original | 0.74 | $4.39 \times 10^{35}$ | $4.39 \times 10^{35}$ |
| | Evolved | | 1.11 | 1.91 |
| pHsm | Original | 0.63 | $8.51 \times 10^{35}$ | $8.51 \times 10^{35}$ |
| | Evolved | | $1.50 \times 10^{14}$ | $1.50 \times 10^{14}$ |
| Restore application | | | | |
| pH | Original | $1.90 \times 10^{38}$ | $1.66 \times 10^{39}$ | $1.85 \times 10^{39}$ |
| | Evolved | | 9.94 | $1.90 \times 10^{38}$ |
| pHsm | Original | $1.01 \times 10^{39}$ | $3.93 \times 10^{39}$ | $4.96 \times 10^{39}$ |
| | Evolved | | $1.11 \times 10^{1}$ | $1.01 \times 10^{39}$ |
| Samba application | | | | |
| pH | Original | $7.95 \times 10^{27}$ | $2.97 \times 10^{30}$ | $3.11 \times 10^{30}$ |
| | Evolved | | 9.94 | $7.95 \times 10^{27}$ |
| pHsm | Original | $1.27 \times 10^{40}$ | $8.96 \times 10^{38}$ | $1.41 \times 10^{40}$ |
| | Evolved | | $7.37 \times 10^{12}$ | $1.27 \times 10^{40}$ |
| Ftpd application | | | | |
| pH | Original | $5.26 \times 10^{30}$ | $3.78 \times 10^{22}$ | $5.26 \times 10^{30}$ |
| | Evolved | | 9.94 | $5.26 \times 10^{30}$ |
| pHsm | Original | $8.03 \times 10^{17}$ | $4.89 \times 10^{25}$ | $4.89 \times 10^{25}$ |
| | Evolved | | $9.86 \times 10^{17}$ | $1.79 \times 10^{18}$ |

**Table 12**
Best exploit lengths 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 |

of the interaction between the attacker and the vulnerable application. Therefore, an evasion method for improving the preambles should model the communication between the attacker and the vulnerable application.

## 5.3. Behavioral analysis of exploits

Having established the effectiveness of the arms race in Section 5, we focus on how the generated exploits evade the detectors. The analysis detailed in this section focuses on the best exploit per detector, application pair to investigate the affects of the application characteristics in the resulting exploits. Specifically, the best exploits from each population (one for each application) were selected, where the 'best' was defined as the exploit with the lowest attack (i.e. preamble + exploit) anomaly rate. The 'best' performing exploits are discussed in terms of the system calls, which they employ to camouflage the malicious intent. Furthermore, the lengths of the best exploits are detailed in Table 12. Best exploits and a discussion of Linux system calls are provided in the Appendix B and C of Kayacık's thesis [44].

### 5.3.1. Exploits evolved against Stide

The best exploit (i.e. that with the lowest attack anomaly rate) against traceroute repeats 6 system calls in the exploit, namely, `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. Needless to say, all of these system calls appear in the top 5 most frequent system calls of the application (Fig. 7). Thus, EEG has successfully discovered that the best system calls for obfuscation are those which are most frequently employed by the application.

The best exploit against samba 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. The exploit 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, Fig. 9, whereas the `open write close` sequence are necessary to define the exploit itself.

Similarly, the best exploit against restore employs a pattern, in which a number of `write` system calls precede a `read` system call (again corresponding to the most frequent application system calls). The exploit deploys this smokescreen pattern and follows up with the attack system calls as the last three system calls.

The best exploit against ftpd is concise with 11 system calls, Table 12, compared with the other exploits generated against Stide. In this exploit, no repeating pattern is employed. However the exploit employs system calls, which involve file checks such as `getcwd`, `fchdir` and `fstat` along with the system calls, which involve timing, such as `alarm`.

The common characteristic of the evolved exploits against Stide [44] is that they identify and employ a set of system calls, which were used frequently by the application during its normal operation. In the case of traceroute, the exploits repeat a certain pattern, whereas in samba and restore, the repetition of system calls is not as clear. On the other hand, the ftpd exploit choose to utilize fewer system calls with no repeating patterns, which seems to indicate that different exploit techniques exist against different applications.

### 5.3.2. Exploits evolved against pH

As opposed to the traceroute exploit against Stide, the best traceroute exploit 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 exploit against pH for samba is fairly long, Table 12. 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 exploit against Stide, in which the exploit focuses on utilizing `read` and `_llseek` system calls. In terms of the system calls, which achieve the attack objectives, the `open` and `write` system calls are toward the beginning of the exploit whereas the `close` system call is toward the end.

The best exploit against pH for restore alternates between the use of 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, as shown in Fig. 8. Similarly, the restore exploit against Stide employed `write` and `read` system calls primarily, which are the two most frequent system calls.

