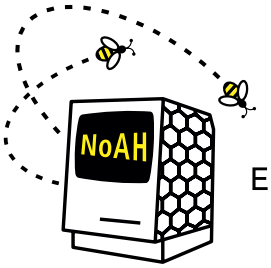
SIXTH FRAMEWORK PROGRAMME

Structuring the European Research Area Specific Programme

RESEARCH INFRASTRUCTURES ACTION



European Network of Affined Honeypots

Contract No. RIDS-011923

D1.2: Attack Detection and Signature Generation

Abstract: The objectives of this contribution to the NoAH project are to develop novel cyberattack detection techniques to provide early warning at the moment the attacks start to spread on the network and to contain or slow down the spread of the cyberattacks by automatically generating a signature that could eventually be read by firewalls or Intrusion Prevention Systems. The deliverable provides an assessment of the feasibility of this objectives for different types of cyberattacks like viruses, polymorphic worms or Denial of Service attacks with respect to the type of sensor that is used in the NoAH project: Honeypots. Based on the assessment and considerations about future threats it is explained why the focus of the detection and signature generation is laid on polymorphic remote attacks. With respect to this focus, new techniques to detect cyberattacks using honeypots are proposed (and their limitations are described). Furthermore a new method to automatically generate signatures using an extended version of the proposed detection system is presented. The requirements for the proposed attack detection and signature generation techniques are described and the integration into the NoAH architecture and into the containment system is discussed.

Contractual Date of Delivery	31/03/2006
Actual Date of Delivery	15/05/2006
Deliverable Security Class	Public
Editor	ETHZ

The NoAH Consortium consists of:

FORTH	Coordinator	Greece
VU	Principal Contractor	The Netherlands
TERENA	Principal Contractor	The Netherlands
FORTHnet	Principal Contractor	Greece
DFN-CERT	Principal Contractor	Germany
ETHZ	Principal Contractor	Switzerland
VTRIP	Principal Contractor	Greece
ALCATEL	Principal Contractor	France

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of the NoAH project is to design an efficient containment system for current and future cyberattacks. Fast containment of cyberattacks involves three consecutive steps. First, an ongoing attack must be detected on one or multiple of the sensors. Second, a meaningful signature or alert for the detected attack must be generated. Third, this signature or alert must be distributed to other vulnerable systems before they get attacked themself. Attack detection, signature generation, and signature dissemination are currently under intensive reasearch. This deliverable specifically addresses the first and the second step, namely cyberattack detection and signature generation in NoAH.

The main aim of this deliverable is to explore the design space for honeypot-based cyberattack detection and signature generation, and to specify the resulting requirements for other parts of the NoAH architecture. Therefore, this deliverable contains the following contributions:

- A detailed review and classification of existing approaches for attack detection and signature generation
- An analysis of the drawbacks and benefits of the individual approaches
- A specification of requirements for the NoAH architecture regarding signature generation and attack detection of present and future attacks
- Four design proposals for attack detection and signature generation within NoAH

In order to evaluate which approaches are best suited for NoAH, we present a novel classification scheme, and outline the drawbacks and benefits of different categories of approaches. Our main classification criteria are i) *attack detection capabilities* such as attack types detected by an approach, and ii) *signature generation capabilities* such as signature generation delay and applicability to polymorphic attacks.

We present four design proposals for signature generation in NoAH that were derived from our preceding analysis. These four proposals are based on Argos, which is described in detail in Deliverable 1.3, as attack detection mechanism, and target different regions of the signature generation design space.

Finally, we specify the requirements for other parts of the NoAH architecture in order to support the most promising approaches for cyberattack detection and signature generation. This includes requirements for the NoAH containment environment as well as for the overall NoAH infrastructure.

