Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Rashid Waraich

Automated Attack Signature Generation: A Survey

Semester Thesis SA-2005-38
July 2005 to September 2005

Tutors: Daniela Brackhoff, Bernhard Tellenbach
Co-Tutor: Thomas Dübendorfer
Supervisor: Prof. Bernhard Plattner

Abstract

Hardening IT infrastructures of today's web-centric society against any form of attacks is a critical factor for the success of internet services. The arising expenses in case of violations of confidentiality, integrity or availability (CIA) of provided data and services are hard to estimate, but usually substantial. In recent years lots of research about how to increase the security of IT infrastructures has been done. The emerging NoAH project focuses thereby on the field of automated attack signature generation. This survey is part of the NoAH project and helps in getting an overview about existing approaches in the field of automated attack signature generation. Criteria for analyzing and comparing existing methods are introduced. The goal of this survey is to lay a base for future work in the area of automated attack signature generation and related research fields.

Contents

Chapter 1

Introduction

1.1 Motivation

In recent years, the internet was frequently targeted by worms. Alarming is not only the fact, that worms like Nimda [26] were able to infect millions of hosts [9], but also the spreading speed. For example the Sapphire worm [28] infected all vulnerable hosts attached to the internet within 10 minutes, after the worm was released. Indeed, worms are becoming a serious threat to a web-centric society. Not only the industry suffers from loss of productivity due to worm epidemics, which cost billions of dollars [30], but also consumers are becoming aware of such attacks, when flights are late or cancelled, emergency phone services do not work as they should and web pages are downloaded at rates, which were thought to be history since the replacement of dialup modems [27].

Manual signature generation, which has been the traditional way of containing worms, is rendered useless against modern worms [33]. In recent years, lots of research projects have been launched, which aim to generate attack signatures in an automatic fashion. One project for countering worms, without human-mediated reaction, is the *NoAH* project. The European Network of Affined Honeypots (NoAH) [61] project aims to design an infrastructure to counter cyber attacks. The project is based on honeypot technology [4] so that attacks can be detected at an early stage. Honeypots will be used in order to gather and correlate data from geographically distributed sites. After having identified an attack, a signature for the attack will be automatically generated and distributed to interested sites.

Until now, to the best of my knowledge, there has not been any survey about automated attack signature generation. For projects like NoAH, which aim to automatically generate signatures for attacks and to distribute them with little delay, a survey like this is a good starting point.

1.2 The Task

The task of this work consists of two major subtasks:

1. *"Create a survey of attack signature generation mechanisms including the thereby used input and output data and information about tools/algorithms that use these signatures to identify an attack."*
2. *"Propose an appropriate classification for the attack signature generation mechanisms."*

1.3 Signature Definition

The word *signature* will be used throughout this report in the following sense: *Any collection of characteristics, which allows classification of given input as benign or malicious, is called a signature*. A definition is required, as in intrusion detection literature, the word signature is often used as a synonym for the word *pattern* [34]. But the definition given above is more general and fully consistent with the definition of this term in [33].

1.4 Overview

The remainder of this report is structured as follows. Chapter 2 gives some background knowledge. In chapter 3 two network intrusion detection systems (NIDS) are presented, in order to understand how signatures are generated manually. Chapter 4 presents and discusses twelve approaches for automatic attack signature generation. In chapter 5 classification criteria are defined and the signature generation mechanisms presented in chapter 4 are classified according to them. Finally, in chapter 6, a conclusion summarizes the report and an outlook is given.

1.5 Related Work

To the best of my knowledge, no survey about automatic generation of attack signatures has been done prior to this work. Some of the papers about automatic attack signature generation (see Chapter 4) mention in their related work section other existing approaches. But they only give a rough overview about this topic. Too often, they do not compare the approaches or do not specify any classification criterion. One reason for the lack of any detailed work about automatic attack signature generation is, that most papers in this field were written in recent years.

Chapter 2

Background

2.1 Worms

Worms exploit software vulnerabilities, in order to get control over machines. The causes of such vulnerabilities are often software bugs, which for example allow buffer overwriting without boundary checks. Such buffers can be exploited by worms, to overwrite a return address, in order to redirect the program flow to a sequence of executable code contained in the network request: The worm's code. If a worm succeeds in infecting an attacked host, it tries to replicate itself by infecting other hosts. Some worms scan IP addresses to identify other vulnerable hosts prior to sending the exploit, while others select the IP address to attack randomly. As this process of replication and infection goes on, one can see that this kind of spreading leads to an exponential growth in the number of infected hosts, as the time passes. Knowing this fact, the spreading speed of worms should not be surprising anymore. A number of factors are relevant for not only boosting-up the spreading speed, but also for the total number of hosts, which are finally infected. One of these factors is the lack of diversity in used operating systems and other software. But the use of such mono cultural software has also some advantages. For instance, when a patch for a specific vulnerability is released, a large amount of hosts can be protected against future attacks exploiting the same vulnerability. But patches are often released several days after the attack has struck. So as not to be delivered defenceless to attacks, people often install so-called *intrusion detection systems* (IDS) [16], on their hosts or network gateways.

Detecting Scanning Worms

To detect new worms, common characteristics of many worms can be utilized: When a worm tries to further spread, it has to pick a new victim. The selection of such a victim is often done by randomly picking one address from the whole space of IP addresses. As not all addresses are used by hosts, probing a non-existent host will lead eventually to an ICMP unreachable message [29] or no message at all (dark holes). In order to monitor such randomly *scanning worms*, an approach called *network telescope* [10] can be used. A network telescope monitors some portion of the routed IP address space, which is unused. If some traffic arrives at the network telescope, one can infer that some malicious activity is going on in the Internet, such as the spreading of a worm. The effectiveness of this method for detecting worms depends on the size of monitored IP address space.

Another approach to detect new worms is the usage of *honeypots* [4]. Honeypots are monitored hosts, which are placed at IP addresses, where no services are supposed to be provided. Although honeypots have services installed on them, there is no reason for a benign user to contact them. As a consequence, any traffic to a honeypot can be classified as suspicious. But it has been pointed out [62] [20], that honeypots often receive packets from benign users for example due to badly configured applications. For this reason, some designers of honeypot systems have proposed [20], that as an evidence for a worm or intruder, unsolicited outbound traffic generated by the honeypot must be present.

Hit-list Scanning Worms

The problem with random scanning worms, from the attacker's point of view, is that they start spreading very slowly. So the most time a random scanning worm needs for spreading is during its starting phase. To accelerate the start, attackers have found a new method, called *hit-list scanning* [63]. In this method, the attacker collects a list of a certain number of potential vulnerable hosts in advance. This list is called hit-list. The worm is released with a copy of the hit-list attached to it. When the worm has infected an initial host, it chooses another host from the hit-list. Then the worm replicates itself, but divides the hit-list into two parts, sending one to the new target host attached to the replicated worm and keeping the other part. This procedure continues recursively. When the hit-list is finished, the worm continues spreading by searching for new targets through scanning.

The difficulty in stopping hit-list scanning worms is that they can only be detected when the hit-list is used up and the scanning begins. But as the starting phase of worm spreading is over by that time, the worm will often be detected too late by the methods presented in the previous subsection.

2.2 Intrusion Prevention Systems

Intrusion prevention systems (IPS) are strongly related to IDS. The main difference between them is that IDSs only try to detect intrusion, whereas IPSs also try to prevent intrusion. Only the structure of an IPS is given below, as one can look at an IPS as an extension of an IDS.

IPSs are used to monitor network traffic and events on computer systems. Typically, an IPS can be divided into three components, as shown in figure 2.1 [5]: An *information source*, an *analysis engine* and a *decision maker*. The information source component monitors data from different sources, like system calls, network traffic and applications. Then it converts the collected data into a format, which is suitable for the second component, the analysis engine. The main purpose of the analysis engine is to detect intrusive behaviour. Therefore, this component is considered as the most important part of an IPS, and is often used as a main classification criteria in surveys about IPS and IDS [16]. But without going into details, there exist two major approaches [34] to detect intrusive behaviour: *Misuse detection* and *anomaly detection*. Systems using misuse detection look for well-defined patterns in data coming from the information source. Misuse detection is therefore only effective if the pattern of an attack is known. On the other hand, misuse detection is almost always useless against new or unknown attacks. As opposed to misuse detection systems [24], systems based on anomaly detection have the potential to detect new attacks. Anomaly detection systems [25] rely on a profile of normal behaviour and report deviations from this behaviour as suspicious. An anomaly detection system often analyzes system activity during a learning phase taking place in a clean environment and classifies patterns of such activity as normal. When during normal operation an anomaly detection system finds a pattern, which is not covered by the normal behaviour pattern, it reports a possible intrusion. A major concern regarding today's anomaly detection systems is, how to tune them so that they produce both low *false positive* rates and low *false negative* rates [34]. False positive is called a misclassification of legitimate activities as malicious ones, also called *false alarm*. False negative is a misclassification of malicious activities as legitimate ones. The reason for such misclassification lies in the fact that there might exist legitimate activities, which lie outside the normal behaviour pattern, and are therefore classified intrusive (false positive). On the other hand, there might exist intrusive activities, which match the normal behaviour pattern of the anomaly system, and thus do not raise any alarm (false negative). The result of the analysis engine, whether it is based on misuse detection or anomaly detection, is handed over to the third component of the IPS, the decision maker. The decision maker applies some rules on the output of the analysis engine and decides what should be done. For example, if malicious traffic is discovered, a possible rule of the decision maker could state, that this traffic should be blocked. Therefore, the decision maker is a tool to increase the usability of an IPS, such that decisions are taken automatically using predefined rules, instead of burdening a pc owner or system administrator with taking such decisions manually.

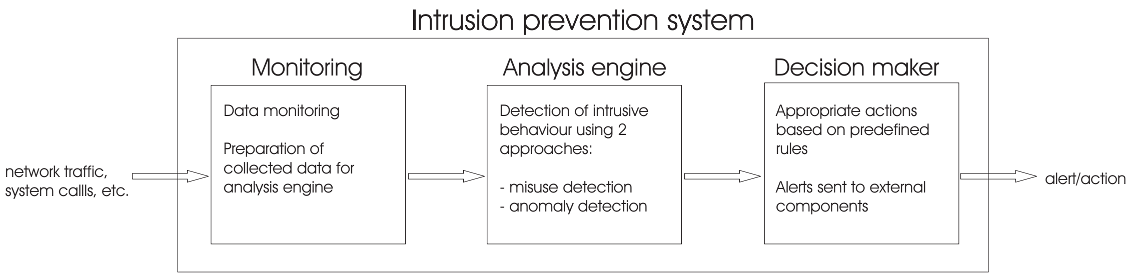

Intrusion prevention system
Monitoring
Analysis engine
Decision maker
network traffic, system callls, etc.
Data monitoring
Preparation of collected data for analysis engine
Detection of intrusive behaviour using 2 approaches:
- misuse detection
- anomaly detection
Appropriate actions based on predefined rules
Alerts sent to external components
alert/action

Figure 2.1: Components of an intrusion prevention system

2.3 Intrusion Detection Methods

As mentioned previously, the IPSs are often categorized into two main categories according to the detection method they use, namely misuse and anomaly detection. Four intrusion detection methodologies, which were presented in [33], are described below.

