Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Nicoletta De Maio

An Implementation of a Byte Frequency Signature Generator Using a Gibbs Sampler

Semester Thesis SA-2006-21
25th November 2006

Tutor: Bernhard Tellenbach
Co-Tutor: Daniela Brauckhoff
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

Traditional pattern-based signatures to identify worms and similar attacks fail when confronted with polymorphic code segments. Developing new types of signatures capable of recognising malicious payload while at the same time allowing for a certain degree of variation within its structure is therefore a large and fast-moving field of research. There are several approaches to this problem, each tackling it from a slightly different angle, but they typically involve the use of byte frequency distributions instead of the more established substrings as signatures. The algorithms used to extract these byte frequency distributions from captured malicious payload also vary depending on the priorities assigned to the various subproblems within the larger task. This project implements one of these signature generation algorithms and tests some of its properties, with the ultimate goal being an evaluation of its usefulness within the NoAH project and Argos. It also proposes a subsystem that could collect and process captured worms in the same way as the SweetBait subsystem within the Argos honeypot architecture.

Contents

List of Figures

Chapter 1

Introduction

Honeypot systems play an important role in capturing malicious network traffic for further analysis. Traditionally, this analysis is done by an expert in network security, resulting in classification of the analysed code segments and a virus or worm signature for easy identification.

As this method is relatively slow, self-propagating worms tend to infect a multidude of systems before a signature protecting capable of identifying a particular worm is ready to be deployed. By this point, the worm has often caused considerable damage to the infected systems, reducing the benefits from using the signature.

A more efficient approach - as well as an important current research topic - consists of automatically generating the required signatures out of the captured malicious code fragments. Argos[1], developed at the Vrije Universiteit Amsterdam in the Netherlands, is an implementation of such an approach. Its unique system of labelling memory with content from suspicious sources allows it to separate true attacks from packets sent to the honeypot as part of legitimate traffic, or as a result of misconfiguration. It does so in a simple and effective way. It first labels all memory content originating from network traffic as potentially harmful and, at the same time, keeps track of its origin. Whenever such suspicious memory content is executed, it raises an alert. As such, it can decide with near-certainty whether a code fragment is a candidate for further processing or whether it can safely be ignored.

In the case of a TCP connection, Argos itself reconstructs the network flow involved in delivering the attack to the system, and derives a first signature from it. The SweetBait subsystem collects such signatures from several systems and refines them into a fingerprint for the given attack. As in the usual case, this fingerprint is extracted by looking for longest common substrings within the byte strings that make up the worm instances.

Detection using longest common substrings of the byte string that makes up the code runs into problems when it comes to polymorphic worms. If the worm instances differ enough to make the longest common substring very short, it may no longer be possible to distinguish between attacks and legitimate traffic that happens to contain the same substring. An unacceptably large number of false positives and spurious alerts is the result, making the intrusion detection system less useful.

An alternative to longest common substrings are byte distribution signatures. Rather than look for identical stretches within a given byte string and a refernce value, they measure the overall similarity between two byte strings. If the similarity score is above a certain threshold, and the reference value was that of a known worm, the new code fragment is classified as such and an alert is generated. Along the same lines, if the similarity is below a certain threshold and the reference value was that of normal network traffic, the code fragment is also classified as anomalous.

These two methods of comparison are largely identical. The main difference is that comparing something to an anomaly requires knowledge of said anomaly beforehand, while

comparing something to a model of normality allows a decision independent of prior knowledge of a particular anomaly. The latter is obviously more useful when it comes to react to attacks as they happen instead of after they have happened (to a number of others, hopefully), so it is the approach taken in the paper that was selected as the basis for this semester project[3].

The goal of the project was to become acquainted with the topic of byte frequency distribution signatures and to choose one of the presented methods of generating such a signature. This method was then implemented to use malicious code captured by Argos to create and store the resulting worm fingerprint in a database for redistribution. Its purpose was to extend the existing framework to include this type of fingerprint in addition to the longest common substring signatures already provided.

The rest of this project report is organised according to the following structure: Chapter 2 explains the theory behind the signature generation method used and presents some related work. Chapter 3 describes the framework developed during the implementation of the algorithm, while Chapter 4 gives an evaluation of the generated signatures. Chapter 5 gives a summary of the project as a whole. Some ideas that might merit further thought are discussed in Chapter 6.

Chapter 2

Motivation and Related Work

In a substring-based signature, certain byte within the worm code either has a certain value or it has not. Contrary to this, a byte frequency distribution for the same worm does not state if a certain value is present or not, but gives an indiciation of how probable it is to find a value at a given position within the same byte string.

One of the main advantages of signatures of this kind lies hidden in this fact. Because they do not limit themselves to looking for substrings, they will not fail automatically if only part, but not all, of a given substring is found. Looking at the overall picture rather than part of it makes them far more successful at detecting polymorphic worms than the traditional approach, as polymorphic worms tend to shift and shuffle their code around within longer stretches of meaningless filler payload to avoid detection by string matching methods.

The higher the probability of a byte, the greater the similarity to the reference distribution for that position, be it of regular network traffic or of an anomaly, if that byte is encountered on a captured byte string. If the values for all or part of the byte string are taken into account in some creative fashion, we obtain an overall score that tells us how much the code fragment deviates from the norm.

The signature generated from a number of instances of the same worm is essentially a representative value for this deviation. There are several approaches to generating signatures from byte distributions of the captured worm code. They differ in areas such as complexity, computational efficiency, accuracy of the generated signature with respect to false positives and type of parameters taken into account for signature generation. What they all have in common is the possibility to be refined over time, as new worm instances are collected and analysed automatically.

For this work, two of these different approaches were considered for implementation to evaluate polymorphic worms captured by Argos. The following is an overview of the properties and characteristics of each of these.

2.1 Z-String Signatures

Ke Wang and Salvatore Stolfo propose an intrusion detection system that organises the byte frequency distribution of a network payload in the form of a Z-string. Ordering the byte frequency distribution according to the number of occurrences of each byte, most common byte first, a characteristic sequence of characters for the distribution is obtained. These *Z-strings* describe the byte frequency distributions for each port and payload length in an easily recognisable way, as they tend to have a distinctive shape depending on the service offered on a given port. This signature is then used for comparison with observed network traffic to determine a score on a scale between normal and anomalous[2].

Because the profile of a byte frequency histogram varies depending on the network ser-

vice, distributions for regular network traffic are generated separately for each service simulated or run on a given system. Along the same lines, potentially malicious traffic is examined per port and compared to the corresponding distribution only. Of a given network packet, only the payload is considered while information contained in the header is discarded.

As far as intrusion detection systems (IDS) go, this one models normality. Byte frequency distributions for each combination of port and message length, so-called centroids, are initially computed during a training phase using attack-free test data. Once they are in active use, they update themselves to take each incoming payload into account, as long as said payload has been classified as normal by the IDS.

This approach obviously results in a large number of centroids, which can cause problems in two ways. First, the overhead associated with storing and maintaining such a large data set can quickly become very large, and second, there may simply not be enough data for a given centroid to result in a meaningful model during the training phase. Therefore, if two centroids of a length difference of 1 are sufficiently similar, they are merged. This process is repeated until several clusters of a certain dissimilarity have been obtained.

To compare an incoming payload to its corresponding centroid (or two centroids to each other), a simplified form of the *Mahalanobis distance* is used. The distance between a centroid $\overline{y}$ and a payload x is calculated according to $d(x,\overline{y}) = \sum_{i=0}^{n-1}(|x_i - \overline{y}_i|/(\overline{\sigma}_i + \alpha))$, where $\overline{\sigma_i}$ is the standard deviation for byte i and α is the smoothing factor for this centroid. α reflects the statistical confidence of the sampled training data: the larger the value of α, the less the confidence that the samples accurately represent the actual distribution. Thus, the byte distribution can be more variable.

As the standard deviations may change because of newly acquired information, it makes sense to keep track of those values as well. Because they are the square roots of the variances, and the variance of a distribution can be expressed as $Var(X) = E(X^2) - (EX)^2$, the only extra values that need to be maintained are the averages of the x_i^2, in a similar vector as $\overline{x}$.

It is apparent that with this system of distance calculation, the comparison of an incoming packet to its corresponding centroid is linear in the length of the packet, i.e. in the number of bytes it contains. So are the update procedures for $\overline{x}$ and $\overline{x^2}$, which makes the whole intrusion detection method very efficient and suitable for application in real time.