The rest of this deliverable is organized as follows. Chapter 2 introduces levels and terms used throughout this report. A review of existing approaches to cyberattack detection and signature generation is given in Chapter 3. In Chapter 4, all previously described approaches are classified, and their drawbacks and benefits are analyzed. The requirements for other parts of the NoAH architecture in order to support the most promising approaches for cyberattack detection and signature generation are presented in Chapter 5. Our own design proposals for cyberattack detection and signature generation within NoAH are described in Chapter 6. Finally, Chapter 7 concludes the deliverable.

Chapter 2

Attack Detection and Signature Generation: Levels and Terms

Attack, attack detection, signature, signature generation and related terms like exploit, attack pattern and injection vector are terms of wide comprehension. It is therefore necessary to specify how they are understood and used in this report. Basically, our use of these terms borrows from [20]:

Exploit: An exploit is an instance of an attack pattern. Its purpose is to compromise a specific piece of target software.

Attack: An attack is the act of carrying out an exploit. It is an event that exposes a software system's inherent logical error(s) and invalid state(s).

Attack pattern: An attack pattern is a generic description on how to build a specific kind of attack. It may involve many dimensions like e.g. timing, required resources and techniques. Basically, it involves an injection vector and some payload [1].

Injection vector: The goal of the injection vector is to break into the target system and to place the attack payload into an activation zone. In most cases, an attack must match special formatting restrictions to do so. The injection vector describes truly generic rules for the formatting/grammer of an attack by taking into account e.g. the position of various fields, the amount of data accepted and the accepted/expected syntax [2].

Activation zone: The area within the attacked service/software where the payload has to be placed in order to be executed/activated.

[1] Usually, for denial of service attacks, a payload is not necessary

[2] Depending on the security mechanisms protecting the target system, an injection vector may become very complex

(Attack) Payload: A payload consists of code/input that the attacker wants to be ”activated” using an injection vector. The intend of the attacker is realized by activating/executing it. Payloads may e.g. be a piece of code to be executed, a shell command to be passed to a command interpreter or a piece of text to be injected into a file.

Signature: A description (signature) for a specific activity/class of activities on a system. The description has (ideally) the following property: it matches only the activity/class of activities it describes (unambiguous).

Attack Detection: Attack detection is the process of observing a system in order to separate normal activities from those representing an attack. The process defines normal activities either explicitly or implicitly.

Signature Generation: Signature generation is the process of creating a signature.

The following section presents a simplistic hierarchical model of the Internet and identifies the different levels at which attack detection and signature generation can be performed. Furthermore, it indicates which of these levels are relevant in the context of NoAH. Section 2.2 introduces the term ”polymorphism” and explains how it is used in this report.

2.1 Attack Detection and Signature Generation Levels

As stated in the definitions in the previous section, attack detection and signature generation require information about the activities of a *system*. But since a system can take many different forms, the eligible system categories have to be identified. The categories of interest are represented by (combinations of) levels in the global system model shown in figure 2.1.

Global View: At this level, information provided by the lower levels can be collected and analyzed. Hence, attack detection and/or signature generation methods using e.g. host-level data provided by multiple hosts have to be installed here. In NoAH, this level is mainly used to store and exchange attack detection and signature generation information.

Autonomous System (AS)/Backbone: Basically, at this level, all inter-AS traffic could be captured and analyzed. Additionally, some information like e.g. the one exchanged with the Boarder Getaway Protocol (BGP) is only available on the AS/Backbone level. Nevertheless, this level is not in the focus of NoAH. Two reasons amongst others are that capturing and analyzing inter-AS traffic is not only