Compared to the ftpd exploit against Stide, the ftpd exploit against pH 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 EEG employed against Stide, the exploits against pH contain system calls, which the applications execute frequently during their normal operation [44], 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 intent of the exploit.

### 5.3.3. Exploits evolved against pHsm

In the best pHsm exploit on traceroute, the true intent of the exploit 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 exploit against pH but different from the Stide exploit, which utilizes timing system calls.

The best exploit on samba does not utilize a clear repeating pattern but uses a combination of `read`, `stat`, `munmap`, `mmap` and other numerous system calls. The system calls, which achieve the attack objectives are 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 exploit on restore learns to hide within normal behavior by utilizing a series of `write` and `read` system calls, which make up the 99% of the executed system calls by restore, Fig. 8. Restore exploits against both Stide and pH utilize the same technique to hide the true intent of the exploit.

As opposed to the concise ftpd exploits for Stide and pH, the ftpd exploit against pHsm 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` to formulate the padding, within which the malicious system calls are hidden.

While employing numerous system calls, which do not appear on exploits against Stide and pH, the exploits against pHsm follow a similar technique for hiding the true intent of the exploit. That is to say, although no repeating pattern clearly exists, the exploits against pHsm utilize the system calls, which are used frequently during the normal operation of the applications. This means that, in cases where the frequency of the system call distribution is even, such as in the case of traceroute, the exploits utilize 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, the exploits utilize a smaller set of system calls to construct the exploits.

### 5.3.4. *Exploits evolved against the Markov Model*

The best exploit on traceroute utilizes the system calls, which perform I/O and memory access operations such as `gettimeofday`, `select`, `write` and `munmap` to hide the true intent of the exploit. As in the traceroute exploit against Stide, EEG utilizes system calls related to timing.

The best exploit on the samba application against the Markov Model detector 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 exploits against Stide, pH and pHsm, the true intent of the exploit is hidden within numerous types of system calls related to I/O operations.

The best restore exploit against the Markov Model detector 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 operations, 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 result in a greater number of system call sequences to be stored in the normal database due to the fact that longer sliding window patterns ‘detect’ more variations. 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 exploit against the Markov Model detector utilizes a set of system calls performing I/O (i.e. `open`, `read`, `write`, `close`) to hide the true intent of the exploit. Given an application, which makes frequent `open`, `write` and `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 paper) 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 small relative to the above-mentioned detectors, EEG 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.

### 5.3.5. *Exploits evolved against the Neural Network*

The traceroute exploit 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 support for multiplexing several data streams. The use of different system calls indicates that the exploit employs different strategies to evade different detection methodologies.

As in the traceroute case, the samba exploit contains system calls such as `getegid32` and `getsockopt`, which are not utilized in the other samba exploits against other detectors. The best restore exploit against the Neural Network detector 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 the exploit employed a different strategy while evading the system call frequency distribution model of the Neural Network detector.

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 in case of the ftpd exploit against the Neural Network detector, 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, Fig. 10.

Since the detection methodology, which the Neural Network utilizes looks for a match in the frequency distribution of system calls in a trace, the exploits utilize rarely used system calls as well as the frequently used system calls in order to match the frequency distribution, for which the detector seeks.

### 5.3.6. *Discussion of the exploit analysis*

The analysis of the exploits indicates that the application characteristics play an important role in determining the strategies that EEG uses to evade detection. If the frequency distribution of system calls in normal behavior is fairly even (i.e. distributed among a number of system calls), as with traceroute, EEG 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, as with restore, then EEG identifies the frequent subset and employ the system calls accordingly. The analysis of the best exploits indicates that EEG uses system calls related to file I/O, memory allocation/deallocation and timing to hide the true intent. In the cases of samba and ftpd, EEG 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 network-based applications among the four employed in this paper.

It is evident that, when exploits against different detectors are compared (on the same application), the evasion strategy is identified 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 EEG 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 (mainly, the frequent system calls).

## 6. Conclusion

This paper proposed an artificial arms race between anomaly detectors and an attacker (i.e. the EEG framework) which was implemented using Linear Genetic Programming. The main motivation for generating such evasion attacks against anomaly detectors is to identify and eliminate detector weaknesses before the attackers. The proposed arms race can be facilitated against various other detectors as long as the detector provides a suitable detection feedback (i.e. similar to anomaly rates).

In this arms race, the attacker interacts with the target detector by utilizing the detector feedback (i.e. anomaly rates and 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 can analyze the evasion attacks and eliminate the weaknesses of the target detector.

Our arms race experiments involved opposing our attacker against five anomaly detectors, which monitor four vulnerable Linux applications. Results indicate that the proposed arms race can successfully reduce 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.