Incoming payload can also be described using Z-strings. If the contents of the packet are deemed to be too far away from the desired normal distribution, its Z-string may serve as a simple worm signature that can be deployed to other IDS systems quickly as soon as the anomaly first occurs, thus slowing or even preventing new attacks early on.

2.2 Position-Aware Byte Frequency Distribution

Yong Tang and Shigang Chen take a different approach[3]. Instead of modelling normality in relatively fine grains, they propose the use of a single distribution for all legitimate network traffic but a more sophisticated type of signature to distinguish anomalies.

In order to capture worms automatically, the paper proposes a double honeypot system. Two honeypots with different configurations are set up to capture and extract malicious traffic. The *inbound* honeypot simulates one or more network services and accepts connections from the outside to attract attackers. Once it has been compromised, it will attempt to open connections to other systems to try and infect them as well.

Because of this behaviour, the inbound honeypot is configured in a way that all outgoing connections are rerouted to the *outbound* honeypot. This outbound honeypot is low-interaction.

It accepts connections but only simulates reactions without offering actual services.

Separation of normal and anomalous traffic happens in two places. One is the inbound honeypot as described above: while incoming traffic is considered to be suspicious, the act of attempting to connect to other systems changes the classification to 'dangerous' and separates it from traffic that ended up in the honeypot because of errors and misconfigurations or unsuccessful attacks. A much larger part of the network traffic is siphoned off by the network gateway before it ever reaches either honeypot, though.

In a typical network, only IP addresses of public servers are visible to the outside world. The ratio of those public addresses to those potentially located within is usually in favour of the inside systems, i.e. the number of public IP addresses is small compared to the total available address range within the network. So when an attacker tries to connect to random addresses within that network, his chances of picking one that has not been published to the outside are quite high. If the router that acts as the gateway to the network is configured to redirect all of this 'randomised' traffic to the inbound honeypot, it is possible to distinguish between mostly harmless connections and fairly suspicious ones in a simple and efficient way.

Once the anomalous payloads have been identified using a double honeypot or another system, they are collected and labelled. Because the ultimate purpose is to extract a signature from captured worms, any anomalous payload is considered a worm from now on. The label serves as a means to identify the type of the anomaly, i.e. the worm.

The proposed signature generation method is that of a *Position-Aware Distribution Signature (PADS)*. It has all the advantages of ordinary byte distribution signatures, namely being able to handle limited amount of polymorphism fairly well. On the other hand, it is also designed to work if the worm is embedded in chunks of normal traffic, an area where traditional substring signatures tend to be better than byte distributions. This is achieved by two main techniques.

First of all, the signature generator does not look at the whole payload, as described in Section 2.1, but only at a region of a given width W where the deviation from the norm is largest. Second, instead of calculating the most likely *byte value* for each position in this region, it creates a complete *probability distribution* that describes how likely it is to find each of the possible values in this specific position. This set of W byte frequency distributions, combined with the byte frequency distribution for normal network traffic, is what defines a Position-Aware Distribution Signature.

If f_0 contains the probabilities for each byte according to the norm (the *normal signature*) and f_1 to f_W the probability distributions for each position of the significant region (the *abnormal signature*, the PADS signature $\Theta = (f_0, f_1, \ldots, f_W)$ is calculated from a set of captured worms $S = S_1, \ldots, S_W$ by counting, for a given position within the significant region, the number of occurrences of each byte at that position within the captured worms, and dividing the resutlting values by the number of worms to scale them down to probabilities. In order to avoid zero values in those probabilities, a pseudo count of d is employed, so that the final estimation for a byte of value x at position pos is given by $f_{pos}(x) = \frac{c_{pos,x}+d}{n+256 \cdot d}$, with $c_{pos,x}$ being the count of bytes of value x encountered at position pos.

To compare the significant region of a worm S_i to a signature Θ, the matching score of the aforementioned region with the anomalous signature is defined as

$$M(\Theta, S_i, s\_pos) = \prod_{pos=1}^{W} f_{pos}(S_i[s\_pos + pos - 1])$$

Along the same lines, the comparison to the norm is done by computing

$$\overline{M}(\Theta, S_i, s\_pos) = \prod_{pos=1}^{W} f_0(S_i[s\_pos + pos - 1])$$

with s_pos being the starting point of the significant region of S_i in both cases.

To measure the quality of the combination of M and $\overline{M}$, which should maximise M but minimise $\overline{M}$, the ratio between the two values is defined as the *matching score* of the *significant region of* S_i with the complete *PADS signature* Θ is defined as

$$\Lambda(\Theta, S_i, s\_pos) = \frac{M(\Theta, S_i, s\_pos)}{\overline{M}(\Theta, S_i, s\_pos)} = \prod_{pos=1}^{W} \frac{f_{pos}(S_i[s\_pos + pos - 1])}{f_0(S_i[s\_pos + pos - 1])}$$

with starting position s_pos that maximises Λ.

Using the logarithm of this score makes it easier to plot the results. Therefore, the *final matching score* is defined as

$$\Omega(\Theta, S_i) = \max_{s\_pos=1}^{length(S_i)-W+1} \frac{1}{W} log(\Lambda(\Theta, S_i, s\_pos))$$

The major remaining problem lies in finding out where exactly the significant region that maximises the final score is located. The paper suggests and tests two algorithms for this task: Expectation Maximisation and Gibbs Sampling.

The Expectation Maximisation algorithm takes a set S of byte sequences (i.e. captured worms) as input. Both the optimal starting positions s_pos_1 to s_pos_n and the signature Θ are unknown.

For a large initial value of W, the algorithm initialises the starting positions for each worm variant S_i randomly and uses this initial guess to computes a first estimate of Θ. Taking this signature as a starting point, it recalculates the starting positions as those that maximise the score when compared to Θ. In other words, the starting position s_pos_i that yields a maximal value for $\Lambda(\Theta, S_i, s\_pos_i)$ is selected for each worm.

Next, the algorithm goes back to the signature and computes a new estimate for Θ based on the new starting positions. It will continue to go back and forth between the two sets of unknowns until convergence. The whole process can now be repeated for smaller values of W until the best combination of Θ, W and the s_pos_i has been found.

One of the main problems of Expectation Maximisation is that once it has reached a local maximum, it will never leave the region again and the truly optimal solution will not be found. Gibbs Sampling offers a solution to this problem: because of a random element when selecting parameters later in the process, it has a small chance of jumping out of a local maximum by moving in a direction that at first glance seems to be worse than what has been achieved so far.

In this particular case, the sampler is also initialised by assigning random starting positions to the worm instances. One of the worms S_x is then chosen at random, while the estimate for the signature Θ is calculated based on the remaining $S - S_x$ worms in the set. An average matching score Ω can now be determined by comparing every possible starting position $s\_pos_x \in [1..length(S_x) - W + 1]$ to the current value of Θ as described in the definition of Ω.

Before adding S_x back to the pool and selecting another worm to repeat the process, its starting position is updated by selecting a random value according to the probability

$$Pr(s\_pos_x) = \frac{\Lambda(\Theta, S_x, s\_pos_x)}{\sum_{s\_pos_x=1}^{length(S_x)-W+1} \Lambda(\Theta, S_x, s\_pos_x)}$$

which is proportional to Λ and will in all likelyhood move s_pos_x further towards the optimal starting position.

This process of selecting a worm to be excluded, generating a signature from the rest, and updating the starting position of the odd one out is repeated until the scores computed

during two consecutive steps of the algorithm meet the convergence criteria of being no more than ϵ apart from each other, or a pre-specified maximum number of iterations is reached. The signature Θ associated with the last computed score is then taken to be the optimal value.

The evaluation of the authors of the paper showed consistently better results for the Gibbs Sampler than for Expectation Maximisation. Not only does the achieved score tend to be higher, but the scores also fluctuate a lot less for different values of W and varying lengths of instances of the same worm. It therefore seems to be the more reliable signature generation method of the two.

2.3 Method Evaluation

Each of the methods described in the previous two sections has its advantages and disadvantages. Z-strings are simple, fast to compute and surprisingly effective. The approach of separating traffic by port automatically sorts any potential attacks into a kind of category, where they can be compared to a centroid representing that specific category. This allows quite fine-grained distinctions based on the type of service that is being targeted: what may be normal for one, may raise an alarm for another. On the other hand, the simplicity of the Z-strings is also responsible for more than a few false positives. The administration and memory overhead associated with creating and maintaining the centroids that model normal traffic can become quite cumbersome as well.