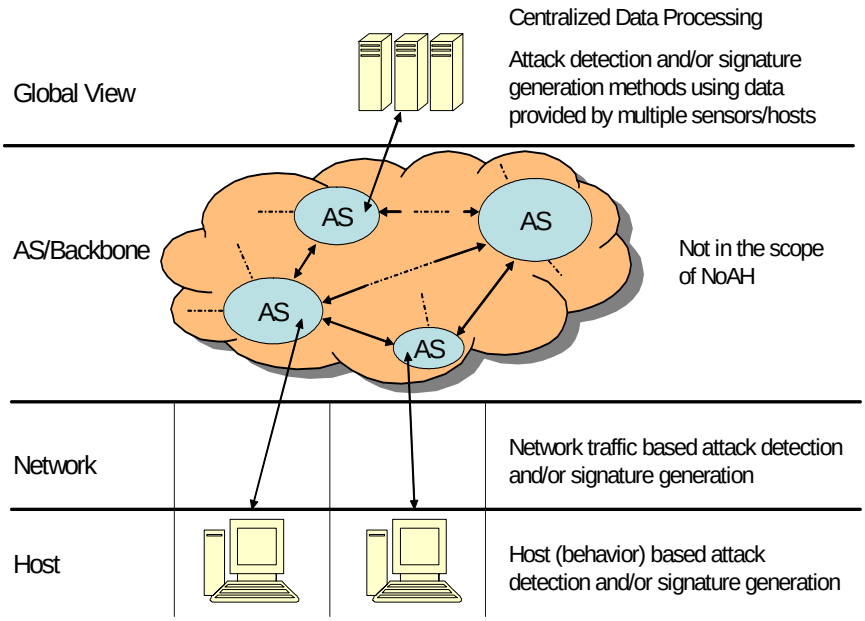


Figure 2.1: Attack detection and signature generation levels

difficult because of technical challenges posed by handling the huge amount of data but also because of legal considerations concerning privacy.

Network: This level comprises actually only the traffic coming from and going to a set of hosts (or a single host) that are logically in the same network. Furthermore, network traffic is captured by a separate device (e.g. sniffer, firewall or router) to counter purposeful tampering of data in case the host gets corrupted. The therewith captured data can then be used for network traffic based attack detection and/or signature generation. In NoAH, as further explained in chapter 6, this level contains sensors that generate information relevant to the attack detection process.

Host: Basically, a lot of data such as e.g. the names of the active processes on a specific host are only accessible on that host itself. Hence, if this kind of information is used for attack detection and/or signature generation, some piece of hardware or software that collects it has to be installed on each host.

Besides the different layers of attack detection and signature generation, a distinction between a local and a remote attacker has to be made. A ”local attacker” can attack the host using input devices like keyboards that are directly attached to it. Hence, the attack can only be detected on host level unless the attacker, after breaking-in, opens connections to other hosts. By contrast, remote attacks are visible on both layers since here the attacker(s) communicate with its target over the Internet or an intranet. As remote attacks are the primary focus of NoAH, this

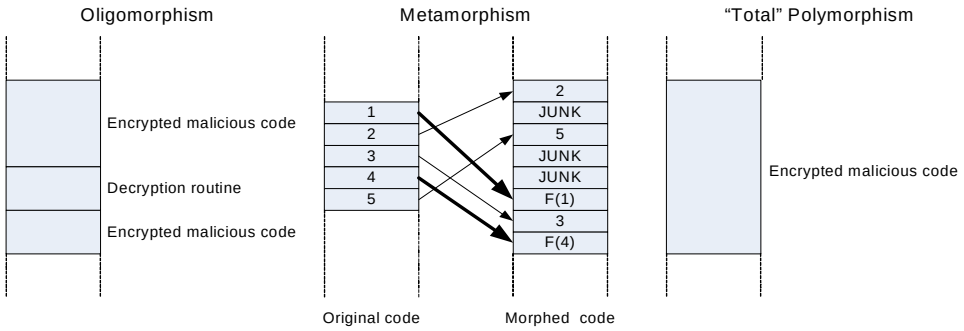


Figure 2.2: Code-morphing techniques: Oligomorphism, metamorphism and "total" polymorphism

report discusses attack detection and signature generation methods considering remote attacks only.