Pattern Matching

Pattern matching compares patterns defined in the signature with some input. For example, if in a network packet, the IP version, the transport protocol and a string in the payload matches a certain pattern, a possible intrusion has been identified.

This method can be deployed on several network protocol layers, because of its generality. But this method is very specific, as it tries to connect an exploit with a pattern. The problem is that if the pattern, which is engineered into a signature, is not unique to a certain attack as was thought by the author of the signature, this approach will lead to lots of false positives. On the other hand, any modification to an attack payload, which is not covered by the signature, will lead to false negatives. This approach only looks at one packet at a time. Therefore, this method is not suitable for stream-based traffic, as used by HTTP.

Stateful Pattern Matching

This approach adds a new concept to the standard pattern matching approach: Keeping state of several packets, which form a stream. By using this method, patterns can be matched, which are distributed over several packets.

This method has the advantage over standard pattern matching, that it is suitable for stream-based traffic, where an attack payload could be distributed over several packets. But this method must keep state, which consumes more resources than standard pattern matching.

Protocol Decode-based Analysis

This approach is a further development of the stateful pattern matching method. This method is implemented by decoding a conversation in the same way, as a client or server would do. After the decoding process, the protocol fields are identified and checked against the protocol's definition in the RFC. If a field is invalid or too long according to the protocol's definition, an alarm is raised.

This method minimizes the chances of false positives, if the protocol is well defined and it is pushed towards a correct use of the protocol. But the problem is that in reality, the RFCs are often ambiguous and give developers the freedom of interpretation, when implementing the

RFCs. Furthermore, the implementation of this approach by building a well-engineered parser is more time consuming than the prior two approaches.

Anomaly-based Analysis

Anomaly-based analysis looks for network traffic, which deviates from normal traffic behaviour. But before this method can be used, it must be defined, what normal behaviour means. If normal behaviour is hard-coded in the system, then the analysis method is called heuristic-based. But a system can also learn about the normal behaviour during a training phase. Afterwards, deviations from the normal behaviour are classified as possible intrusions.

This method has the potential to find unknown attacks. The main difficulty with this approach is how to fine tune the parameters, which decide when the system should raise alarm. It is often reported [34], that anomaly-based systems have higher false positive rates than the other analysis methods described above. Furthermore, systems using this method depend on the environment, where they have learned what normal behaviour is.

Chapter 3

Manual Signature Generation

Almost all signatures and rules deployed in today's antivirus software, firewalls and IDSs are developed manually by skilled security experts. A detailed analysis and reverse code engineering of a virus is shown in [59]. Manual generation of a worm signature typically consists of the following steps. First, one or more instances of the worm have to be captured. Then the binary code of the worm is converted to assembler code. It is then analyzed by a human expert, who looks for suspicious code sequences and in particular for invariant code between all of the captured worm instances. If such code sequences are found in the assembler code, the corresponding bytes in the binary code of the worm are identified, and used as signature. If network traffic traces containing the worm are available, further characteristics can be incorporated into the signature, such as which port the worm uses for spreading or which flags in the packet header are set, and which transport layer protocol is used. All these characteristics are collected and put into a signature or rule for the worm, which can then be used for example in IPSs to protect hosts against that worm.

In this chapter, two popular network intrusion detection systems (NIDS) are presented together with the language they use to describe rules. Looking at the languages, in which such rules are written, gives an insight into the power of manually generated signatures. It is not the claim, that this chapter provides an introduction to IDS. A survey about IDS with classification of the signature types they use, can be found in [16].

3.1 Bro

Overview

- Open source (release information can be found under [64]).
- Implementations available for Digital Unix, FreeBSD, Linux and Solaris OS.
- Currently no GUI based administration tool available.

System Structure

Bro [7] is an NIDS, which monitors traffic that enters or leaves a network. As large volumes of high-speed data might be exchanged between the network and the outside world, the system has to be built in a way that no packets are dropped due to slow processing. For this reason, the system is structured into layers. The lowest layer has to deal with the greatest amount of data, but it does the least processing on it. When going up the layers, the amount of data to be processed decreases. This in turn allows more sophisticated processing algorithms to be used.

The lowest layer resides just on top of the network and consists of libcap [8], a packet-capturing library. Packet filtering is performed on packets arriving at the system, in order to filter out packets, which are not of interest to the system: Only packets with certain source or destination ports (e.g. FTP, Telnet), some TCP control packets and IP fragments are passed to the higher layer, the event engine. The event engine performs some validity checks on packet headers, such as verifying the IP header checksum. On this level, also IP fragments are reassembled, in order to analyse the whole IP datagram. If certain checks fail, an event corresponding to the problem is generated and the packet is dropped. On the other hand, if all checks were ok and if the datagram does not belong to an already registered connection, a connection state is created, which contains the source and destination IP addresses and ports. The system keeps track of changes of the connection's state and generates different events for different state transitions (e.g. connection established, connection rejected). These events are handled by the next upper layer, which consists of a *policy script interpreter*. The policy script interpreter runs an event handler script corresponding to the triggered event. Such scripts are written in the *Bro language*. These scripts can generate notifications of a detected intrusion, store the content of a packet to the hard disk or even generate new events.

At the moment, Bro allows application-specific processing for six applications (e.g. Telnet, FTP, Rlogin). But Bro is also extensible and allows adding new protocol analyzers to the event engine. For every event, which can happen during the protocol analyzing phase, an event handler script has to be written in the Bro language. In the next subsection, a short description of the Bro language is given.

Bro Language

Security policies in Bro are written in a special language, the *Bro language*. This language is strongly typed, allowing discovery of type inconsistencies already at compile-time. As atomic types, the Bro language introduces some new types besides the one's known from traditional programming languages (eg. `bool`, `int`, `double`, `string`). Examples of new types are `port` and `addr`, which stand for port number and IP address. Using the keyword `record`, more complex types can be built, based on these atomic types. For example a connection type could be introduced as a record of two IP addresses and port pairs.

In order to specify Bro security policies, the type `table` could be especially helpful. This type is similar to arrays in C, except that the type of the index does not have to be of type `int`. Instead, any type can be chosen for indexing the array. An interesting type, especially for describing worms using some polymorphism, is a type called `pattern`. It is based on Unix-style regular expressions.

Only one new operator is added to the operators, which are known from C, `in`. This allows for example to check, if there exists a value in a table corresponding to a given index.

The Bro language consists of only a small number of statements. Example of such statements are if-then-else statements and an `event` keyword, which allows to generate new events. But there are no for-style loops in the language, because it is feared, that this could lead to arbitrary large processing times, leading to dropping of packets. But generating loops is still possible in Bro, for example through recursive calls to functions.

3.2 Snort

Overview

- Open source (further information can be found under [65]).
- Implementations available for Linux, Solaris, Free/Net/OpenBSD and MacOS X OS.
- GUI based administrative and analysis tools are available.

System Structure

Snort [7] is an NIDS, which uses libpcap [8], to sniff and filter packets. Snort is based on three subsystems: a *packet decoder*, a *detection engine*, and a *logging/alerting subsystem*.

Each component of the packet decoder tries to extract useful elements from the raw network traffic into predefined data structures. The decoding routines are placed at each layer, beginning at the protocol stack and ending at the transport layer. During this decoding phase, pointers are set into the packet payloads, in order to accelerate further processing by the detection engine.

The detection engine consists of a collection of rules. These rules are written in a Snort specific rule language, which is described in the next subsection. The collection of rules is managed in a way, which allows putting together similar characteristics of rules into a separate data structure. This helps speeding up rule matching for packets, which are received from the packet decoder. The first rule that matches a packet, triggers the action defined in that rule, which is taken by the logging/alerting subsystem.

The logging/alerting subsystem allows the user to choose, if and how it should log a packet that matches a rule. Snort allows three methods for logging, e.g. logging each decoded packet into a separate file or logging several packets in binary format into a single file. For alerting the user, Snort allows the user to choose between five methods, e.g. writing alerts to the systems log or to text files, or let a window popup at some user-specified console.

Snort Rules Language

Snort rules allow defining, in which packets what pattern should be searched for, and what should be done, if a packet matches that pattern. It can be specified, which packets Snort should filter out for further inspection. For example, only TCP packets coming into the network or packets leaving the network with a certain source port and destination port can be chosen. Such filtered packets can then be matched against further characteristics defined in the rule, such as TCP flags, the TTL field in the IP header or the packet content. If a packet matches the predefined pattern in the rule, Snort can take actions as specified in the rule, such as logging the packet or alerting the user.

3.3 Comparison of the Bro and Snort Signatures

In this section an attack signature for the Windows Media Player exploit is given in Bro and Snot languages (see below), which was taken from [37]. By looking at this example signature, some differences between the Bro and Snort languages are discussed.

For Bro, both the request and response message corresponding to the exploit are present in the example signature. But as Snort does not have the ability to relate a request with a response message, only the request is part of the signature.

```
Bro:

signature nsiislog {                           signature http_200_ok {
 ip-proto == tcp                                ip-proto == tcp
 dst-port == 80                                 src-port == 80
 http /.*/scripts/nsiislog.dll                  payload /.*HTTP\/1\.. 200/
 requires-signature-opposite ! http_200_ok      event ''HTTP 200 OK''
 tcp-state-established                          tcp-state-established
}                                              }
```

```
Snort:

alert tcp any any -> 10.0.0.0/8 (msg: "(msg:WEB-IIS nsiislog.dll access";
flow:to_server,established; uricontent:"/scripts/nsiislog.dll") nocase;
reference:...)
```

The Bro language is much richer in its expressiveness than the Snort language. For example, Bro allows incorporating regular expressions into signatures, whereas Snort only allows for exact string matching. Regular expressions can be very helpful in finding polymorphic worms, which can change certain parts of their payload. Bro keeps state of TCP connections, which allows tracing requests and response messages. Snort on the other hand does not keep any state. Both of these advantages of Bro over Snort are visible in the example above. The star in the Bro signature (both in request and response), can match any arbitrary number of characters. Furthermore, the Bro signature relates the request message with the response message (requires-signature-opposite ! http_200_ok), which would not be possible in Snort. But these advantages of Bro are not for free: Matching exact strings is much more efficient than matching regular expressions. And keeping state consumes resources, and makes a NIDS susceptible to attacks [7] where the monitor can run out of resources. Launching such attacks on stateless NIDS is much more difficult.