In particular the results in this paper demonstrate that, in practice, evading anomaly detectors can be more difficult due to the fact that, during the break-in (i.e. the preamble phase), the attacker does not have full control over the system calls that the vulnerable application executes. It is only after the shellcode begins to execute (i.e. exploit phase), that the attacker can alter the system call sequences to evade detection.

Although the evasion methodologies (introduced in this paper and the previous work [6,7,10]) are effective in generating exploits with zero anomaly rates, the results discussed in this paper demonstrated 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. For example, the results indicate that for an ftpd exploit, which raises no alarms, the corresponding attack still produced a 10.62% anomaly rate because of the anomalies from the preamble.

Additionally, 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, evasion attacks against samba have low anomaly rates yet the delays associated with them are expressed in terms 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 behavior, which is unknown to the detector can cause substantial delays as well.

In the light of these observations, we believe that the evasion attack 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. Moreover, our future arms race efforts will employ adaptive detection methods 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.

## Acknowledgements

The authors gratefully acknowledge the support of SwissCom Innovations SA. (Switzerland), TARA Inc. (Canada), Killam, Precarn, MITACS and NSERC funding programs.

## References

[1] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, J. Ellis, E. Hayes, J. Marella, B. Willke, State of the practice of intrusion detection technologies, Tech. rep., Carnegie Mellon Software Engineering Institute (2000).
[2] D.E. Denning, An intrusion detection model, IEEE Trans. Softw. Eng. 13 (2) (1987) 222–232, doi:http://dx.doi.org/10.1109/TSE.1987.232894.
[3] R. Kemmerer, G. Vigna, Intrusion detection: a brief history and overview, Computer 35 (4) (2002) 27–30.
[4] H.G. Kayacık, A.N. Zincir-Heywood, M. Heywood, S. Burschka, Generating mimicry attacks using genetic programming: a benchmarking study, in: CICS'09: Proceedings of the IEEE Symposium on Computational Intelligence in Cyber Security, IEEE Computer Society, 2009, pp. 136–143.
[5] S. Christey, R.A. Martin, Vulnerability type distributions in cve, MITRE Website http://cwe.mitre.org/documents/vuln-trends/index.html (May 2007).
[6] D. Wagner, P. Soto, Mimicry attacks on host based Intrusion Detection Systems, in: ACM Conference on Computer and Communications Security, 2002, pp. 255–264.
[7] K. Tan, K. Killourhy, R. Maxion, Undermining an anomaly-based Intrusion Detection System using common exploits, in: Recent Advances in Intrusion Detection Systems (RAID), vol. 2516 of LNCS, 2002, pp. 54–73.
[8] S. Forrest, S.A. Hofmeyr, A. Somayaji, T.A. Longstaff, A sense of self for unix processes, in: SP'96: Proceedings of the 1996 IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 1996, p. 120.
[9] G. Vigna, W. Robertson, D. Balzarotti, Testing network-based intrusion detection signatures using mutant exploits, in: CCS'04: Proceedings of the 11th ACM Conference on Computer and Communications Security, ACM, New York, NY, USA, 2004, pp. 21–30, doi:http://doi.acm.org/10.1145/1030083.1030088.
[10] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, G. Vigna, Automating mimicry attacks using static binary analysis, in: USNIX Security Symposium, 2005, pp. 717–738.
[11] J.T. Giffin, S. Jha, B.P. Miller, Automated discovery of mimicry attacks, in: Recent Advances in Intrusion Detection, 9th International Symposium, RAID 2006, vol. 4219 of Lecture Notes in Computer Science, Springer, 2006, pp. 41–60.
[12] Securityfocus vulnerability archives – wu-ftpd remote format string stack overwrite vulnerability, http://www.securityfocus.com/bid/1387 (Last accessed July 2010).
[13] H.G. Kayacık, A.N. 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, IEEE Computer Society, Washington, DC, USA, 2008, pp. 213–223, doi:http://dx.doi.org/10.1109/PST.2008.25.
[14] Securityfocus vulnerability archives – lbnl traceroute heap corruption vulnerability, http://www.securityfocus.com/bid/1739 (Last accessed July 2010).
[15] Securityfocus vulnerability archives – samba 'call_trans2open' remote buffer overflow vulnerability, http://www.securityfocus.com/bid/7294 (Last accessed July 2010).
[16] Securityfocus vulnerability archives – redhat linux restore insecure environment variables vulnerability, http://www.securityfocus.com/bid/1914 (Last accessed July 2010).
[17] D. Wagner, P. Soto, Mimicry attacks on host-based intrusion detection systems, in: CCS'02: Proceedings of the 9th ACM Conference on Computer and Communications Security, ACM, New York, NY, USA, 2002, pp. 255–264, doi:http://doi.acm.org/10.1145/586110.586145.
[18] K.M.C. Tan, J. McHugh, K.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, Springer-Verlag, London, UK, 2003, pp. 1–17.
[19] W. Banzhaf, F.D. Francone, R.E. Keller, P. Nordin, Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and its Applications, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.
[20] H.G. Kayacık, M. Heywood, N. Zincir-Heywood, Evolving buffer overflow attacks with detector feedback, in: Proceedings of the EvoWorkshops (EvoCOMNET), vol. 4448 of LNCS, Springer, 2007, pp. 11–20.
[21] H.G. Kayacık, M. Heywood, N. Zincir-Heywood, On evolving buffer overflow attacks using genetic programming, in: Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO), SIGEVO, ACM, 2006, pp. 1667–1674.
[22] K.M.C. Tan, K.S. Killourhy, R.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, 2002, pp. 54–73.
[23] K. Deb, Multi-Objective Optimization using Evolutionary Algorithms, John Wiley and Sons, 2001.