2.2 Polymorphism

To evade network level attack detection methods relying on pattern matching as used e.g. by the intrusion detection system Snort [41] or template matching as used e.g. in [8] [3], new techniques changing that pattern from attack to attack popped up. These techniques have to operate within the design space set by the attack pattern and the injection vector. While the design space specified by the injection vector is often very limited, the design space for the payload is not. Thus, if such techniques are found in the wild, they are almost always applicable to the payload and not to the injection vector.
In this report we use the term polymorphsim to refer to these techniques in general while the terms oligomorphism, metamorphism and "total" polymorphism are used to refer to a specific category of payload-modifications. A similar categorization and a more detailed description for these types can be found in [22]. Figure 2.2 illustrates the different types.

Oligomorphism Encrypting the payload is a way to change its byte pattern. Nevertheless, to decrypt the payload on the target system, a decryption routine has to be sent along with the encrypted payload. But because the decryption routine itself can not be encrypted, this is supposed to leave some room for pattern matching if the decryption routine is not modified from attack to attack. Now, if always the same or only a few different decryption routines are used, the attack payload is said to be oligomorphic. In [14] Crandall et al. show that a conventional decryp-

[3] They apply the concept to application binaries on a machine but it could be applied similarly to code encapsulated in the network traffic.

tor requires at least 10-20 Bytes and therefore still leaves a fingerprint for attack detection by pattern or template matching. But the shorter such a fingerprint is, the higher is the probability that the same byte sequence appears in benign traffic causing false positives.

Metamorphism If techniques to modify the structure of the malicious code without modifying its basic functionality are used, the attack payload is matamorphic. Common techniques are:

- Instruction reordering: Often instructions can be shifted inside a certain frame. One border of the frame is defined by instructions that depend on the result of the instruction to be shifted. The other is defined by the instruction responsible for the result on which the instruction to be shifted depends.
- Block reordering: This technique divides the code into blocks, reorders them and reestablishes the correct execution sequence by inserting appropriate jump/goto instructions. Figure 2.2 illustrates this technique. The function F(x) that apperas in this figure adds the appropriate jump/goto instrutions to block x.
- Junk code insertion: Code that contributes nothing to the attack. It may perform some meaningless stuff or it may even not be executed at all. Figure 2.2 shows the inserted junk code blocks.
- Replacing instructions: The one and the same functionality is almost always achievable using different instructions. A possible procedure is to divide the malicious code into functional blocks[4] and to replace them with other blocks doing the same stuff but by using different instructions.
- Register replacement: Most general purpose computer architectures provide a set of registers to operate on. Moreover, most operations can be executed using any of the general purpose registers. An example: In the IA-32 architecture, the following two code samples have the same functionality but have a different byte-signature:

```
LI EDI, 5;            LI EAX, 5;
MOV EDI, 0008H;       MOV EAX, 0008H;
```

"Total" polymorphism If the attack payload contains N bytes of malicious code and if there exist 2^N variants of the same code that have the same effect on a specific target, it is totally polymorphic. One example where not only the attack payload but the whole attack is totally polymorphic is an attack over an encrypted channel[5]. Another example for (almost) totally polymorphic code is taken from [14] and is shown below (leaving a 2-byte signature only).

[4] This could be done manually by the author of the malicious code

[5] This implies that establishing the encrypted channel is part of the normal operation of the attacked system

The basic idea is to move a randomly chosen value into a register and successively add to it a random value and then a carefully chosen complement and push the predictable result onto the stack, building the shellcode or perhaps a more complex polymorphic decryptor backwards on the stack using single-byte operation:

```
mov eax,030a371ech ; b8ec71a339
add eax,0fd1d117fh ; 057f111dfd
add eax,0b00c383fh ; 053f380cb0
push eax ; 50
add eax,03df74b4bh ; 054b4bf73d
add eax,0e43bf9ceh ; 05cef93be4
push eax ; 50
...
add eax,02de7c29dh ; 059dc2e702
add eax,014b05fd8h ; 05d85fb014
push eax ; 50
add eax,06e7828dah ; 05da28786e
call esp ; ffd4
```

The 2-byte signature is due to the ”CALL ESP” at the end as well as the sequence, ”PUSH EAX, ADD EAX...”. These could be trivially removed respectively by making the last 32-bit value pushed onto the stack a register spring to ESP to use a ”RET” instead of ”CALL ESP”, and by using different registers with a variety of predictable 8-, 16-, and 32-bit operations, leaving no byte string signature at all.