Although the Bro language allows for many constructs, which are missing in Snort(e.g. if-then-else), Snort enjoys a greater popularity than Bro. One reason for Snort's popularity lies in its simplicity. If one wants to write Bro signatures, one must be able to do programming. But writing Snort rules is even possible without having programming skills, as Snort rules always have the same fix structure.

Chapter 4

Automatic Signature Generation

As highlighted in the introduction, manual signature generation is too slow to contain modern worm outbreaks. In the following sections, twelve automatic attack signature generation mechanisms are presented and discussed. Particular attention has been paid to the input and output of each signature generation mechanism (SGM).

4.1 Nemean

Overview

Input: Packet traces from a honeynet [39] deployment.
Output: Regular expressions.
System/Algorithm based on [37].

Input

Honeynet [39], consisting of a network of honeypots, is deployed on unused IP address space. HTTP requests for the unused address space are routed to a host, running a real implementation of a HTTP server on it. NetBIOS/SMB related packets are sent to a virtual honeypot similar to honeyd [2]. All packet traces seen at the honeynet are collected and sent to the Nemean [37] signature generation algorithm.

Signature Generation Mechanism

As input for the SGM, Nemean receives packet traces from honeynet [39], a network of honeypots. As a first step, *transport-level normalization* [37] is performed on the packets. The reason for performing transport-level normalization is that normally OS do validate characteristics of received packets, such as packet header fields. If a certain packet header field is invalid, the packet is dropped. As Nemean receives all traffic trough a packet sniffer, attackers could try fooling the Nemean system by sending packets to it, which are handled by Nemean in an other way, than by an OS [41]. The goal of transport-level normalization is to resolve such ambiguities, by performing the same steps on packets, as an OS would do.

The second step is *flow aggregation*. During flow aggregation, normalized packets from the previous step are composed into flows and stored. After a certain period of time, flows expire and are converted into connections. Connections consist of requests and responses. Connections between the same pair of hosts are grouped together in order to form a session.

In the next step, *service-level normalization* [37] is applied on sessions. The reason for performing service-level normalization is that Nemean should perform the same steps on invalid fields of application protocols (eg. HTTP), which a server implementation would do. After performing server-level normalization, the normalized sessions are encoded in an XML-format, which is suitable for further processing by the clustering modules.

Clustering is performed both on connections and sessions separately. For clustering an algorithm called star clustering is used, which can cluster documents according to a similarity metric [38]. The goal of clustering is the following: Different attacks should be separated into different clusters, such that one cluster does only consist of one single attack or its variations. Out of every cluster a finite state automaton (FSA) is created, representing all connections and sessions respectively in the cluster. The transitions in the automaton are labelled with request/response message strings.

Finally *generalization* steps are performed on the FSA, using different generalization algorithms for connection [42] and session [40] FSAs. By the generalization step, invariant and variant parts of a FSA are identified: Transitions with several possible labels are identified as variant part of the attack. The generalized FSA forms a signature for a polymorphic worm, which can be converted into a regular expression.

Output

The SGM outputs an attack signature both on connection and session-level. Such a signature is described by a finite state automaton (FSA), with a *starting state* and possibly many different paths which lead to an accepted *final state*. If an attack consists only of one single request, it can be matched with a connection-level FSA. If a request string is given as input to the FSA and a final state is reached, a possible attack was identified. A session-level FSA consists of requests and responses. If a sequence of requests and responses given as input to the session-level FSA leads to a final state, a possible attack was identified. As this FSA is a deterministic finite automaton, it can be converted to a regular expression.

Evaluation

As it has been pointed out in [37], building a perfect normalizer is a difficult task, as ambiguities on transport-level are OS dependent and ambiguities on service-level are server-application dependent. So there exist two main difficulties with the normalization approach. The first issue is that this approach is not general: Although one can build transport-level normalizers for the few main stream OSs, building a service-level normalizer is a much complicated endeavour. Building a new normalizer for every server-application that needs to be protected, and doing this for every application protocol does not seem to be practical for ubiquitous employment. The second problem with this approach is that if the built normalizer does not solve all ambiguities, it exhibits explicitly vulnerabilities to potential attackers. One can not argue that the intruders will not use the few uncovered situations by the normalizer, because intruders especially look for such cases.

In [37], service-level normalizers for only two services (HTTP and NetBIOS/SMB) were implemented and discussed. But the implemented normalizer for the HTTP protocol does not handle all ambiguities. So it is worth thinking about this service-level normalizer approach, only if it is practicable in a large scale. Fact is that even building a perfect normalizer for a single service seems to be a difficult task.

The signature generated is able to detect attack strings, even if data reordering and data modification is performed on the variant part of the payload. The SGM uses a generalization step, in order to conclude from captured variants of the worm to unseen variants. It is obvious that a representative number of variants of the worm must be at hand to the SGM, in order to generate signatures, which can cope with polymorphism.

4.2 Dynamic Taint Analysis

Overview

Input: Network traffic.
Output: 3 bytes.
System/Algorithm based on [11].

Input

By default, the SGM considers all input from network sockets in the signature generation process. But the SGM can be configured, in oder to take also other sources into consideration, such as the standard input or data from certain files.

Signature Generation Mechanism

This system aims to detect overflow attacks. All data coming from untrusted sources is marked as *tainted* data. There exists a configurable policy for defining, which data source is not trusted. By default, all data coming from network sockets is considered untrusted and therefore tainted. Whenever a source register of a CPU operation is loaded with tainted data, the result of the operation is also marked as tainted, when the result is written back to memory. For CPU operations, where the result of an instruction is independent of the source registers, the result is never marked as tainted. Furthermore, whenever tainted data is copied or moved around in memory, the target memory location is marked as tainted.

The system allows to configure the policy for defining, what kind of data in memory should not be overwritten by tainted data. By default, it is not allowed to overwrite destination addresses of control flow operations with tainted data and the use of tainted data as format strings in printf like functions is prohibited.

If the system detects an attempt to overwrite locations in memory with tainted data, for which according to the defined policy it is prohibited to overwrite them with tainted data, a post-analysis of the attempted attack begins. The three higher bytes of the value, which was used during the overwrite attack, represent a signature for the attack. The idea behind this choice is that the value, which is used for the overwrite attack can only point to one or a few fix memory locations, in order to be successful [11].

Output

The signature produced by the SGM consists of the three most significant bytes of the value, which were used during the overwrite attack.

Evaluation

This system has a great advantage above other similar systems [12], as it does not need source code of the application it should monitor. This is essential, as source code of commercial software is hardly ever available.

Signatures generated by this system are only three bytes long. Such short signature may generate lots of false-positives. But the authors of the system [11] have made some suggestion for future development, to generate more accurate signatures. For example, it could be identified, which minimal length a certain part in a request must have, such that an attack can succeed.

A major drawback of this system is that it has a very poor performance: In some experiments, it slowed down the normal execution of programs up to 37 times [11]. A major reason for this bad performance is that programs monitored by this system run in an emulation environment, called Valgrind [13]. Furthermore, the system changes the code of monitored programs by adding checks after every operation on data and registers. But this additional code, also called instrumentation code, is not optimized yet. The authors plan to address both of these issues. They argue that the use of Valgrind is one main reason for the poor performance of their system. They want to replace it by DynamoRio, which is reported [14] to perform better on the same programs than Valgrind.

As the monitoring with this system is not cheap regarding the CPU time required, its deployment seems suitable on low-load servers and honypots, as suggested by the authors of the system [11]. But also on high-load servers, a deployment is possible only if a portion of some randomly chosen requests are checked with this system.

This system detects worms by evidence, not by suspicion. This renders payload encryption and obfuscation techniques [15], which are often used by polymorphic worms to evade NIDS, useless against this system. This approach not only detects most kinds of overwrite attacks, but it also produces signatures, which are robust against polymorphism. If the attack payload is encrypted, then the signature is only useful for an IDS that monitors network data propagation in the memory, which is true for the presented system.

As mentioned above, a system that can use the generated signature also in case of encrypted attack payload, must emulate a monitored program in a similar way, as the presented system does. So it is worth thinking about, why one would like to use a new system, which is similar to the one used in the SGM, only to be able to use the generated signature. Instead it might be more appropriate to install the detection part used in the presented SGM directly into an IDS.

4.3 Honeycomb

Overview

Input: Network traffic from a honeypot.
Output: Signatures suitable for use in Snort and Bro.
System/Algorithm based on [1].

Input

The input to the Honeycomb SGM consists of network traffic seen at a honeypot.

Signature Generation Mechanism

This signature generation system is based on honeyd [2], an open-source honeypot implementation. Honeyd allows the configuration for the OS type and services it should simulate. When traffic is seen at the honeypot, a protocol analysis is performed on the packet headers at network and transport layer: Anomalies in the headers, such as invalid fields or unusual field combinations, are recorded as part of the worm signature.

In the next step, a comparison on packet contents is performed on different connection flows using the longest-common-substring (LCS) algorithm [3]. This is done in two ways: *horizontal detection* and *vertical detection*. In the case of horizontal detection, if the number of the current message in a connection is n, then this packet is compared with the n-th message of all connections with same destination port. With vertical detection, two connections are compared with each other by concatenating packets in each flow to a configurable maximum length. Then the two connection flows are compared. All detected patterns are added to a signature pool. Signatures from the signature pool are converted periodically by an output module into a format, which is suitable for use in the Bro [7] and Snort [6] NIDSs.

Output

The generated signatures include information about the attack, such as which transport layer protocol and destination port was used and some part of the attack payload string. These signatures can be converted into a format suitable for use in Bro and Snort NIDSs.

Evaluation

Horizontal detection is suitable for protocols, where the messages must have fix order and can not be interleaved with any other messages: The n-th messages in different instances of the same attack must be the same. But protocols where messages can be interleaved, create some sort of asynchrony, if seen from the point of view of the horizontal detection. Here is where the vertical detection fits in. It compares longer sequences of connections with each other, such that the LCS algorithm is able to detect shifted strings.

There are limitations to the length of bytes, which are concatenated in the vertical detection phase. An attacker who is aware of this limit can evade the system by adding some harmless messages before the attack messages, so that the first part of the worm is in one "comparing-set" and the end in another. Therefore, the signature found for the worm in each "comparing-set", eventually becomes too short to be usable as a signature.

One drawback of this system is that it can not generate signatures for polymorphic worms. The reason for this handicap is that the generated signatures are not based on the idea of the worm payload consisting of variant and invariant parts. Due to this fact, also variant parts of the worm are likely to be incorporated into the signature.