PADS signatures are more difficult to compute and their generation is more time-consuming, but they tend to be more accurate at distinguishing anomalous traffic from the background. Particularly those obtained with the Gibbs sampling algorithm. By looking at a characteristic region instead of the whole worm, they tend to be able to handle embedded malicious code containing some polymorphism. The administrative overhead, aside from the signature generation itself, is small. The reason for this is that it consists of training and maintaining a single probability distribution that models all normal, attack-free network traffic. A major drawback of the paper that proposes this method is that it suggests that the captured worms should be assigned a type of some sort, but never gives any indications on how this should be accomplished.

Because this work is primarily concerned with signature generation, the PADS signature calculated by the Gibbs sampling algorithm, seems to be the most promising. It can be safely assumed that assigning a type to worm instances will happen elsewhere, if at all. Gibbs sampling in combination with position-aware distributions seem to be a flexible and reliable, if somewhat slow, solution that produces results of a better quality than those obtained with Z-strings. The overhead not concerned with the actual signature generation, i.e. the modelling of 'normality' is also drastically reduced. So the method of choice to be tested to see if the results published in the paper can be reproduced under slightly different circumstances will be Position-Aware Distribution Signatures.

Chapter 3

Design and Implementation

The SweetBait subsystem described in [1] receives pattern-based preliminary signatures from several systems running Argos. It then attempts to combine those signatures into a single more refined one that is saved for redistribution. It makes sense that a new subsystem dealing with the same kind of input but producing a different type of signature should operate along the same lines: recieve the relevant data from somewhere else, generate a signature from it, and use it to refine an existing signature for the given type of attack if one can be found in the database. This is the idea implemented in the BowNet package.

BowNet consists, at its core, of three files: the programs *WormColl* and *SigGen*, and the input file *NormalTraffic.dist* containing the byte frequency distribution *f0* to compare the significant worm regions to. In order to compile properly, SigGen requires the MySQL client as well as the developer package for that client. For BowNet to be able to operate, the MySQL server must be running.

3.1 BowNet Overview

In order to generate signatures like those described in Chapter 2.2, an application consisting of two main parts and a MySQL database was developed. As shown in Figure 3.1, the two components WormColl and SigGen communicate using a private message queue. Worm instances sent to WormColl from the honeypot or an intermediate source have their length checked to avoid overflowing the message buffer. Each worm instance has an associated ID; if two worm instances have identical IDs, they are considered variations of the same worm or exploit of a vulnerability. The whole structure is buffered and added to the message queue.

At the other end, SigGen waits for new messages from WormColl. Whenever one arrives, its contents are further disassembled and stored within the individual fields of a worm struct for easy access. This worm structure is then sorted into a buffer according to the worm ID, where it awaits further processing.

The actual signature generation is carried out in a separate procedure called by SigGen if enough instances of the same worm have been recieved. This procedure implements the Gibbs sampling algorithm proposed in [3] and stores the result along with some other parameters in a MySQL database. If an entry for this particular worm already exists, it is refined using the newly generated byte frequency distribution signature.

The lenghty process of generating a signature and comparing it to the contents of the database, as well as the fact that it is impossible to predict at what interval this process will take place, are the main reasons to divide the application into components WormColl and SigGen. Structured in this way, one half will continually be able to receive outside input and there is no danger of any of the captured worm instances being lost in transit because the receiver is not

ready. On the other hand, the second half is guaranteed to have enough time to complete the Gibbs sampling and store the signature in the database, as incoming messages will simply be buffered in the message queue until they are picked up. To ensure that all of this works as intended, the capacity of the message queue must be large enough to buffer the amount of data that will arrive, in the worst case, during the time that SigGen is busy running the signature generation algorithm and cannot remove new items from the queue.

3.2 The WormColl Component

The ears of BowNet, so to speak, is the WormColl application. It is implemented as an endless loop that accepts connection after connection from an external component providing the collected and labelled instances of attacks captured with Argos.

Specifically, the program creates a socket and binds it to port 4224, where it listens for clients to connect to it. The clients send their captured and labelled worms to this port using the message format as depicted in Figure 3.2. WormColl expects exactly one message per connection.

Besides the worm code, a message consists of the following fields:

1. *length*: denotes the total length of the message, including the length field itself.
2. *worm ID*: labels the worm for later identification of its type; assigned by the client during evaluation of the attack.
3. *port number*: records the port on which the attack occurred.
4. *protocol*: stands for the contents of the protocol field in the IP header in the packets that delivered the payload, namely TCP, UDP or ICMP.

In addition, WormColl creates and maintains the message queue used to send the worms from one program to the other. Because it is a private message queue with a dynamically generated identification number, the only way for both processes to know this number is to have one of them be a descendant of the other. This is accomplished by a call to *fork()* within WormColl to create a child process. A call to *exec()* then replaces the second instance of WormColl by the code and memory segment for SigGen. In doing that, the ID of the shared message queue can be passed on to SigGen as an argument of the *exec* call.

Because the two processes share a message queue with a dynamically assigned ID that both of them have to know, it does not make sense to have one component running without the other. Restarting the missing component after a crash or manual termination would result in an orphaned message queue unless special precautions are taken. It was therefore decided to implement signal handling in such a way that, whenever one component is shut down in a halfway ordely fashion, the other component terminates itself as well.

If it is WormColl that recieves a termination signal, it checks for a child process and, if it finds one, sends it a termination signal in turn. It then waits for confirmation that the child has terminated before completing its own cleanup procedures. If the signal handler was called into action due to a SIGCHLD signal (meaning that the parent has survived its child), the proceedings are exactly the same. The only difference lies in the result when checking for a child, which makes it unnecessary to send any kill signals and wait for their outcome.

If, on the other hand, SigGen is terminated first, the parent process automatically recieves a signal and handles it according to the description above, while the child only has to take care of its own memory management. Figure 3.4 illustrates the signal handling behaviour for both components.

3.3 The SigGen Component

The SigGen component can be found at the other end of the message queue linking the application together. It assembles the messages read from the queue and stores their individual components in a *worm* data structure. This structure consists of the same four fields as the original message in Figure 3.2 and a character array for the worm code. This worm structure is then lined up with others of the same worm ID in a buffer structure, consisting of a counter for the number of elements, a field for the worm ID for easy identification, a pointer to the next buffer, and enough room to hold a pre-determined number of N worm instances.

The buffers are collected in a linked list, as there is no way of predicting how many different worms will be captured and the actions of adding or removing a buffer are simple and straightforward. An array design, which would be faster in terms of access because it can be sorted according to the buffer ID, has been discarded in favour of the flexibility of having no restrictions on ID ranges and list size.

Once a buffer has accumulated N copies of a given worm, it is disconnected from the list and passed to the actual signature generation procedure PADS. There, a byte frequency distribution signature of pre-determined width W is generated using a Gibbs sampling algorithm as suggested in the relevant paper[3].

If a signature of width W does not exist in the database for a worm of a given ID, a new entry is added to the database table *signatures.PADS*. If a signature of a different width W is found in the database, this signature is replaced completely by the one that has just been computed. If both the ID and the width W match, the new signature is mixed in with the old one by taking into account the number of previous updates *nupdt* and scaling the result back down to result in a probability distribution. The purpose of mixing the old and the new signature in this way is to refine it by incorporating new information without discarding old results completely.

3.4 The Signature Database

The worm signatures and various other parameters are stored in a MySQL database. MySQL was chosen because it is freely available, of small size, and easy to set up and use. It contains one database called *signatures*, which is made up of a single table *PADS* that keeps track of the signatures.

In addition to the actual signature, *PADS* consists of the following fields:

1. *worm ID*: type of vulnerability.
2. *created* and *last updated*: timestamps.
3. *nupdt*: number of updates (starting with 1 for the time that a signature is first stored in the database).
4. *port*: the port that was attacked.
5. *protocol*: contents of the protocol field in the IP header.
6. *W*: signature width, i.e. the length of the significant region of the worm instances.

Of these, port and protocol are not currently used as sorting criteria, but they may be in the future, which explains their inclusion in the parameter list here. Figure 3.3 shows the fields in *PADS* in more detail.