Chapter 3

Review of existing Signature Generation Approaches

This chapter presents a review of existing attack detection and signature generation approaches that are believed to be relevant for NoAH. We sorted the approaches according to the attack detection method they use. In section 3.1 we review approaches that do not apply any attack detection mechanism prior to signature generation, i.e., all observed traffic is used for signature generation. Approaches which apply network-based attack detection, such as scan-detection, prior to signature generation are presented in section 3.2. In section 3.3 we review approaches which use host-based attack detection methods such as memory tainting. Finally, section 3.4 describes approaches which apply both, host- and network-based attack detection.

3.1 Approaches without Attack Detection

3.1.1 Honeycomb

Honeycomb [24] was the first approach aiming at the automated generation of attack signatures. It is built as an extension to honeyd, a low-interaction honeypot. Honeycomb does not make a distinction between normal and benign traffic. Hence, all traffic (incoming and outgoing) that is seen on a honeypot is used as input for the signature generation algorithm.

In general, Honeycomb's signature generation mechanism is based on pattern detection techniques: All incoming network traffic is compared to traffic previously seen on the honeypot. This requires Honeycomb to store information about previous connections on the honeypot: Honeycomb stores the reassembled message payloads for UDP connections (packets exchanged between the same IP addresses and port number pairs) and TCP connections up to a maximum number of bytes.

If a new packet is received by the honeypot, an empty signature record is generated. Afterwards protocol analysis is performed on the network and transport

level. Detected header anomalies, such as unusual TCP flag combinations, are recorded in the signature. Next, the received packet is assigned to an already stored flow/connection with identical source and destination IP addresses and ports (if exists), and its payload is reassembled, i.e. packet payloads are concatenated. Finally, the connection to which the received packet belongs is compared to all other currently stored connections.

For each pair of connections destination ports are compared. If an overlap is detected, the signature record is cloned and becomes specific to the two compared connections. Subsequently, the longest common substring (LCS) algorithm is applied to the pair of reassembled connections in two ways: horizontally by comparing the received message n to the nth message of the second connection, and vertically by comparing the concatenated payloads of two connections. If a common substring of minimum length is found, the substring is added to the signature. If the signature contains no facts at this point, the processing for this packet is finished. Otherwise, the signature is used to improve (new signature is superset of an existing signature) or extend the signature pool. Hence, Honeycomb generates network level signatures which are continuous byte patterns enhanced with protocol analysis information if available.

A prototype implementation of the Honeycomb system was tested during 72 hours on a cable modem connection. The authors' evaluation of their approach is rather incomplete. They claim that their system created 25 signatures containing flow content strings, i.e. for the Slammer and CodeRedII worms. However, they provide no evaluation for the most critical characteristic, the false positive rate, of their approach.

3.1.2 Polygraph

Polygraph [30] is a system specifically targeted at generating signatures for polymorphic worms. A flow classifier reassembles flows per port and puts them in a suspicious flow pool. However, Polygraph does not specify the classifier to be used. Therefore, we classify it as not using any attack detection mechanism.

The authors define three different signature types which are specifically designed to detect polymorphic worms. All these signatures are built from substrings, called tokens. *Conjunction signatures* consist of a set of tokens, and match a payload if all tokens in the set are found in it, in any order. *Token-subsequence signatures* consist of a ordered set of tokens, and match a payload if it contains the sequence of tokens in the same ordering. *Bayes signatures* consist of a set of tokens, each associated with a score, and an overall threshold. The signature matches a payload if the sum of all present token scores is over the defined threshold.