Honeycomb can be fooled by attackers, to generate signatures for legitimate traffic. This can be achieved by sending harmless requests from a set of hosts to the same set of randomly chosen IP addresses. Eventually a honeypot will be found with a honeycomb implementation running on it and will generate whichever signature the attackers desire. To counter this sort of deception attempts, the exchange of information between honeypots could be used. So if requests at different honeypots come "too" frequently from the same host or set of hosts, such hosts should be marked suspicious. Due to the spreading behaviour of scanning worms, across all honeypots, a more or less random distribution of the probing hosts should be sighted.

4.4 IBM-94

Overview

Input: An identified virus example embedded in a program executable.
Output: One or a few machine code byte sequences.
System/Algorithm based on [60][1].

Input

The input to this SGM is a virus, which is embedded in some program executable. It is not mentioned in the paper, how this input was obtained.

Signature Generation Mechanism

As input, the signature generation mechanism is given an identified virus example that resides in a program executable. The SGM consists of two independent steps: First is *signature extraction*, which consists of finding some machine code byte sequences, which are likely to be present in all instances of the virus. Thereafter *signature evaluation* is performed,

[1] In [60], the authors did not give any name to the proposed algorithm; it will be called *IBM-94* in this report.

which selects one or a few signatures from the set generated during the first step. The signature(s) chosen are those with the least probability of producing false-positives when deployed.

Signature extraction is done as follows: The sample virus is run on a separate machine, which is equipped with decoy programs to attract virus infection. To speed up the infection of the decoys, they are placed in directories, which are preferred by viruses. From time to time the system looks for changes in the decoy programs. If they were modified, they are considered to be infected and are stored at some special location. After having enough modified decoys, their infected areas are compared with each other, in order to find regions, which are invariant from one infection instance to another. If the invariant regions are too small, then the virus is considered as too polymorphic and a human expert has to analyse it. But if the virus is not excessively polymorphic, then the code portion and data portion of the invariant regions are separated and only the code portions are taken as candidates for signatures. The data portions are not taken as part of the signature, because they can be modified frequently, in order to help viruses evading virus scanners.

For signature evaluation, a well known method from the speech recognition community is used: Trigram technique or generally *n-grams* [44]. N-grams, for a given data set, consist of all substrings of length n in the given data. From a large database of identified viruses, all n-grams are computed with their corresponding frequencies. These frequencies are used to extrapolate to statistically all viruses, even such which have not been seen yet. If a new candidate signature is given, this method can assign a probability to it describing how likely it is that the same sequence of bytes could turn up in some legitimate program. In this manner, the signatures with the lowest false-positive rates are selected.

Output

One or a few strings are selected as a signature for the virus.

Evaluation

The generated signature consists of one or a several strings, which seem to be invariant between instances of the same virus. There is a constraint of the maximum length of the produced signature strings. It is not mentioned in the paper either if the strings are ordered in the signature or not, in the case where a signature contains more than one string. As an ordered set of strings is more specific than an unordered set of strings for matching, it is assumed here, that the strings are not ordered. Unordered sets of strings have the potential to deal with polymorphism: Such as byte reordering, modification of the variant part of the payload and encrypted payloads where an invariant encryption/decryption routine is present in the payload.

The basic idea behind the signature evaluation method is that when looking at some program bytes, one can decide how characteristic these bytes are for malicious behaviour. An attacker, who has access to the database of viruses, which are used by the algorithm, can compute the sequences of length n that are most prevalent in the database. Then he could integrate a few of these sequences in his code. When the attacker releases his virus, the virus might be detected after a while. Then the virus will be analyzed by the presented algorithm. The byte sequences, that the virus author integrated into his code, will be selected as signature for the virus. But as the virus author has computed a lot of such prevalent sequences from the virus database, he can potentially release lots of new version of his virus by integrating other prevalent sequences into his virus code. What should be noted is that whenever a version of the virus is captured, the algorithm will not find a signature specific to the virus. Instead, a signature will be generated, which is not related to the virus at all. At this point, one could argue as the authors of the algorithm [60] do, that one must hide the database of viruses from the attackers. But an attacker actually does not need the whole database to find prevalent sequences, because it should be possible to derive such prevalent sequences also from a much smaller set of viruses due to their prevalence.

Another problem with this signature evaluation is the idea that the most prevalent sequences of bytes in the virus database will not be contained in benign code, as the byte sequences are really "evil". The fact is that it is not avoidable, that the most prevalent sequences in the virus database will also turn up in some benign program. For this reason, this approach could potentially lead to lots of false-positives.

To relieve the problem of false-positives, one could compute prevalent byte sequences from a database containing legitimate programs. These "benign" prevalent sequences could be deleted from the prevalent sequences computed from the virus database. By doing so, the false-positive rate will obviously decrease.

4.5 Autograph

Overview

Input: Traffic coming into a network.
Output: Signatures suitable for use in Bro [7].
System/Algorithm based on [29].

Input

Input to the Autograph system consists of all traffic, which comes into a network.

Signature Generation Mechanism

The Autograph system resides at the edge of a network and monitors all traffic coming into the network. If some external host attempts to reach non existing targets in the network more than s times, it is marked by the autograph system as a *scanner* and stored in a scanner-list. If a scanner finally connects to an existing host in the network, all traffic from it will be stored in a so-called *suspicious flow pool*. After a certain period of time, scanners are deleted from the scanner-list and packets in the suspicious pool as well, in order to ensure freshness of packets. All TCP flows in the suspicious flow pool are reassembled and each flow is grouped by the destination port. For each destination port, if a certain threshold number of flows are reached, the signature generation phase starts.

In the signature generation phase, all flow payloads are partitioned into *variable-length* content blocks. This is done using a method called *content-based payload partitioning* (COPP) [31]. To chop flows into content blocks, COPP slides a window of k-bytes over the flow content, one byte at a time. For each observed sliding window, a Rabin fingerprint is computed [32]. The COPP method uses a predefined *break marker* and a configurable *average content block size*. Whenever the Rabin fingerprint of a sliding window is equal to the break marker modulo the average content block size, a content block ends at that position.

Using COPP, all flows are chopped into content blocks. Such content blocks, which only appear in flows coming from a single IP address, are discarded. The idea behind this step is that these content blocks correspond to badly configured sources, which are not malicious. Afterwards, the frequency of all remaining content blocks is computed. The content block which was found in most flows is selected as a signature. Then all flows, where the selected content block was found, are deleted. For the remaining flows, the content block, which occurs in most flows is selected as a signature. This process continues, until the set of flows shrinks to a configurable fraction of the original flow count.

The signatures found in the last step are converted into a format, which is suitable for use in Bro [7]. Also a distributed version of the signature generation mechanism is proposed, where an IP address list of scanners is shared among autograph systems [29], which should lead to faster signature generation.

Output

The output of Autograph consists of strings, which are converted into the Bro signature format.

Evaluation

The authors of the system propose that the classification of the suspicious traffic could also be done using other methods like honeypots. In this case only minor changes are needed to the presented algorithm, in order to be deployed in a honeypot environment for signature generation.

This system has some problems concerning performance: The flow reassembly is an expensive operation [7], which is excessively used when a worm is spreading. So the system needs enough spare hardware such that it does not become a voluntary target of a DoS attack at the time when it is mostly needed.

The COPP method, which is used to chop flows into content blocks, is more suitable for computation of prevalent content blocks than other methods that divide flow contents into fixed-size, non-overlapping blocks [29]. The reason is that when frequencies of fixed-size, non-overlapping blocks are computed, they can easily be evaded by a worm, which inserts or deletes a single byte in its content. But there also exist some difficulties with the COPP approach. As this approach uses a break marker to decide when to end a content block, too short or too long content blocks may be generated. As too short signatures are highly unspecific, they may lead to lots of false-alarms. On the other hand, very long signatures are too specific, so that polymorphic worms may evade them easily by modifying a single byte. To cope with this problem, the authors [29] define a minimum and maximum content block size. But due to this changing of the COPP algorithm, it becomes a *hybrid* between the original COPP algorithm and the algorithm, which divides flows into fixed-size, non-overlapping content blocks. This hybrid approach also inherits the drawbacks of the algorithm that produces fixed-size, non-overlapping content blocks, which were mentioned before.

Autograph takes into consideration that a worm consists of variant and invariant parts. The generated signature strings have a limited size. Therefore, the generated signatures have the potential to deal with worms that use byte reordering and those who modify variant parts of the payload. These signatures also unveil worms using encrypted payloads, where an invariant encryption/decryption routine is present in the payload.

4.6 Paid

Overview

Input: Source code of application, which should be protected.
Output: Deterministic finite-state automaton (DFA) of system calls.
System/Algorithm based on [49].

Input

The input for the Paid SGM is the source code of the application that the Paid system should protect against control hijacking attacks.

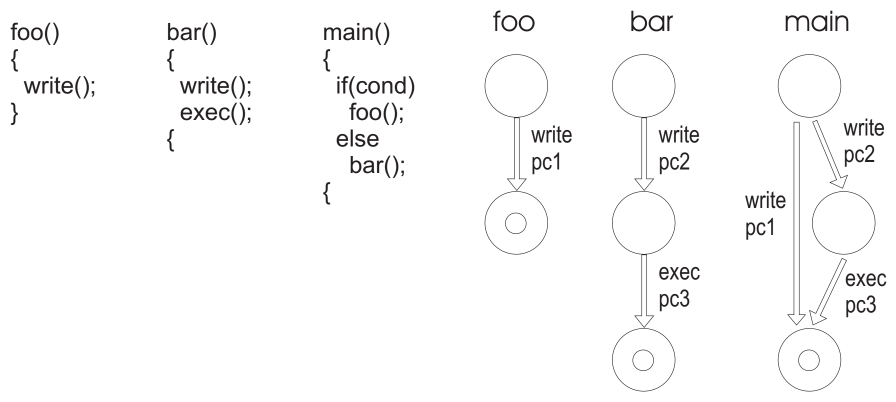
Signature Generation Mechanism

The goal of this system is effective defence against control hijacking attacks [50]. From the source code of an application, the system is able to automatically derive a system call behaviour model, a deterministic finite-state automaton (DFA) for the application in question. A verifier based on this DFA resides in the kernel, which checks each system call's legitimacy.