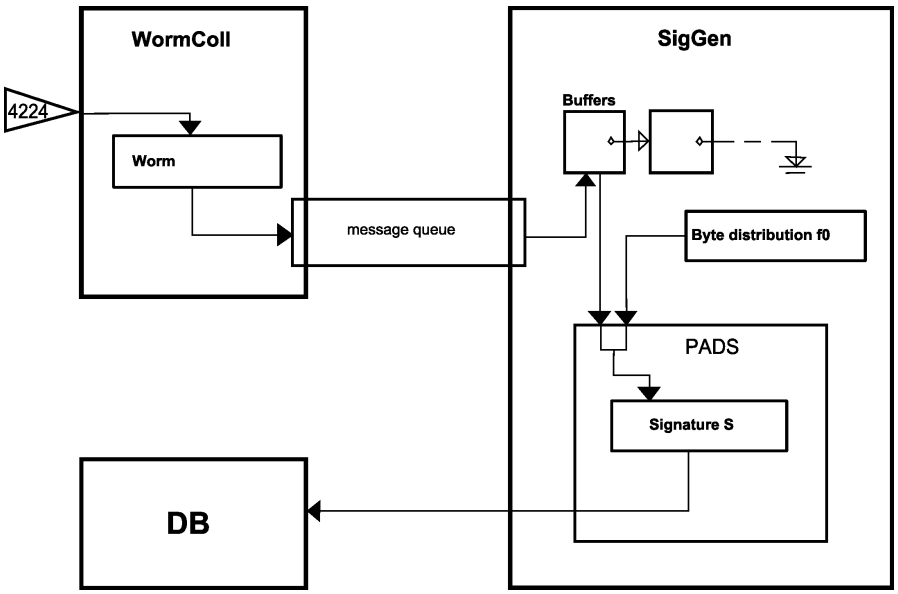

Figure 3.1: Structure of the BowNet application.

message length (int)	worm ID (int)
port number (int)	protocol (char)
worm code (byte string)	

Figure 3.2: Structure of the message format as it is received by the WormColl component.

3.4.1 Database Security Issues

The user *SigGen* has selection, insertion and update privileges on the database *signatures*, but only when connecting from the local host. He is not allowed to change the database itself, though. Only the *root* user of the database can modify the layout of the tables or add and delete tables in the database. This restriction grants an ordinary user exactly the privileges he needs, but not more, and serves as a measure against accidental or deliberate modifications that could affect BowNet's ability to safely and reliably store the information it has extracted.

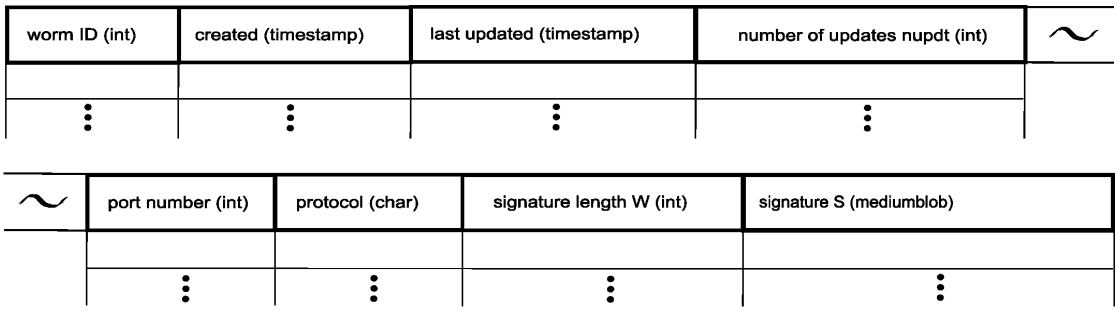

Figure 3.3: Layout of the fields within the MySQL database table.

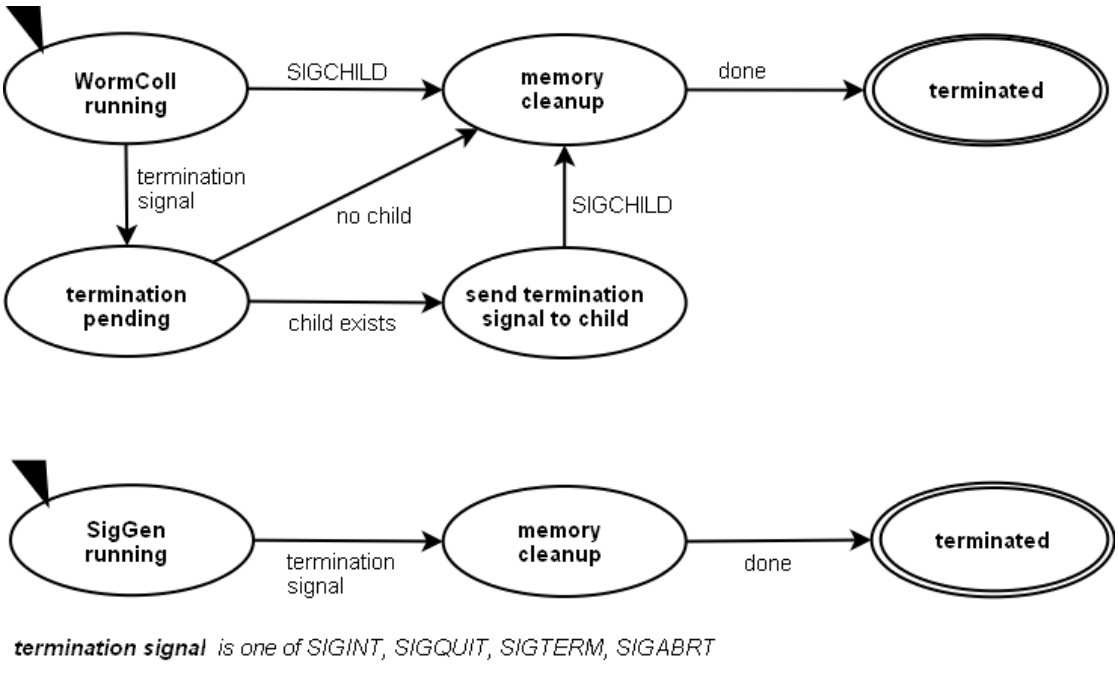

Figure 3.4: State diagrams for signal handling in each component.

Chapter 4

Testing and Evaluation

4.1 Training the Byte Frequency Distribution f0

For the most part, the signatures generated by SigGen consist of a number of byte frequency distributions for the significant region of a polymorphic worm. But the distribution for benign network traffic *f0* is included as well, since it is required to calculate the score of a code segment as described in Section 2.2. Within BowNet, this distribution is read from a file, and is the same for all signatures.

As the byte frequency distribution of normal network traffic across all ports and protocols is not uniform, it needed to be determined from a training set. Instead of capturing live traffic, the training was carried out using weeks one and three of the Darpa IDS evaluation data set from 1999[4]. There are two main reasons for this choice. First of all, the files contain ten days' worth of network trace that can be read an processed within a few minutes. More importantly, though, weeks one and three of the data set are guaranteed to contain no attacks whatsoever. This made sure that *f0* was not accidentally trained to take into account anomalies that it should not have.

Because the Darpa data sets are stored in tcpdump format, *NormalDist*, the program that trained the byte frequency distribution for normal network traffic makes use of the *pcap* library to access packet after packet with relatively little fuss. The content of the payload is then examined byte after byte and the occurrence of each possible value is counted. These sums are divided by the total number of bytes that were read to scale them to values that will sum up to one. To avoid zero values in the distribution fields, a pseudo count of 1 is used and taken into account during scaling, resulting in a division by *total_count + 256*.

The result was saved in the file *NormalTraffic.dist*, which is read by SigGen on startup and passed to its central procedure PADS for score calculation.

The training phase needs to be run through only once. After that, the file *NormalTraffic.dist* simply needs to be included with the files that make up BowNet as it is.

4.2 Test Runs

The Gibbs sampler forms the core of the application, so testing focussed on its ability to create signatures that would produce high matching scores when compared to anomalies of the same type.

To this end, two different data sets were created by the program *DataGen*. Each of these consists of 100 entries in the message format defined in Section 3.2, representing 100 worms of the same type. The payload of each message contains a sequence made up entirely of a predefined

number of As starting at a random position that stands for the malicious code segment. The rest of the space is filled with random bytes according to the distribution read from the file *Normal-Traffic.dist* to simulate the garbage payload of normal traffic that the worm code is embedded in.