Signatures are generated for flows in the suspicious flow pool. This flow pool may contain misclassified flows which are actually not worms, as well as different flows exploiting the same protocol. In a first step, all distinct substrings of a minimum length (tokens) that occur in at least K out of the total n samples in the suspicious pool are extracted. After token extraction, each suspicious flow can be

represented as a sequence of tokens, and the rest of the payload can be removed. This is identical to a conjunction signature for all flows in the suspicious pool. To generate a token-subsequence signature of all flows in the suspicious pool, an ordered sequence of tokens that is present in each flow sample must be determined. An adaption of the Smith-Waterman algorithm is used to find an ordered token sequence preferring subsequences with contiguous substrings. Bayes signatures are generated by analyzing the tokens present in a suspicious flow pool with a simple Bayes classifier. This classifier calculates for each token occurring in a sample the probability that this token is present in a worm flow, and in an innocuous flow respectively. The threshold associated with the signature is then chosen to minimize the false positive rate in the innocuous flow pool, and to minimize the false negative rate in the suspicious flow pool.

The authors show that the Bayes signature generation algorithm can be used unmodified in case multiple worms and/or innocuous flows are present in the sample. For the other two signature types, the innocuous and multiple worm flows must be clustered first in order to generate meaningful signatures. The authors suggest to use a hierarchical clustering method. Beginning with one cluster and signature for each flow, these specific clusters are iteratively merged. After each merge, the signature generation mechanism is run again on the combined set of flows to produce a new signature. To decide which flows to merge, for all possible combinations the false positive rate of the resulting signature in the innocuous flow pool is determined. The pair of clusters that results in the lowest false positive rate is merged next. This process is repeated until the resulting signature would result in an unacceptably high false positive rate, or when only one cluster remains.

Polygraphs performance is evaluated for three scenarios: 1) the flow pool contains only flows of one worm, 2) the flow pool contains flows of one worm and innocuous flows, and 3) the flow pool contains flows of multiple worms as well as innocuous flows. Polygraph was evaluated using several network traces as input. For the HTTP experiments, a 5-day trace taken from the perimeter of Intel Research Pittsburgh in October 2004 was used as innocuous HTTP flow pool, and a 10-day trace taken 10 days after the end of the first trace was taken as evaluation trace for the generated signatures. For the DNS experiments, a 24-hour DNS trace taken from a university DNS server was used. The authors generated polymorphic versions of three real-world exploits, two exploits using the HTTP protocol and one using the DNS protocol, which were used to build the suspicious flow pool. The authors found that the quality of the generated signatures, i.e. the number of false positives and false negatives they generate, depends on the number of worm samples present in the trace.

3.1.3 Earlybird

Earlybird [39] proposes an automated approach for quickly detecting previously unknown worms and viruses. Detection is based on two key behavioral characteristics - a common exploit sequence, and a dispersed range of unique sources.

Earlybird measures the prevalence of all content entering a network. Only in a second step, address dispersion is used to reduce the false positive rate of the system. Therefore, we classify Earlybird as an approach that is not using attack detection.

For each incoming packet, Earlybird computes a variant of Rabin fingerprints for all possible substrings of a certain length. Each computed fingerprint is hashed together with the destination port and protocol. These hash values are then stored in a prevalence table. For each table entry a counter, a list of unique sources, and a list of unique destinations is maintained. The authors refer to this approach as *content sifting* since sorting this table on the substring count and the size of the address lists gives the set of likely worm traffic. To reduce the false positives, and to distinguish real worm traffic and content that frequently occurs between two computers, address dispersion is additionally considered. For estimating the address dispersion a so-called *scaled bitmap* is used. Scaled bitmaps leverage the fact that during an outbreak address dispersion increases continuously, and reduce memory requirements about five times compared to existing algorithms. Finally, the system generates pattern-matching signatures formatted for the Snort-inline intrusion detection system.