The main challenge in building such a DFA lies in the elimination of non-determinism: The call graph of an application is often a non-deterministic finite-state automaton (NFA), because of control constructs like if-then-else. To achieve this goal, an epsilon-transition removal algorithm is used on the NFA of the application, in order to remove non-system call edges. To remove non-determinism related to functions, which have many call sites, a method called graph in-lining [49] is used. On Linux, system calls must be made indirectly through system call stubs [49], which leads to another source of non-determinism: System calls through the same stub cannot be differentiated. This problem is solved by uniquely identifying each system call with a label. In order to remove the remaining non-determinism, which is left over after applying these steps, Paid has introduced a new system call named "notify". The last steps from a NFA towards a DFA are the following: The whole NFA is visited, in order to detect non-determinism, and the points in the NFA leading to non-determinism are marked. Then at every marked point, a notify call is inserted to remove the non-determinism there. By performing this step, all non-determinism in the NFA is removed, and it becomes a DFA.

Output

The output of the SGM is a DFA representing the system calls during normal execution of the program according to its source code.
An example of such a DFA is shown in figure 4.1. The DFA of the `main` program is composed of the DFAs of the functions `foo` and `bar`. The transitions are assigned unique labels, as described above.



```
foo()
{
  write();
}
```

```
bar()
{
  write();
  exec();
{
```

```
main()
{
  if(cond)
    foo();
  else
    bar();
{
```

Figure 4.1: Example of DFA in Paid.

Evaluation

As this system generates a behaviour model for legitimate actions, no instance of the attack worm or virus is needed for signature generation. Therefore, Paid is potentially able to detect new worms.

A drawback of Paid is that the source code of the monitored application is needed, which is hardly available for commercial software.

As the Paid system monitors the system call behaviour of a program, it is useless to apply any kind of polymorphic technique on the attack payload, in order to fool Paid.

Although features like "stack integrity check" [49] and "Random Insertion of Null System Calls" [49] have been used to improve the detection strength of Paid, it is still possible to launch mimicry attacks [51] against this system.

4.7 PADS

Overview

Input: Network traffic captured at double-honeypot [20] system
Output:

- Anomalous signature: byte frequency distribution (BFD) [20] at each position in the signature.
- Normal signature: BFD of legitimate traffic.

System/Algorithm based on [20].

Input

A *double-honeypot* [20] system captures the unsolicited outbound traffic of a honeypot by redirecting this traffic to a second honeypot. Any unsolicited outbound traffic at a honeypot delivers evidence that the honeypot is under control of an intruder by definition of a honeypot.

Signature Generation Mechanism

The input to the signature generation mechanism comes from a double-honeypot [20] system. According to the design proposal, a double-honeypot system is able to separate attack traffic from normal background traffic, which is often received by traditional honeypots [62]. Having received a number of variants of a worm from the double-honeypot system, the signature generation algorithm begins to generate a *Position-Aware Distribution Signature* (PADS), which should have the potential to match unseen worm variants. A PADS of length w consists of two parts, an *anomalous signature* and a *normal signature*. The anomalous signature assigns each of the w positions in the PADS a *byte frequency distribution* (BFD). The normal signature consists of a BFD of legitimate traffic.

In order to understand, how this signature is generated, the worm identification process needs to be discussed. If a sequence is to be examined with regard to existence of a worm, a window of length w slides over the sequence and computes a matching score for every window. This matching score is computed with the help of the PADS signature and a given formula. If the matching score for a window position is higher than a threshold, the sequence is assumed carrying a worm. The position of the window with the highest matching score is called *significant region*. The anomalous signature is the BFD of the significant region of all worm variants at hand during the signature generation process. It would be easy to compute the PADS signature, if the significant regions of the worm variants were given. This is a "missing data problem" [20]. Two algorithms were proposed to solve this problem. The first algorithm is Expectation-Maximization (EM) [21], which starts with assigning significant regions randomly to the given worm variants. As a next step, the PADS signature is computed based on these significant regions. From the obtained PADS signature, a new significant region is computed for each worm variant. This procedure is repeated until the average matching score does not improve significantly in an iteration step. A problem with the EM algorithm is that it may get stuck in local maxima. To improve the EM algorithm in this regard, the Gibbs sampling algorithm [22] is used, which is an example of simulated annealing [23]. With the selection of some random parameters, this approach provides the opportunity to jump out of local maxima, and finally reach the global maximum. When hopefully such a global maximum is found, the generation of the PADS ends.

Output

The SGM's output consists of a PADS. As described above, a PADS of length w consists of two parts, an anomalous signature and a normal signature. The anomalous signature assigns to each of the w positions in the PADS a byte frequency distribution (BFD). The normal signature consists of a BFD of legitimate traffic.

Evaluation

The double-honeypot [20] system design seems to be superior to traditional honeypots, as it classifies only unsolicited outbound traffic at the honeypot system as a worm. While traditional honeypots may receive lots of benign inbound traffic [62], unsolicited outbound traffic is an evidence that the honeypot is controlled by an intruder. But currently, the double-honeypot system is only an architecture design, for which an implementation and evaluation needs to be done.

In the PADS paper, the double-honeypot and the SGM is described in detail. But the paper does not explain, how the variants of the same worm are separated from other worms.

As described in the SGM subsection, a PADS is matched against a possible worm candidate by calculating a matching score: If this matching score is higher than a certain threshold, then the candidate is classified as a worm. PADS is based on byte frequency distribution of the positions in the signature. Therefore, this signature can deal with polymorphism such as minor modification of bytes and byte reordering in both the invariant and variant part of the attack payload.

4.8 PAYL

Overview

Input: Host inbound and outbound traffic.
Output: One or multiple strings.
System/Algorithm based on [43].

Input

The SGM in PAYL monitors a host's inbound traffic. If some suspicious traffic is found in the inbound traffic for port i, then PAYL takes also outbound traffic from the host with destination port i into consideration during the signature generation process.

Signature Generation Mechanism

The PAYL *anomaly detection sensor* [45] computes during a training phase the "normal profile" of a site using *n-grams* [44]. For a packet payload, an n-gram consists of any sequence of n consequent bytes in the payload. When a new packet arrives, all possible n-grams are computed for it and also the frequencies of these n-grams are registered. Then a formula is used to compute the *distance* between arriving packets and the n-gram distribution, which was seen during the training phase. If this distance is larger than a threshold and the incoming traffic was intended for port i, then such packets are put into a buffer list of "suspects" for port i. Any outbound traffic to port i, which is also detected as anomalous using the anomaly detection sensor, is compared with this buffer. For the compared strings, a *similarity score* is computed based on a formula, which requires the generation of the longest common substring (LCS) and the longest common subsequence (LCSeq) of the two strings. If the similarity score is greater than a threshold, the outgoing traffic is blocked. As a by-product of the correlation between inbound and outbound traffic, a signature for the worm is generated in the form of a LCS and a LCSeq.

Output

It was suggested, that the LCS and LCSeq strings, which were generated during the SGM, could be used as signatures.

Evaluation

Through the use of LCSeq as signatures, some polymorphic worms can also be caught. LCSeq is robust against worm evasion efforts like insertion, deletion or reordering of bytes, which are not related to the worm code. But if bytes, which were part of the LCSeq signature, are reordered in a worm payload, then the LCSeq becomes useless. Both LCS and LCSeq have the potential to detect worms, which use payload encryption, but have an invariant encryption/decryption routine present in their payload.

4.9 Polygraph

Input: network traffic
Output:

- Conjunction signatures: set of tokens.
- Token-subsequence signatures: set of ordered tokens.
- Bayes signatures: set of tokens with assigned scores.

System/Algorithm based on [17].

Input

Polygraph monitors can be situated at the entrance link of a network, which connects the network to the rest of the Internet, or at an end host. In both cases, the observed network traffic forms the input to the SGM.

Signature Generation Mechanism

The combined use of three different signature classes is suggested in the Polygraph system, as none of these signature classes alone is suitable for all polymorphic worms. The basic idea behind Polygraph signatures is that some invariant bytes must be present in every instance of a polymorphic worm so that an attack can succeed. The input for the signature generation mechanism comes in form of network traffic at a network edge or host. A flow classifier reassembles the flows intended for the same port number and puts them in a suspicious flow pool respectively innocuous flow pool. Polygraph does not propose any concrete flow classifier; instead any classifier like a honeypot could be used for this purpose. Contiguous byte sequences longer than a specified minimum length are called *tokens*. Tokens, which occur at least in a certain number of flows in the suspicious pool, are collected. Afterwards, only the tokens in the flows are left, and the rest of the payload is removed so that finally each flow is represented as a sequence of tokens. The generation of *conjunction signatures*, *token-subsequence signatures* and *bayes signatures* is presented below.

Generation of Conjunction Signatures and Token-subsequence Signatures

A conjunction signature is a set of tokens. A payload matches such a signature, if it contains all tokens present in the signature in any order. Given a set of tokenized flows, a conjunction signature can be generated by simply extracting the tokens present in all flows.

A token-subsequence signature consists of a set of ordered tokens. A flow matches such a signature, if it contains the same set of tokens in the same order. Generating a token-subsequence signature for a given set of flows is analogue to the problem of finding the longest common subsequence of a set of strings. In Polygraph, an adaptation of the Smith-Waterman algorithm [19]was used, where contiguous token-subsequences are given a higher weight than separated token-subsequences.

In the suspicious flow pool there might be several different worm types and innocuous flows. Producing one signature for the whole suspicious flow pool would lead to a very unspecific signature, which would finally result in lots of false positives. So the next step is to generate a set of signatures, which together match the whole suspicious pool, but one signature matches only one worm and possibly its variants. To achieve this goal, hierarchical clustering [18] is used, in order to group similar flows into the same cluster. Then for each cluster, which contains a "sufficient" number of flows, a conjunction signature or token-subsequence signature is generated as described above.

Generation of Bayes Signature

A Bayes signature consists of a set of tokens with a score assigned to each of them. A flow is matched against this signature by adding up the scores of tokens, which are present in the flow. If the computed sum is greater than a threshold, the flow is classified as a worm.

To generate bayes signatures, the suspicious flow pool is tokenized as described before. Then for each of these tokens, the probability is computed, that the token is present in a worm. This probability is calculated as the fraction of flows in the suspicious pool, where the token is present. After that, the probability for each token is computed, that it appears in some innocuous traffic. This probability is estimated by the fraction of flows, where the token appears in the innocuous flow pool, and a technique described in [17]. These two probabilities for each token are inserted into a formula, which is derived form the Bayes law [17], in order to assign a score to each token.

Output

Three signature types are generated by the Polygraph system: conjunction signatures, token-subsequence signatures and bayes signatures. They all consist of byte sequences, called tokens. Conjunction signature is a set of such tokens. Token-subsequence signature consists of a set of ordered tokens. Bayes signatures consist of a set of tokens with a score assigned to each of them.