[24] R. Kumar, P. Rockett, Improved sampling of the pareto-front in multiobjective genetic optimizations by steady-state evolution: a pareto converging genetic algorithm Evol. Comput. 10 (3) (2002) 283–314, doi:http://dx.doi.org/10.1162/106365602760234117.
[25] H.G. Kayacık, M. Heywood, N. Zincir-Heywood, On evolving buffer overflow attacks using genetic programming, in: GECCO'06: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, ACM, New York, NY, USA, 2006, pp. 1667–1674, doi:http://doi.acm.org/10.1145/1143997.1144271.
[26] G. Dozier, D. Brown, H. Hou, J. Hurley, Vulnerability analysis of immunity-based intrusion detection systems using genetic and evolutionary hackers, Appl. Soft Comput. 7 (2) (2007) 547–553, doi:http://dx.doi.org/10.1016/j.asoc.2006.05.001.
[27] P. LaRoche, N. Zincir-Heywood, M.I. Heywood, Evolving tcp/ip packets: a case study of port scans, in: CISDA'09: Proceedings of the Second IEEE International Conference on Computational Intelligence for Security and Defense Applications, IEEE Press, Piscataway, NJ, USA, 2009, pp. 281–288.
[28] P. LaRoche, N. Zincir-Heywood, Genetic programming based wifi data link layer attack detection, in: Communication Networks and Services Research Conference, 2006. CNSR 2006. Proceedings of the 4th Annual, 2006, p. 8, p. 292. doi:10.1109/CNSR.2006.30.
[29] Libtiff 'lzwdecodecompat()' remote buffer underflow vulnerability, http://www.securityfocus.com/bid/35451 (Last accessed July 2010).
[30] A.B. Somayaji, Operating system stability and security through process homeostasis, Ph.D. thesis, The University of New Mexico, chairperson: Stephanie Forrest (2002).
[31] H. Inoue, A. 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, pp. 9–19.
[32] K.M.C. Tan, R.A. Maxion, Determining the operational limits of an anomaly-based intrusion detector, IEEE J. Selected Areas Commun. 21 (1) (2003) 96–110.
[33] D. Gao, M.K. Reiter, D. 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, 2006, pp. 19–40.
[34] R. Sekar, M. Bendre, D. Dhurjati, 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, IEEE Computer Society, Washington, DC, USA, 2001, p. 144.
[35] L.E. Baum, T. Petrie, G. Soules, N. Weiss, A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains, Ann. Math. Stat. 41 (1) (1970) 164–171.
[36] Stide website – source code of stide and system call data sets, http://www.cs.unm.edu/immsec/systemcalls.htm (Last accessed July 2010).
[37] M.A. Kramer, Nonlinear principal component analysis using autoassociative neural networks, AIChE J. (1991) 233–243.
[38] L. Manevitz, M. Yousef, One-class document classification via neural networks, Neurocomputing 70 (7–9) (2007) 1466–1481, doi:http://dx.doi.org/10.1016/j.neucom.2006.05.013.
[39] J. Lee, S. Cho, J. Baek, Trend detection using auto-associative neural networks: intraday kospi 200 futures, in: Proceedings. 2003 IEEE International Conference on Computational Intelligence for Financial Engineering, 2003, pp. 417–420.
[40] N. Japkowicz, C. Myers, M. Gluck, A novelty detection approach to classification, in: Proceedings of the Fourteenth Joint Conference on Artificial Intelligence, 1995, pp. 518–523.
[41] D.-K. Kang, D. Fuller, V. Honavar, Learning classifiers for misuse and anomaly detection using a bag of system calls representation, in: Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC, 2005, pp. 118–125.
[42] C.M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, Inc., New York, NY, USA, 1995.
[43] J. Erickson, Hacking: The Art of Exploitation, No Starch Press, 2003.
[44] H.G. Kayacık, Can the best defense be a good offense? evolving (mimicry) attacks for detector vulnerability testing under a black-box assumption, Ph.D. thesis, Dalhousie University (2009).