The length of the anomaly is different for each data set. One has no specifically inserted As at all; any that may occur within the data do so naturally, i.e. they are there as a result of the distribution read from f0. The other contains a sequence of 40 As per somewhere in each entry's 2000 byte of payload data. The set with length zero is used as a control group, as it models attack-free network traffic and the scores the Gibbs sampler produces should be uniformly low, no matter what the width of the significant region may be.

The paper forming the motivation for this work tests various properties of both the Gibbs sampler and the Expectation Maximisation algorithm. As this work focusses on the Gibbs sampler, an attempt was made to replicate some of the results, to see, amongst other things, whether they could be reproduced under slightly different circumstances.

Two different evaluations of the original paper were chosen for this task. Figure 3 in [3] illustrates the final score Ω with respect to the number of iterations for each signature generation method. Because Expectation Maximisation does an update on the starting point of each worm variant in every iteration but the Gibbs sampler only changes one of them, the authors define the scale to be the *number of per sequence iterations*. This means to say that for each iteration of the Expectation Maximisation algorithm, a hundred iterations of the Gibbs sampler's main loop need to be performed in order to, on average, select each worm once for comparison to Θ. So if the scale goes up to 40, 4000 iterations are necessary for a set of 100 worms.

The sort of behaviour seen in this figure was to be reproduced for the Gibbs sampler. Instead of running the algorithm three times and plotting the individual results, however, it was to be executed 100 times to be able to compute a reliable mean value of the final scores, as well as a few selected individual curves. This was done to give an indication of the variety within the runs, and to take a look at how far a selected run may deviate from the average.

On top of that, the algorithm does not record the current score Ω for the selected worm, but the average of the score of each worm given the current configuration. This is done for both the normal and the anomalous data set.

The right half of figure 4 in [3] plots the matching scores of anomalous and normal traffic against the width of the signature W for both Expectation Maximisation and Gibbs sampling. This plot essentially shows two things. One, even for widths that are many times smaller than the malicious code segment, both algorithms clearly distinguish between good and bad, the only differences being the average quality and the variance of the respective scores. And two, the final matching score Ω for the data sets containing malicious payload actually begins to decrease after a certain signature length, because more and more and more of the garbage payload is being taken into account.

The shortest signature width that was tested was 10, however. Since the anomalous set created by *DataGen* contains a sequence of interest that is much shorter than the Blaster worm used in the paper, it made sense to start with a minimal width of 1 and go no further than 50. The assumption was that little or no information could be extracted for very small widths and it seemed interesting to find out from when on the signature would become consistently reliable. On the other hand, a maximum of 50 was deemed to be wide enough to observe the effect of score decrease as normally distributed bytes begin to influence the result.

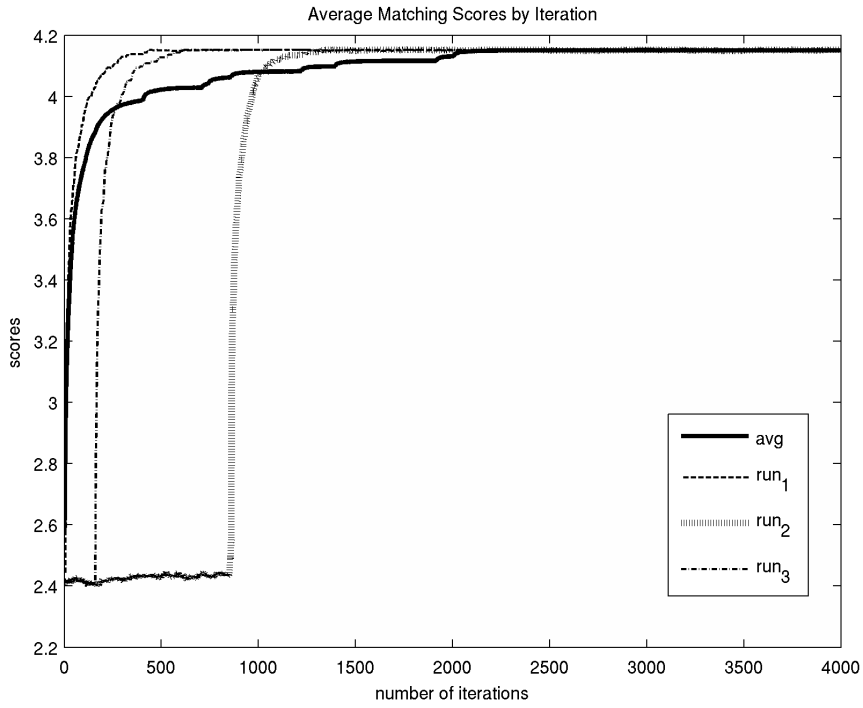

Figure 4.1: Mean values of final scores by number of iterations in the main Gibbs sampling loop for a string of As of length 40 and a significant region of width 10

4.3 Results

It became clear during testing that convergence using a pre-set ϵ to decide whether two consecutive scores are close enough to each other to stop does not yield the desired results in practice. The reason for this is that, because of the random element when choosing an improved starting position for the worm, the new score may improve as well, but it may improve by a large step, or hardly at all. It is not possible to predict which case will occur, and both cases may occur well before a value close to the global maximum has been reached. If the algorithm is implemented to stop when a new value is within a certain small range of the old one, it will usually do so too early.

For practical purposes, it is therefore better to determine the number of iterations until the score begins to oscillate around a value close to the optimum, making further computations unnecessary. This was the point of the test runs that form the basis for Figure 3 in [3] and in the recreation of these that was attempted here under slightly differnet circumstances. The reproduced results can be seen in Figure 4.3. Overall, they match the paper well: on average, the score converges to a reasonably stable value after about 600 iterations of the main loop, which corresponds to the six iterations per sequence as seen in the original article.

The selected runs also show another reason why it would be unwise to break out of the loop too soon. In some cases, the algorithm gets stuck in a local maximum early on and it takes quite a while before it manages to jump out of it, at which point the score rises sharply and quickly approaches the global maximum. If an unsufficient number of iterations had been run through, the global maximum may never have been reached and the returned score would have been fairly inaccurate.

Since a total of 600 iterations was established as a value producing reliable results in the first batch of tests, it was kept as the number of iterations, now constant for every run, for the second one, which was an explorations of the behaviour of the average score depending

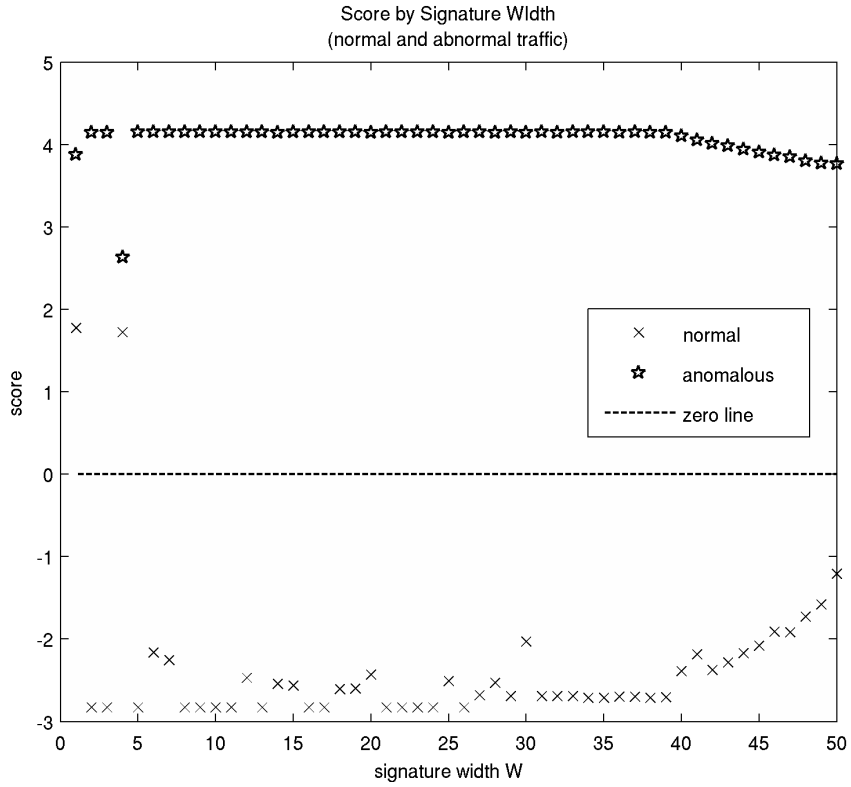

Figure 4.2: Final scores depending on the width of the significant region W after 600 iterations of the Gibbs sampling loop. Scores for the anomalous data set are above zero, scores for normal traffic are below.

on the width of the significant region W. Again, the results match the paper.