Evaluation

The three different classes of signatures presented in Polygraph are targeting different types of worm behaviour. Token-subsequence signatures are more specific than conjunction signatures, because of the ordering constraint. So conjunction signatures are more robust against reordering of bytes in the worm payload than token-subsequence signatures. Both conjunction and token-subsequence signatures are robust against deletion and insertion of bytes, which are not part of the tokens in these signatures. As bayes signatures do not look for exact matching, they are resilient to red herring attacks, where a worm initially includes some garbage tokens in its payload so that these tokens are incorporated into a signature. Then after some time, the worm stops including such garbage tokens. Such an attack would render the other two types of signatures useless in most cases. Polygraph’s basic idea is to find invariant tokens in different variants of a worm. Basically all three generated signatures are suitable to detect worms that use payload encryption and have an invariant encryption/decryption routine present in their payload.
It has been suggested by the authors that all three signatures should be used: Over time an evaluation of the signatures could be done to find out, which signature produces the least amount of false positives and false negatives. The signature producing the least amount of wrong classifications could then be further deployed.

4.10 StonyBrook

Overview

Input: Network traffic.
Output: Signature based on input field length and/or distribution of characters.
System/Algorithm based on [46][2].

Input

The StonyBrook system is built for monitoring a server application. As such it sends all inbound traffic for ports, which are used by the server application, to the SGM.

Signature Generation Mechanism

The suggested approach in this system aims to stop attacks, which are based on modification of return or function pointers such as in buffer overflow attacks. For each protocol type being used by a server application, a simplified specification has to be written for its message format. The specification is then compiled and a parser is generated for this message type with help of the Flex tool [48]. This parser produces fields according to the format specification of the input messages.

Address-space randomization (ASR) [47] is used, in order to randomly place different parts of the server program in the address space, such as executable code and shared libraries. If an attacker changes a pointer through a buffer overflow attack, the new pointer will point with a high probability to an invalid memory location. This will cause a memory exception, when the forged pointed is dereferenced. This memory exception triggers the execution of a signal handler. In this handler an analyzer is implemented. This analyzer first finds out, where the corrupted pointer is located in the memory. Then it tries to locate the attack string in the input by looking for the longest substring around the corrupted point in the address space, which matches some string in the input. All matching strings in the input are marked. For these candidate strings, the message type and the field they belong to in the parsed input, is determined. A signature can then incorporate attack characteristics, such as the minimal field size needed for an attack to succeed.

Output

The output of the SGM is a signature written in a language, which has been introduced in the StonyBrook paper [46]. The generated signature is based on fields, which have been defined in the specification of a message format. Furthermore, two main characteristics of an attack can be incorporated into a signature: field size and distribution of characters.
A modified example from the StonyBrook paper [46] is presented here. In the message format specification language, messages and sub messages are enclosed in curly braces. The first field in every message and sub message has a special name called type. Fields have an identifier and a value so that a field can be denoted as *identifier= value*. A SMTP message could have been parsed according to its specification into fields and values as follows.

```
{type="DATA", data={{type="From", email_from="joe@x.net"}
                    {type="To", email_to="john@y.com"}
                    {type="Body", body="Hello there!"}}
```

This is a message of type DATA, which consists of a field called data. The data field further consists of three sub messages. The first of these messages has type "From" and contains a field with identifier "email_from" and value "joe@x.net". A signature for an attack could look as follows.

[2]In [46], the authors did not give any name to the proposed algorithm; it will be called *StonyBrook* in this report.

```
{type = "DATA"; data = {{type = "From"; email_from.size > 200}
                        {type="To", && non-ASCII(email_to) > 0}}}
```

This signature says that an input message should be dropped, if its message type is "DATA", and it contains a field called "data", with two sub messages and the following properties: The first sub message has type "From" and a field called "email_from" with filed size greater than 200 bytes and a second sub message of type "To" and a field called "email_to", which contains some non-ASCII characters.

Evaluation

A new message format specification language has been proposed in [46], which allows to write short message specifications, as non-important details can be left out.

This approach is able to cope with polymorphic worms, as long as the payload is not encrypted. This means that byte reordering, deletion and insertion of bytes in the worm payload are not of much use, when the characteristics of the vulnerability, such as how large a field must be to overrun a certain buffer, is included in the signature. But the generated signature has problems with payloads on which encoding/decoding is performed by the server application, before the payload is copied to the vulnerable buffer. The reason for this issue is that the SGM is no more able to find the original worm payload of a request, if different encodings are used in the memory and the payload.

4.11 Dalhousie

Overview

Input: A database of legitimate and malicious software code.
Output: Profiles of legitimate and malicious software code represented by n-grams.
System/Algorithm based on [57][3].

Input

The input for the SGM presented in the Dalhousie paper is a database containing malicious and benign code already classified according to their true behaviour.

Signature Generation Mechanism

Motivated by the success of the common n-gram analysis (CNG) method [58], which is used to automatically find out the authors of some given text, an approach for generating signatures for unseen attacks is proposed. The CNG method is based on *n-grams*. Given a text, n-grams of this text are all substrings of this text of length n. The CNG method can create a profile for a given input by creating all possible input n-grams and computing their frequencies. Only the n-grams with the highest frequency are kept. These n-grams form a profile or signature for the given input.

The SGM receives a database of legitimate and malicious software code as input. It separately generates profiles for both sort of code using the CNG method, as described above. To find out, if some given code is benign or malicious, its profile is created using the CNG method. A formula is given, which can compute the distance between two profiles. This formula is used to measure the distance between the profile of the given code and the legitimate/malicious profiles. The given code is then classified as legitimate or malicious depending on the shortest distance to the profiles.

[3] In [57], the authors did not give any name to the proposed algorithm; it will be called *Dalhousie* in this report.

Output

The output of the SGM consists of profiles for legitimate and malicious software. Each of these profiles consists of a set of n-grams, i.e. strings of length n.

Evaluation

Today, it is not possible to generate profiles from all known malicious code, because of the huge amount of data available. One way to solve this problem is to make a selection out of the whole data, for instance only to consider the most current malicious code. Another issue is choosing the set of legitimate data used as input.

As the generated signature only looks for frequency of n-grams, it is able to deal with reordering of byte blocks. Furthermore, if n-grams are deleted from an attack payload, which is not part of the malicious profile, the generated signature will still be able to detect the attack. But if an attacker inserts a certain amount of such n-grams in the worm payload, which are part of the legitimate profile, then this approach could easily be fooled by an attacker.

4.12 PISA

Overview

Input: Packets seen at some network element.
Output: Pairs of identified fields and values and additional information such as from how many flows this signature was generated, how much bandwidth these flows consumed etc.
System/Algorithm based on [52].

Input

PISA monitors traffic at some backbone link and periodically takes some samples from the observed traffic and forwards them to the SGM.

Signature Generation Mechanism

The PISA algorithm inspects packets at some network element, in order to generate signatures for similar flows, which consume most part of the bandwidth. Instead of looking at all packets at the network element, only a sample of packets is taken periodically, from which network flows are generated. There are specified fields of interest (e.g. protocol type, IP-Address, TCP flag, port, etc.), which are extracted from the packets. Then hierarchical clustering [53] [54] is applied on a lattice, which consists of all distinct subsets of the given set of flows. As an exhaustive search in the lattice is expensive [56], a randomized approach [55] is used. The goal is to find subsets in the lattice, which have a minimum number of similar field and value pairs in common and do contain more flows than a given threshold. When such subsets of flows, called clusters, are found, the signature is generated by capturing similar field and value pairs and additional information, such as in how many samples the same field pattern was seen.

Output

The generated signature consists of fields and values. An example is given below.

```
{(packet_size, 100), (src_port, 80), (src_addr, 11.0.0.1),
(type, tcp), (tcp_flag, Syn ack), (dest_port, 7050)}
```

This signature matches flows originating from a host with IP address 11.0.0.1 and source port 80 to a host with destination port 7050, containing TCP SYN-ACK packets of size 100 bytes.

Together with a generated signature, also additional information is stored, such as the number of samples, flows and packets, in which the signature was observed.

Evaluation

The goal of the algorithm is to find such flows, which consume most of the bandwidth. It is very likely that signatures for legitimate traffic are also generated, when there is lots of similar legitimate traffic, which is often present in peer-to-peer context. This could cause lots of false positives.

PISA generates signatures, which are robust against polymorphic worms trying to obfuscate their payload by reordering/insertion/deletion of bytes or use perfect encryption on the whole payload. The reason for this robustness against polymorphism lies in the fact that PISA signatures only depend on packet header fields.

It is clear, that signatures which only depend on packet header fields will be less specific than those signatures, which also take some part of the worm payload into account. But also if too few fields are considered by the PISA signature, it might become too indistinctive, rendering lots of false positives.

Chapter 5

Classification of Automated Signature Generation Mechanisms

In the following sections classification criteria are defined for signature generation mechanisms (SGM) and the SGMs from Chapter 3 are classified according to these criteria.

5.1 System Location

When choosing a SGM for a system, it is important to know, where the signature generation system will be located. For example, a system which is located at the entrance of a network is suitable for system administrators who monitor a local area network. But companies that own large infrastructures of the internet, such as backbone links, might use a SGM which generates signatures only by monitoring network traffic at a router or backbone link. The amount of resources available plays a key role when choosing a SGM. A SGM relying on honeypot technology, for instance, may only be affordable for bigger institutions, such as governments.

Parameters

attacked-host means that the system generating the signature is placed at a host, which is connected to the internet. Therefore this host can potentially be attacked.

secure-host means, that the generation of the signature is performed on a host, which is assumed to be located at a secure place. This host does not need any connection to the internet. The main purpose of this host is to generate signatures.

network-entrance means, that the system for generating the signature is placed at the entrance link of a network. The host where this system is placed is neither a server, nor a client machine.

honeypot means, that one or many honeypots are part of the system, which generates the signature.

backbone means, that the system for generating the signature is placed at some network link with a large amount of traffic, such as a backbone link. The host where this system is placed is neither a server, nor a client machine.

5.2 Input

It is important to know what input a SGM requires in order to generate an attack signature. This criterion helps deciding, which SGM one to choose. For example, if the source code of software is required for the SGM, then this approach is not applicable in all contexts, as for commercial products, one has hardly ever access to the source code. When looking at the input, it is also possible to infer, how specific a generated signature is. For example, if an approach only takes a database of benign and malicious programs as input and does not consider any attack instances at all, the generated signature has to be general and unspecific.

Parameters

network-traffic means, that the SGM input consists of network traffic.

isolated-instance means, that the SGM receives an isolated attack instance. It was not specified in the related paper, how the attack was detected and isolated.

benign-db means, that the SGM input consists of a database, which contains benign software.

malicious-db means, that the SGM input consists of a database, which contains malicious software.

source-code means, that the system requires source code of the application, for which a signature against attacks is to be generated.

5.3 Signature Output Format

Knowing the output language and the output format of a SGM can help deciding, which IDS or SGM to use. For example a paper, which proposes a SGM that produces a signature, which is immediately deployable in an IDS, is more likely to be used by other projects, than one that does not generate a signature in an output format suitable for any existing IDS. There may be researchers, who want to use new languages for describing signatures in their system. For these people it could be interesting to know, which alternatives exist to the main-stream signature format languages.

Parameters

Word means, that the generated signature is less or equal than 4 bytes long.

String means, that the signature consists of one single string.

Strings means, that the signature may consist of one or several strings.

Communication-DFA means, that a deterministic finite state automaton (DFA) is created, which captures the communication between an attacker and a server. The transitions in the DFA consist of request/response strings.

Systemcall-DFA means, that a deterministic finite state automaton for a program is created, which captures the legitimate system call behaviour of the program in question.

Byte-score means, that the SGM output consists of bytes with assigned scores/weights.

String-score means, that the SGM output consists of strings with assigned scores/weights.

Field-length means, that the generated signature specifies a maximum length for application protocol layer fields.

Char-distrib means, that the ASCII character distribution of the packet content is considered in the signature, e.g. ASCII/non-ASCII characters.

Header-fields means, that TCP/IP header fields with their values are part of the signature.

Statistics means that also statistical information about the generated signature is stored along with the signature, such as in how many flows/packets the signature was seen during the signature generation process.

Bro means, that the SGM generates a signature suitable for the Bro [7] NIDS.

Snort means, that the SGM generates a signature suitable for the Snort [6] NIDS.

StonyBrook means, that the output is produced in a language, which is defined in the Stony-Brook paper [46].

A distinction is made between what characteristic is incorporated in the signature (e.g. string, header-field) and which output format is used for the generated signature (Bro, Snort, Stony-Brook). These distinct parts are separated by an arrow: *characteristic → output format*.

5.4 Worm Detection Mechanism

In this section, a classification of the SGMs is done according to the method used for detecting a worm. Researchers, who are looking for a SGM and are convinced that a method for detecting a worm's behaviour is superior to other detection methods, may find the classification presented in this section helpful. Others for instance, who believe that scanning behaviour will not be useful in future to detect worms, as hit-list worms are coming up, may want to know, which SGMs do not use *scanning* as a criterion for detecting worms. But as some SGM could also use other worm detection methods, one should not only rely on this criterion, when choosing a SGM.

Parameters

prevalence means, that the SGM takes advantage of the fact, that during worm spreading, the traffic related to a worm will be prevalent in the network traffic.

scanning means, that the SGM takes advantage of the scanning characteristic of worms, in order to detect the worm.

spreading means, that the algorithm detects a worm by exploiting its characteristic, where it continues to infect new hosts after having infected one host.

attack means, that the worm is detected when it is attacking a host, e.g. when a worm tries to overwrite memory locations.

anomaly means, that the worm is detected due to unusual traffic content, e.g. the byte distribution of monitored traffic deviates too much from a byte distribution of network traffic observed during training phase.

not-specified means, that the method for detecting the worm is not explicitly specified in the related paper.

5.5 Number of Attack Instances Required as Input

The number of instances, which are required for signature generation, is indeed an important criterion for the usefulness of a signature generation mechanism. The more instances of an attack are required, the more time is given to the attack for further spreading. So the most ideal SGM would not need any instance of the attack in question, such that a signature could be generated before the attack starts. This seems to be wishful thinking, but some SGMs have been proposed having this characteristic.

One thing to be noted here is, that only instances of new attacks are counted, which are required to generate signatures for the new attack in question. Therefore, even if a large database with "other" known attacks from the past is used during signature generation, it will not appear in the classification.

Parameters

None means, that no instance of the attack is needed by the SGM.

One means, that exactly one instance of the attack is needed for the SGM.

Several means, that more than one instance of the attack is required by the SGM.

5.6 Usage of Honeypot Technology

This classification criterion is important, particularly when looking at the fact that this work is part of a project, which is based on honeypot technology (see Chapter 1). But this criterion could also be useful for projects that are not using honeypot technology but relying on scanning behaviour of worms for worm detection.

Parameters

not-used means, that no honeypot was used or proposed for the SGM.

proposed means, that the employment of honeypot technology has been proposed for the SGM, but not used.

used means, that a honeypot was used for the SGM.

5.7 Usefulness Against Polymorphism

The degree of polymorphism that the generated signature can deal with is essential for the usability of the SGM, as generating variants of the same attack is much easier than finding new attacks. Most of the SGMs presented can deal with polymorphism. Knowing the degree of polymorphism that signatures can cope with is important, because it directly reflects the usefulness of a signature in practice.

Parameters

unqualified assumes that the attack content is not encrypted, and there exists a variant and an invariant (e.g. encryption/decryption routine) part in the attack content. But even for this simple case the generated signature is not able to detect any kind of polymorphism, such as the modification of variant parts of the attack content.

variant-modification assumes that the attack content consists of a variant and invariant part. If only the variant part of the attack content is modified by insertion, deletion or modification of bytes, then it will be detected by the generated signature.

invariant-modification assumes that the attack content consists of a variant and an invariant part. If such parts of the attack, which were recognized as invariant by the SGM and incorporated into the signature, change over time, the signature is still able to detect the attack content.

reordering means that some byte blocks of the attack payload may be reordered, but the generated signature will still be able to detect the reordered attack payload.

perfect-encryption means, that even in case of perfect encryption, where no information can be gained from the attack content, the generated signature is still able to detect the attack. Perfect-encryption is considered as the strongest possible polymorphism technique, which includes all parameters mentioned above.

5.8 Quality of Generated Signature

Assume an oracle, which can classify every possible input as legitimate or malicious, without any misclassifications. This oracle has the best possible signature deployed in it. The more a signature's classification defers from the one of the oracle's, while comparing the same input, the poorer it's quality is.

The purpose of a signature is to have the ability to distinguish legitimate input from malicious one. And in the end this is the most important goal. But as the generated signatures are not perfect, different importance is given to different misclassifications. For some systems, false-alarms, also called false-positives, are considered more harmful, as a user would not use an IDS [16], where a lot of benign traffic is misclassified. Other people require the opposite: No malicious traffic should be misclassified as legitimate. They want a low false-negative rate. But there often exists a trade-off between false-positives and false-negatives.

To present a fair evaluation of the quality of generated signatures, all signatures should classify a representative test-set consisting of malicious and legitimate input. As the evaluation in the proposed papers is done on different input of different size, it would be inappropriate to present the results of those evaluations here. The quality of a signature is important and an evaluation should be done in this area in the future, in order to get ahead towards a complete evaluation of today's SGMs.

Classification

In the tables 5.1, 5.2 and 5.3 a classification of all signature generation mechanisms from chapter 4 is given.

-	Nemean	Dynamic Taint Analysis	Honeycomb	IBM-94
system location (5.1)	honeypot and secure-host	attacked-host	honeypot	secure-host
input (5.2)	network-traffic	network-traffic	network-traffic	isolated-instance and benign-db
output format (5.3)	communication-DFA $\rightarrow$Bro[a]	word	header-fields, string $\rightarrow$ Bro, Snort	strings
detection mechanism (5.4)	scanning	attack	scanning	not-specified
No. of attack instances (5.5)	several	one	several	one
usage of honeypots (5.6)	used	proposed	used	not-used
polymorphism (5.7)	variant-modification	perfect-encryption	unqualified	variant-modification, reordering

Table 5.1: Classification of the automatic signature mechanisms

[a]Nemean produces *Regular Expressions* which can be used in Bro through Bro policy scripts.

-	Autograph	Paid	PADS	PAYL
system location (5.1)	network-entrance	attacked-host	honeypot	attacked-host
input (5.2)	network-traffic	source-code	network-traffic	network-traffic
output format (5.3)	strings → Bro	systemcall-DFA	byte-score	strings
detection mechanism (5.4)	scanning	attack	scanning and spreading	spreading and anomaly
No. of attack instances (5.5)	several	none	several	one
usage of honeypots (5.6)	proposed	not-used	used	not-used
polymorphism (5.7)	variant-modification, reordering	perfect-encryption	variant-modification, invariant-modification, reordering	variant-modification

Table 5.2: Classification of the automatic signature mechanisms

-	**Polygraph**	**StonyBrook**	**Dalhousie**	**PISA**
system location (5.1)	network-entrance or attacked-host	attacked-host	attacked-host	backbone
input (5.2)	network-traffic	network-traffic	benign-db and malicious-db	network-traffic
output format (5.3)	strings and string-score	field-length, char-distrib $\rightarrow$ StonyBrook	score-string	header-fields, statistics
detection mechanism (5.4)	not-specified[a]	attack	anomaly	prevalence
no. attack instances (5.5)	several	one[b]	none	several
usage of honeypots (5.6)	proposed	not-used	not-used	not-used
polymorphism (5.7)	*token-subsequence signature:* variant-modification *conjunction signature:* variant-modification, reordering *bayes signature:* variant-modification, invariant-modification	variant-modification, invariant-modification	reordering	perfect-encryption

Table 5.3: Classification of the automatic signature mechanisms

[a] Polygraph did not choose any method for detecting a worm. Instead the authors of Polygraph pointed out, that any flow classifier can be used as a input to their signature generation algorithm.
[b] There exists a small probability, that this SGM needs more than one attack instance.

Chapter 6

Conclusion and Outlook

6.1 Conclusion

A survey in the area of automatic attack signature generation was conducted, in order to provide a starting point for future research. This survey had to face similar challenges and difficulties, as most surveys, which are done for the first time in any research area. It was not possible to find an exiting survey on this topic, although some information might be present under different label in the literature. Key concepts were highlighted and irrelevant information was sorted out from the gathered papers. New criteria for classifying the most relevant papers were defined and a classification of the papers in question was made according to those criteria.

The survey identified fields of problems, such as how the quality of produced signatures could be measured in a standardized way, which is not possible currently. Researchers of new automatic attack signature generation methods can use this survey to find out key properties of existing methods, with which their emerging systems will have to compete. The survey has laid a base for future work in the area of automated attack signature generation and related research fields.

6.2 Outlook

Surveys need to be updated as time passes. New methods for automatic attack signature generation have to be included in a future survey, as they arise. New criteria may be identified for new signature generation methods. Furthermore, classification criteria identified in this survey might require adaptation, as new discoveries are made. Some identified criteria in this work might become obsolete, as new types of worms arise. Future surveys in the area of automatic attack signature generation are provided with an initial survey here.

As mentioned in chapter 5, an evaluation framework could be built in future, to measure the quality of the signatures produced by the different SGMs. This is important, as the final purpose of every signature is to classify an input with as less misclassifications as possible.