The comparison of Figure 4.3 looks equally promising. Up to a width of about five, the scores exhibit little consistence, implying that the corresponding signature is not particularly reliable. Not only do the scores for the anomalous set vary a more than usual, but the fact that, at those lengths, two of the scores for the normal set are to be found well above zero illustrates this point even better. Zero sets the threshold for a score outside the norm; anything from the random data set that ends up above that must be on the wrong side and the result can therefore not be trusted.

Once past the five character mark, however, the score values, especially those for the simulated worm, remain remarkably stable until around a width W of 40. After this value, they begin to decrease as predicted, and again in accordance with the original plot in the paper. Both results imply that the detector will have similar reactions to any kind of anomaly. They also imply that the anomaly does not have to be very long to be picked up.

There is a drawback to all of this, though, and it lies in the time needed to compute the scores and the corresponding signatures. Even a run of 600 iterations, the minimum requirement to getting reliable values with a high probability, takes several minutes to complete. While the longer ones with 4000 iterations were executed only once to establish the lower value, the time requirement for 600 iterations needs to be reckoned with if the algorithm is to be used in practice. An effort could (and probably should) be made to optimise the code for perfomance, but even then, incoming data must be buffered until it can be processed while the signature generator is running. The expected volume that is received during this time will have an impact on the size of the buffer (the message queue in this concrete case), which must be calculated beforehand to avoid data loss.

Chapter 5

Summary and Conclusions

The goal of this work was to look at the class of byte frequency distribution signatures, choose one, and implement it for possible use within the Argos honeypot system.

Two methods were considered. The first one involves splitting network connections by port and length, modelling normal behaviour for each combination, and comparing incoming traffic to these centroids to decide whether an alarm should be raised. The byte distribution of such a payload is reordered into a Z-string by the number of occurrence of each individual byte value. This distribution can then be used as a simple signature for redistribution.

The second method only looks at network traffic in general, and compares new information to the model using PADS (Position-Aware Distribution Signatures). Those are more complex to calculate but on the whole seem to be much more accurate. On the other hand, administration overhead and potential memory consumption for the single byte frequency distribution are much smaller and more manageable. Because of this, and because the first method is primarily an intrusion detection system rather than a signature generator, PADS, specifically the variant that uses Gibbs sampling, was chosen for implementation.

The BowNet package was designed as a framework for collecting labelled worms, extracting a signature from them, and storing this signature in a database for eventual redistribution. It consists of three components that fill these roles: WormColl, which listens on a predefined port for incoming worm instances, SigGen, which creates the signatures and stores or updates them in the database, and a MySQL database that serves as the repository. Testing the whole setup without a working client that labels and delivers the worms proved difficult, though, and so only individual parts of it are known to work.

The Gibbs sampler at the core of the application was tested with the results of the original paper in mind. Two of these results were replicated successfully under slightly different circumstances, which implies that Gibbs sampling used for signature generation is as reliable a method as initially expected.

The one major drawback of the method is the computation time needed to acquire the desired results. However, it should be possible to deal with this issue by buffering incoming new data until it can be processed. This means that the chosen method is suitable for use in practical applications in spite of the computational effort it requires.

Further work is needed to accomplish this. The next section, Section 6, offers a few suggestions and possibilities on how to take the work done in this semester thesis a few steps further.

Chapter 6

Outlook

There are plenty of areas where the current setup could be improved upon. The following list gives a few pointers, but as the whole topic is one of active research a comprehensive overview would be far beyond the scope of this report.

- Due to the lack of a proper client application feeding it information, the BowNet package as a whole has not been tested. Individual parts, such as the message queue communication mechanism, signal handling, database access and the signature generator at the core all work individually when run outside the main application. If the whole package is ever to be used for anything, further testing and debugging will be necessary.
- There is a definite tradeoff between the accuracy of a signature and the time required to compute it. These two aspects need to be balanced so as to give the best possible results without leaving the system unable to receive new input for too long a time period. Further experiments are required to find out the best combination under the given circumstances.
- If Gibbs sampling as a signature generation algorithm is to be used in practice, the issue of classifying captured worm instances before they are fed to the signature generation routine. The Gibbs sampler needs to work under the initial assumption that its input is of a somehow similar type to produce meaningful results. If this is not the case, it will inevitably end up comparing apples and oranges, so to speak; the outcome will be unusable or even harmful to the systems that employ it.
- As the chosen signature generator does not use them, the port and protocol fields in the messages containing the captured worms are ignored. Specifically, sorting of the worms into their buffers happens based on the worm ID only. When storing the signature in the database, the port and protocol values are simply copied from the first worm instance without checking if they are the same in all cases. It would be good if the sorting process during buffering took into account parameters besides the ID as well.
- The Gibbs sampler used will, in principle, also work if the significant regions used to compute the score are non-contiguous. It would require extension of the starting point selection method to handle multiple points. It would also require fine-tuning of the new parameters such as number of blocks and length of an individual block. The sampling algorithm itself does not need to be altered, however, and the extended method may produce even more accurate signatures.

Appendix A

Schedule

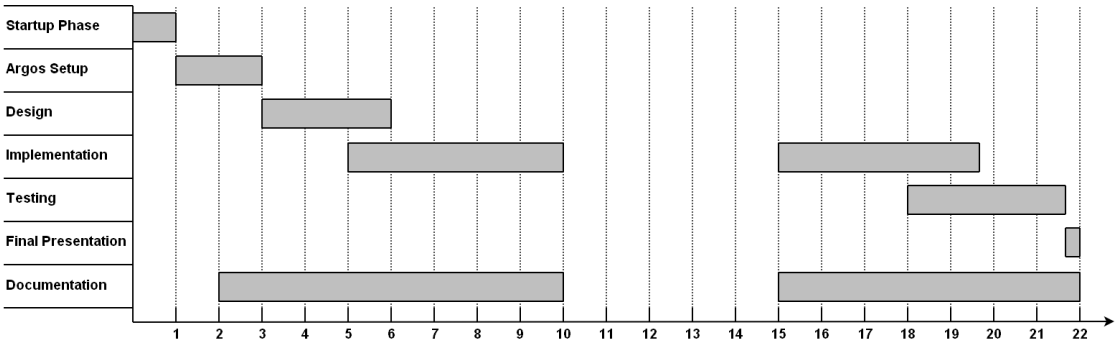

Figure A.1: Overview of the approximate project schedule.

Needless to say, this schedule was not met...

Appendix B

Original Problem Description

The tasks as defined in the original problem sheet were the following:

1. **Set up an appropriate test and development environment for Argos:**
   The test environment should be designed with security in mind. This is necessary because we plan to attack Argos with real-world exploits. The code and some documentation can be found at: http://www.few.vu.nl/porto/argos/.
2. **Study the class of byte frequency distribution signatures:**
   Study publications about byte frequency distribution signatures and select or even improve a specific type and/or calculation method.
3. **Extend the system so that it generates byte frequency distribution signatures:**
   Adapt the selected type/method for use with the Argos system and do a proof-of-concept implementation.
4. **Evaluation:**
   Evaluate the implementation by launching different attacks against Argos and by checking the quality of the generated signatures (false positives/false negatives, handling of polymorphic attacks).

There were five deliverables tied to those tasks:

1. A test and development environment for Argos along with an installation/configuration and user guide.
2. A brief analysis of the concept and its implementation of at least two existing approaches generating and using byte frequency distributions signatures.
3. A detailed description of the concept applied in this thesis along with a brief overview about its implementation.
4. A running prototype of the implemented software together with appropriate documentation (code/installation, e.g. using e.g. doxygen) and the source code.
5. Documentation presenting the evaluation setup, methodology and the corresponding results.

Appendix C

Software Installation and Configuration

C.1 Argos Installation Guide

The following is a guide to install and run Argos on a Debian system as provided by the Neptun group at ETH Zurich, so specifics may vary, particularly when it comes to installed packages.

Most of the information was collected form the official Argos website[5]. Instructions to set up the network on Debian in an elegant fashion came from a different website[6], also accessible from the official one.