Bibliography

[1] C. Kreibich and J. Crowcroft. *Honeycomb - creating intrusion detection signatures using honeypots*. In Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II), November 2003.

[2] N. Provos, *Honeyd - A Virtual Honeypot Daemon*, in 10th DFN-CERT Workshop, Hamburg, Germany, February 2003.

[3] E. Ukkonen, *On-line construction of suffix trees*, Algorithmica, no. 14, pp. 249-260, 1995.

[4] N. Provos. *A Virtual Honeypot Framework*. In Proceedings of the 13th USENIX Security Symposium, pages 1.14, August 2004.

[5] R.G. Bace, *Intrusion Detection*, Macmillan Technical Publishing, Indianapolis, USA, 2000.

[6] M. Roesch, *Snort: Lightweight Intrusion Detection for Networks*, in Proceedings of the 13th Conference on Systems Administration, 1999, pp. 229-238.

[7] V. Paxson, *Bro: A System for Detecting Network Intruders in Real-Time*, Computer Networks (Amsterdam, Netherlands: 1999), vol. 31, no. 23-24, pp. 2435-2463, 1998. [Online]. Available: http://citeseer.nj.nec.com/article/paxson98bro.html

[8] S. McCanne, C. Leres and V. Jacobson, libpcap, available via anonymous ftp to ftp.ee.lbl.gov, 1994.

[9] M. Erbschloe. Computer Economics VP Research Statement to Reuters News Service, Nov. 2001.

[10] D. Moore, C. Shannon, G. Voelker, and S. Savage. *Network Telescopes*. Technical Report CS2004-0795, CSE Department, UCSD, July 2004.

[11] J. Newsome and D. Dong. *Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software*. In The 12th Annual Network and Distributed System Security Symposium, February 2005.

[12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. *Point- Guard: Protecting pointers from buffer overflow vulnerabilities*. In 12th USENIX Security Symposium, 2003.

[13] N. Nethercote and J. Seward. *Valgrind: A program supervision framework*. In Proceedings of the Third Workshop on Runtime Verification (RV'03), Boulder, Colorado, USA, July 2003.

[14] V. Kiriansky, D. Bruening, and S. Amarasinghe. *Secure execution via program shepherding*. In Proceedings of the 11th USENIX Security Symposium, August 2002.

[15] P. Szor. *Hunting for metamorphic*. In Virus Bulletin Conference, 2001.

[16] S. Axelsson. *Intrusion detection systems: A survey and taxonomy*. Technical Report 99-15, Department of Computer Engineering, Chalmers University, March 2000.

[17] 12] J. Newsome, B. Karp, D. Song, *Polygraph: automatically generating signatures for polymorphic worms*, Proceedings of the IEEE sysmposium on security and privacy, 2005.

[18] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[19] T. Smith and M.Waterman. *Identification of common molecular subsequences*. Journal of Molecular Biology, 147:195-197, 1981.

[20] Yong Tang, Shigang Chen, *Defending Against Internet Worms: A Signature-Based Approach*, in Proc. of IEEE INFOCOM'05, Miami, Florida, USA, May 2005.

[21] C. E. Lawrence and A. A. Reilly, *An Expectation Maximization (EM) Algorithm for the Identification and Characterization of Common Sites in Unaligned Biopolymer Sequences*, PROTEINS: Structure, Function and Genetics, vol. 7, pp. 41-51, 1990

[22] C. E. Lawrence, S. F. Altschul, M. S. Boguski, J. S. Liu, A. F. Neuwald, and J. C. Wootton, *Detecting Subtle Sequence Signals: A Gibbs Sampling Strategy for Multiple Alignment*, Science, vol. 262, pp. 208-214, Oct. 1993.

[23] S. Geman and D. Geman, *Stochastic Relaxation, Gibbs Distribution, and the Bayesian Restoration of Images*, IEEE Trans. Pattern Anal. Machine Intell., vol. 6, pp. 721-741, 1984.

[24] Giovanni Vigna and Richard A. Kemmerer. *NetSTAT: A Network-based Intrusion Detection System*. In 14th Annual Computer Security Applications Conference, December 1998.

[25] Laurent Eschenauer. Imsafe. http://imsafe.sourceforge.net, 2001

Parminder Chhabra, Ajita John, Huzur Saran: *PISA: Automatic Extraction of Traffic Signatures*. NETWORKING 2005: 730-742

[26] CERT, *Nimda Worm*. CERT Advisory CA- 2001-26, Sept, 2001.

[27] Paul Boutin, *Slammed! An inside view of the worm that crashed the Internet in 15 minutes*. http://www.wired.com/wired/archive/11.07/slammer.html. July 2003.

[28] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford and N. Weaver, *The Spread of the Sapphire/Slammer Worm*. Technical report, Feb 2003, http://www.cs.berkeley.edu/ nweaver/sapphire/

[29] Hyang-Ah Kim, Brad Karp, *Autograph: Toward Automated, Distributed Worm Signature Detection*, USENIX Security Symposium, to appear, 2004.

[30] LEMOS, R. *Counting the Cost of Slammer*. CNET news.com. http: //news.com.com/2100-1001-982955.html, Jan. 2003.

[31] MUTHITACHAROEN, A., CHEN, B., AND MAZI 'E RES, D. *A Lowbandwidth Network File System*. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001) (Oct. 2001).

[32] RABIN, M. O. *Fingerprinting by Random Polynomials*. Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[33] White Paper, *The Science of Intrusion Detection System: Attack Identification*, Cisco Systems, Inc.
Available from http://www.cisco.com/warp/public/cc/pd/sqsw/sqidsz/prodlit/idssa_wp.htm

[34] Sundaram, A. 1996. *An introduction to intrusion detection*. ACM Crossroads-Special Issue on Computer Security, 2(4). Available from http://www.acm.org/crossroads/xrds2-4/intrus.html

[35] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. Ph.D. Dissertation, August 1995.

[36] Henry S Teng, Kaihu Chen and Stephen C Lu. *Security Audit Trail Analysis Using Inductively Generated Predictive Rules*. In Proceedings of the 11th National Conference on Artificial Intelligence Applications, pages 24-29, IEEE, IEEE Service Center, Piscataway, NJ, March 1990.

[37] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantic-aware signatures. Technical Report 1507, University of Wiscsonsin, 2004. http://www.cs.wisc.edu/˜vinod/nemean-tr.pdf

[38] Javed Aslam, Katya Pelekhov, and Daniela Rus. *A practical clustering algorithm for static and dynamic information organization*. In ACM-SIAM Symposium on Discrete Algorithms (SODA), Baltimore, Maryland, January 1999.

[39] The Honeynet project. http://project.honeynet.org, April 2004.

[40] Jon Patrick, Anand Raman, and P. Andreae. *A Beam Search Algorithm for PFSA Inference*, pages 121-129. Springer-Verlag London Ltd, 1 edition, 1998.

[41] Thomas Ptacek and Timothy Newsham. *Insertion, evasion and denial of service: Eluding network intrusion detection*. Technical report, Secure Networks, January 1998.

[42] Anand V. Raman and Jon D. Patrick. *The sk-strings method for inferring PFSA*. In 14th International Conference on Machine Learning (ICML97), Nashville, Tennessee, July 1997.

[43] K. Wang, G. Cretu, S. Stolfo. *Anomalous Payload-based Worm Detection and Signature Generation*, submitted to Usenix Security 2005

[44] M. Damashek. *Gauging similarity with n-grams: language independent categorization of text*. Science, 267(5199):843–848, 1995

[45] K. Wang and S. Stolfo. *Anomalous payload-based network intrusion detection*, in Proceedings of Recent Advance in Intrusion Detection (RAID), Sept. 2004.

[46] Zhenkai Liang and R. Sekar. *Automated, Sub-second Attack Signature Generation: A Basis for Building Self-Protecting Servers*. To appear in 12th ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, November 2005

[47] The pax team. http://pax.grsecurity.net.

[48] http://www.gnu.org/software/flex/

[49] Lap-Chung Lam, Tzi-cker Chiueh. *Automatic Extraction of Accurate Application-Specific Sandboxing Policy*. RAID 2004: 1-20

[50] CERT Corrdingation Center. Cert summary cs-2003-01. http://www.cert.org/summaries/, 2003.

[51] D.Wagner and P. Soto. *Mimicry attacks on host-based intrusion detection systems*. In Proceedings of the 9th ACM Conference on Computer and Communications Security, November 2002.

[52] Parminder Chhabra, Ajita John, Huzur Saran: *PISA: Automatic Extraction of Traffic Signatures*. NETWORKING 2005: 730-742

[53] R. O. Duda and P. E. Hard. *Pattern Classification and Scene Analysis*. Wiley-Interscience, NY, 1973.

[54] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, New Jersey, 1988.

[55] H. V. Jagadish, J. Madar, and R.T. Ng, *Semantic Compression and Pattern Extraction with Fascicles*, Proceedings of 25th VLDB, pp. 186-198, 1999

[56] H. Mannila and H. Toivonen. *Level Wise search and borders of theories in knowledge discovery*, Data Mining and Knowledge Discovery, 1,3, pp 241-258.

[57] Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan. *Detection of New Malicious Code Using N-gram Signatures*. In Proceedings of the Second Annual Conference on Privacy, Security, and Trust (PST'04), Fredericton, New Brunswick, Canada, October 2004.

[58] V. Keselj, F. Peng, N. Cercone, and C. Thomas. 2003. *N-gram-based Author Profiles for Authorship Attribution.* In Proceedings of the Conference Pacific Association for Computational Linguistics, PACLING'03, Dalhousie University, Halifax, Nova Scotia, Canada.

[59] Konstantin Rozinov, *Reverse Code Engineering: An In-Depth Analysis of the Bagle Virus*, 6th Annual IEEE Information Assurance Workshop, United States Military Academy, West Point, NY, June 2005. http://rozinov.sfs.poly.edu/papers/bagle_analysis_v.1.0.pdf

[60] J.O. Kephart and W.C. Arnold. *Automatic extraction of computer virus signatures*. In Proceedings of the Fourth International Virus Bulletin Conference, 179-194. Virus Bulletin Ltd., 1994.

[61] http://www.fp6-noah.org/

[62] Patrick Diebold, Andreas Hess, Günter Schäfer, *A Honeypot Architecture for Detecting and Analyzing Unknown Network Attacks*. KiVS 2005: 245-255

[63] S. Staniford, V. Paxson, and N. Weaver, *How to 0wn the Internet in Your Spare Time*, in Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, Aug. 2002.

[64] http://www-nrg.ee.lbl.gov/bro-info.html, June 2005

[65] http://www.clark.net/˜roesch/