First of all, make sure the necessary libraries and packages, particularly *bridge-utils* and the *SDL* binaries and developer package are installed. If the TUN/TAP device is not enabled, switch it on. (When running the Neptun-provided version of Debian, the interface will already be on.)

Download the Argos tarball from the official website and unpack it. Switch to the directory containing the files and run the included configuration script:

```
$ ./configure
```

This is the point where you find out if you are missing any packages because the configuration script will tell you. If that is the case, install the necessary parts and run the script again.

As root, build and install Argos by typing

```
# make
# make install
```

This will install Argos into the directory */usr/local/bin*.

To create an image for the guest operating system, use *qemu-img*. The following command will create the image file 'Apfel.img' of maximum size 3GB, using the emulator's copy-on-write file format:

```
$ qemu-img create -f qcow Apfel.img 3G
```

It is now possible to install the guest operating system on this image. The command is the same both for installation from a CD or DVD and from an ISO image:

```
$ argos -cdrom <device> -hda Apfel.img -boot d -m <RAM>
```

where <device> denotes either the cdrom drive or the path to an image file, while <RAM> stands for the amount of virtual RAM in MB that Argos should run with.

In order to capture attacks, Argos will need network access. The Nepenthes website[6] offers a very elegant solution to setting up the network interface in Debian. Essentially, it involves creating a bridge *br0* to your network interface (presumably *eth0*), assigning this bridge an IP and network mask, and setting it the default route to your gateway. All this can be done by changing the settings in the network configuration file */etc/network/interfaces*: if your initial configuration looks something like this

```
auto eth0
iface eth0 inet dhcp
```

you need to change it to

```
# auto eth0
# iface eth0 inet dhcp

auto br0
iface br0 inet static
   address         <your IP>
   network         <your network IP>
   netmask         <the network mask to go with your network>
   broadcast       <the broadcast address for your network>
   gateway         <your default gateway>
   bridge_ports eth0
   bridge_fd       1
   bridge_hello    1
   bridge_stp      off
```

filling in the appropriate values for IP, network, broadcast address, netmask and default gateway as necessary.

Run Argos using the command

```
# argos -hda Apfel.img -m <RAM> -snapshot -cleanvm -<OS>
```

<OS> can be one of the following: *linux*, *win2k* or *winxp*. The *snapshot* option is not mandatory, but it prevents changes from being written directly to the image. Note that Argos must be run as root because it needs access to */dev/net/tun*.

C.2 MySQL Setup

To install and set up the MySQL server needed to store the signatures, start by adding a user and a group *mysql* that the server will run as to your system. Pick any password you like for this user; on the test system, the password for user *mysql* is *m!$Q1*.

Next, install both the MySQL server and the MySQL client. The most recent version for Debian's stable release is 4.1, so that was the version that was used on the test system.

Switch to user *mysql*, as there is no need to do the rest of the setup as root. Using the *mysqladmin*, set the root password for access to the MySQL databases:

```
$ mysqladmin -u root password 'your_password'
```

where 'your_password' is replaced by the root password chosen for the server. On the test system, it is set to *T0o2+6Q#*.

The server is now accessible for furhter configuration by typing

```
$ mysql -u root -p
```

which will get the system to ask for the root password that has just been set.

The remaining configuration steps are the following:

```
mysql> use mysql
mysql> insert into host(Host,Db) values('localhost','signatures');
mysql> insert into user(Host,User,Password)
       values('localhost','SigGen','$i9g3nn);
mysql> insert into db(Host,Db,User,Select_priv,Insert_priv,Update_priv)
       values('localhost','signatures','SigGen','Y','Y','Y');
mysql> create database signatures;
mysql> use signatures
mysql> create table PADS(ID int,created timestamp,updated timestamp,
       nupdt int,port int,protocol char,W int,S mediumblob);
mysql> quit;
```

This allows the host *localhost* to connect to the database *signatures* as dbuser *SigGen* with password *$i9g3nn* and grants him select, insert and update privileges for the tables within. Note that the user and password for access to *signatures* must be *SigGen* and *$i9g3nn* respectively, as they are hardcoded into the SigGen component for communication with the database.

The actual database *signatures* and the table *PADS* have also been created; reload the new configuration with

```
$ mysqladmin -u root -p reload
```

Needless to say, the server has to be running while all of this happens. On Debian, it will be started automatically after installation. By default, it will also start when the system is booted and stop when it is shut down.

C.3 The BowNet Package

Besides the MySQL database, BowNet consists of two components that make up the tarball of the same name. The following are instructions on how to install and run the application and a more detailed overview of the procedures and functions contained in it.

C.3.1 Installation

Copy the BowNet tarball to the directory that you want the application to be in and unpack it. Change to the newly created directory *BowNet/src*.

Compile the two components *WormColl.c* and *SigGen.c*. Technically, the output for WormColl.c can be given any name (although it is probably easiest to simply call it *WormColl*. SigGen, however, needs to be compiled using the following command

```
$ gcc -lmysqlclient -o SigGen SigGen.c
```

because the name 'SigGen' for the second component is a constant in the first, which uses it for the *exec* call to start it. The option *-lmysqlclient* links the external functions for database interaction used within SigGen.

Once the two components have been compiled, they can be run from the directory they are in, or they (along with the file *NormalTraffic.dist*) can be copied to a location of your choice if you don't want them to be in the same place as the source files. At any rate, they are now ready for use.

C.3.2 User Guide

BowNet as a whole is started by starting WormColl (or whatever name the component was given during compilation) on the command line. The second component SigGen will be started automatically within WormColl. Note that it does not make sense for SigGen to ever be started manually by the user, as that program requires the dynamically assigned ID of a message queue initialised in WormColl.

To stop the application (by *Ctrl-C* or one of the various *kill* commands), it is sufficient to stop WormColl only. The signal handlers will see to it that SigGen is terminated as well. In other words, any interaction with the application should happen with WormColl alone, while everything else is designed to run in the background.

C.3.3 Component Overview

The following gives a more detailed overview of the functions contained in each of the two components and of their input and output values.

WormColl

Application Constants

- *INT_SIZE*: the size of an integer in bytes to make the code more readable.
- *STRT_TXT*: the starting point of the worm information (after the length and type fields) within a message recieved on the port WormColl listens on.
- *DEFAULT*: tells the *socket()* call to use the default protocol for data streams.
- *PORT*: the port that WormColl binds to to listen for messages.
- *MAXCONN*: the maximum number of connections accepted by WormColl on port 4224.
- *S_PERM*: the access permissions of the message queue used to communicate with SigGen.
- *MAXLENGTH*: the maximum size of a message that WormColl will receive via TCP, and that it will pass on to SigGen through the message queue.

Specialised Data Structures

1. *message*: the message format used to send the collected worm instances to the signature generator for further processing. Each of these messages is filled with the information received over the TCP connection and appended to the message queue. It consists of the following fields:
    - *mtype*: integer field to denote the message type; set to the worm ID.
    - *mtext*: character array of size MAXLENGTH that contains the auxiliary fields as described in Section 3.2 as well as the actual worm code.

main
Contains all of the code needed to run WormColl as long as no signals are received. Calling sequence:

```
./WormColl
```

1. Called by:
    - *user*, on the command line.

2. Calls:
    - *SigGen*, via an *exec* call.
3. Input Parameters:
    - *none*
4. Return Value:
    - -1, in case of failure.
    - *child exit status*, in case of success.

sig_handler
Handles the signals that terminate or abort the application, as well as the case of unexpected child termination. Calling sequence:

```
sig_handler(signal);
```

Note that this procedure is never called by the application itself, but by the system when an asynchronous event occurs.

1. Called by:
    - *system*, in case of an external event.
2. Calls:
    - *none*
3. Input Parameters:
    - *signal*, an integer value that stands for the type of signal that was raised; not used.
4. Return Value:
    - *none*

SigGen

Application Constants

- *N*: the number of worm instances needed to compute a signature.
- *W*: the width of the computed signature, i.e. the length of the significant region.
- *INT_SIZE*: see Section C.3.3.
- *FLOAT_SIZE*: the size of a floating point number in bytes to make the code more readable.
- *DIST_ELEMS*: the number of elements in the byte frequency distribution for one position, i.e. the maximum number of different bytes.
- *HEADER*: the length of the header of the message format as specified in Section 3.2, i.e. the number of bytes to be read from the dequeued message before the actual worm code begins.
- *NEXT_MSG*: tells the *msgrcv()* call to read the next message in the queue, intependent of its type.

- *MAXLENGTH*: see Section C.3.3.

Specialised Data Structures

1. *message*: see Section C.3.3.
2. *worm*: stores information about a captured worm, as well as the worm itself; analogous to the message format described in Section 3.2. It consists of the following fields:
   - *size*: the total size of the data in bytes.
   - *ID*: a unique identification number that labels the captured type of attack to classify the individual worm instances.
   - *port*: the number of the port that was attacked.
   - *protocol*: the protocol under which the attack occurred; copied from the TCP header that originally delivered the malicious payload.
3. *buffer*: the buffer structure used to sort the received worms into until *N* instances of a given type have been collected. It consists of the following fields:
   - *elements*: the number of worms currently contained in the buffer.
   - *ID*: the type of worm contained in the buffer.
   - *data*: a worm array of size *N* to temporarily store the worm instances.
   - *next*: a pointer to the next buffer in the list.

lambda
Calculates the matching score of a worm by comparing a potential significant region of width W to a given signature. Calling sequence:

```
score = lambda(theta, Sx, s\_pos);
```

1. Called by:
   - *PADS*
2. Calls:
   - *none*
3. Input Parameters:
   - *theta*, a float array of size *W+1* by *DIST_ELEMS* that contains a signature generated by *PADS*.
   - *Sx*, the worm instance that the signature *theta* is being compared to.
   - *s_pos*, the starting position of the potential significant region for worm *Sx*.
4. Return Value:
   - *score*, the matching score resulting from the comparison of worm and signature.

loglambda
Calculates the matching score of a worm and a signature in the same way as *lambda()*, but returns the base 10 logarithm of the result. Calling sequence:

```
log\_score = loglambda(theta, Sx, s\_pos);
```

1. Called by:

- *omega*

2. Calls:

- *none*

3. Input Parameters:

- *theta*, a float array of size *W+1* by *DIST_ELEMS* that contains a signature generated by *PADS*.
- *Sx*, the worm instance that the signature *theta* is being compared to.
- *s_pos*, the starting position of the potential significant region for worm *Sx*.

4. Return Value:

- *log_lambda*, the logarithm of the matching score.

main
Calling sequence:

```
./SigGen <message queue ID>
```

1. Called by:

- *WormColl*, via an *exec* call.

2. Calls:

- *PADS*

3. Input Parameters:

- $<$*message queue ID*$>$, to communicate with WormColl.

4. Return Value:

- 0, upon completion of memory cleanup. This is independent of whether the termination signal was the result of an internal error or an external event.

omega
Calculates a final score by finding the starting position of the significant region in a worm instance that maximises the logarithmic score when comparing the region to a given signature.
Calling sequence:

```
final = omega(theta, Sx);
```

1. Called by:

- *PADS*

2. Calls:

- *loglambda*

3. Input Parameters:

- *theta*, a float array of size *W+1* by *DIST_ELEMS* that contains a signature generated by *PADS*.
- *Sx*, the worm instance that the signature *theta* is being compared to.

4. Return Value:

- *final*, the maximum logarithmic score, weighted by the signature width *W*, that can be achieved when comparing worm *Sx* to signature *theta*

PADS
Implements two different but equally important parts of the application: the Gibbs sampler that computes the optimal signature for a buffer full of worms, and the communication with the database to store this signature. Calling sequence:

```
PADS(current_buffer, f0);
```

1. Called by:

- *main*

2. Calls:

- *lambda, omega*

3. Input Parameters:

- *current_buffer*, a buffer with *N* accumulated instances of the same type of worm.
- *f0*, a float array containing the byte frequency distribution for normal network traffic as read from the file *NormalTraffic.dist*.

4. Return Value:

- *none*

sig_handler
Handles the signals that terminate or abort the application. Calling sequence:

```
sig_handler(signal);
```

Note that this procedure is never called by the application itself, but by the system when an asynchronous event occurs. In most cases, the signal in question will have been sent by the parent process, i.e. WormColl.

1. Called by:

- *system*, in case of an external event.

2. Calls:

- *none*

3. Input Parameters:

- *signal*, an integer value that stands for the type of signal that was raised; not used.

4. Return Value:

- *none*

C.4 Test Tools

C.4.1 NormalDist

The byte frequency distribution for normal network traffic *f0* that is contained in the file *NormalTraffic.dist* was trained from the Darpa IDS evaluation data set using this tool. It reads files in *tcpdump* format and counts the occurrence of each possible byte. As it uses the *pcap* library, it must be compiled with the *-lpcap* option:

```
$ gcc -lpcap -o NormalDist NormalDist.c
```

This will link the needed library functions.

NormalDist takes one or more files as input. The calling sequence is therefore:

```
$ ./NormalDist <tcpdump file 0> <tcpdump file 1> ... <tcpdump file n>
```

which will result in the file *NormalTraffic.dist*. Note that this file is already included in the BowNet package and does not need to be retrained. The instructions given here are purely for completeness' sake, or for the case that one might want to test a different data set of normal network traffic.

C.4.2 DataGen

DataGen is used to generate files of test data to feed to the signature generator or directly to the Gibbs Sampler. As it stores the generated byte strings in the same format as the message described in Section 3.2, it requires several additional input parameters apart from the name of the output file. It is called in the following way:

```
$ ./DataGen <outfile> <keyword> <ID> <f0>
```

where <keyword> stands for the anomalous string that is to be hidden among the random bytes, <ID> denotes the worm type and can be assigned an arbitrary value, and <f0> is the file containing the previously computed byte frequency distribution for normal traffic. The program will create 100 byte strings of length 2000 characters and write them to <outfile>.

C.4.3 GibbsSampler

This application implements the Gibbs sampler that forms the core of the SigGen component, but the focus lies on the calculated matching scores instead of the signature. The goal here is to test the behaviour given an anomalous string within the random test data in relation to various lengths *W* for the significant region. Because it uses a couple of functions from the *math* library, it needs to be compiled with the option *-lm*. It is then called with the command

```
$ ./GibbsSampler <datafile> <random datafile> <width> <iterations>
        <final scorefile> <average scorefile> <average random scorefile>
```

<datafile> contains the test data generated with *DataGen* that is to be read. The size of the significant region *W* is given by <width>, while the results are stored in <scorefile> and <iterfile> for the scores and the number of iterations, respectively. Note that the application appends to these files whenever it is run rather than overwriting anything.

The functions contained within are analogues to those in SigGen, but the number of input parameters may differ. They are the following:

```
double lambda(int W, float theta[W+1][DIST_ELEMS], worm Sx, int s_pos);
double loglambda(int W, float theta[W+1][DIST_ELEMS], worm Sx, int s_pos);
double omega(int W, float theta[W+1][DIST_ELEMS], worm Sx);
double gibbs(b_ptr current_buffer, float f0[], int W,
                            int iters, FILE* worms, FILE* sc, FILE* rsc);
```

where *theta, Sx, s_pos, current_buffer* and *f0* are the same as in SigGen. *W* is, again, the width of the significant region (now no longer an internal constant), while *iter* gives the number of iterations that the main loop has to complete before returning. *worms*,*sc* and *rsc* point to the file descriptors of the random data set, the output file for the anomalous matching scores and that for the normal ones, respectively. The function *gibbs* takes the role of the procedure *PADS* in SigGen, but it returns the matching score right away and does not have to interact with a database.

Bibliography

[1] Georgios Portokalidis, Asia Slowinska, Herbert Bos: *Argos*: an x86 emulator for fingerprinting zero-day attacks by means of dynamic data flow analysis. Technical Report IR-CS-017, October 2005

[2] Ke Wang, Salvatore J. Stolfo: Anomalous Payload-based Network Intrusion Detection. In *7th International Symposium on Recent Advances in Intrusion Detection*, Sophia Antipolis, France, September 2004

[3] Yong Tang, Shigang Chen: Defending Against Internet Worms: A Signature-Based Approach. In *Proceedings of IEEE INFOCOM*, March 2005

[4] http://www.ll.mit.edu/IST/ideval/data/1999/1999_data_index.html

[5] https://gforge.cs.vu.nl/projects/argos/

[6] http://nepenthes.mwcollect.org/howto:setting_up_argos_the_0day_shellcode_catcher/