A prototype implementation of the Earlybird system showed that under high traffic loads, the CPU cannot handle computation of Rabin fingerprints for all packets anymore. The authors suggest a method called *value sampling* to cope with this problem. With value sampling all strings whose fingerprints match a certain pattern are ignored, instead of doing random sampling. Performance of the Earlybird prototype was tested with data that was mirrored from a campus router (managing traffic to 500 hosts) to the Earlybird sensors. For measuring false positives and false negatives generated by the system, traces were used additionally. Apart from generating signatures for the Slammer, and Blaster worms, the system also generated numerous false positives, mostly for distributed port scans, common protocol headers, spam email, and also popular Bittorrent files. Testing the same trace with snort-inline with a rule-base of 340 signatures revealed no additional alerts. Hence, the authors conclude their system did not produce any false negatives.

3.1.4 Nemean

Nemean [50] provides automatic generation of intrusion signatures from honeypot packet traces. The authors argue that most of the traffic observed at a honeypot is malicious, and the negligible amount of misclassified traffic can be separated easily from the malicious traffic. However, they do not specify the details of such a classification.

The Nemean system consists of two components: the Data Abstraction Component and the Signature Generation Component. Packet traces entering the Data Abstraction Component are normalized first, i.e. fragmentations, invalid packets, and retransmissions are removed giving a consistent packet trace. Second, flow aggregation is performed by ordering the packets into connections (multiple packets between two hosts) and sessions (multiple connections between two hosts). In

a last step, a pre-defined service specification is used to normalize the aggregated sessions. The authors developed incomplete service specifications for HTTP and NetBIOS/SMB. The output of this component, a semi-structured session tree, is used as input for the Signature Generation component. The first step of the Signature Generation component is to group sessions and connections with similar attack profiles according to a similarity metric. On-line star clustering is used as clustering algorithm, and two different similarity metrics, cosine similarity and hierarchical edit distance, are implemented. Then automata learning is used to construct an attack signature from a cluster of sessions or connections. Finally, these Finite-State-Automata signatures are transformed into specific signatures for one or more target intrusion detection systems. The authors suggest simple sanity checks, which are not further specified, to exclude clusters of irrelevant sessions from the signature generation process.

The effectiveness of Nemean's HTTP and NetBIOS signatures was tested with a) two data traces from two unused /19 IP address blocks that was redirected to a honeypot environment, and b) productive HTTP traffic traces from the border router of a /16 network. Cluster quality was evaluated quantitatively by manually tagging sessions with known attack types, and measuring two common metrics, precision and recall, for varying similarity thresholds. Moreover, the authors evaluated Nemean's detection and false alarm rate by comparing it to Snort. For evaluating the detection rate, one honeynet data set was used to generate the signatures, and another data set from the same network was used to test the generated signatures. They claim their system has a HTTP-based attack detection rate of 99.9%, whereas Snort detected only detection only 99.7% of the attacks. However, the authors do not explain one very important fact: How do they get to know the actual number of attacks present in the data set. This also makes the author's claim of a zero false positive rate somewhat questionable.

3.2 Approaches Using Network Level Attack Detection

3.2.1 Autograph

Autograph [23] is a system for automated generation of worm signatures. The authors restrict their investigation to worms that propagate over TCP. The system uses heuristics, more specifically a port scanner detection mechanism, to classify incoming traffic as either suspicious or non-suspicious. Scanning detection is implemented by observing inbound unsuccessful TCP connections. Each external host that has made unsuccessful connection attempts to more than x internal IP addresses is considered to be a scanner. Hence, one can say that Autograph will only generate signatures for worms that propagate by randomly scanning IP addresses. This detection mechanism is also inapplicable when port scanners use spoofed source addresses. To address this fact, the authors plan to use different anomaly detection techniques in the future.

Autograph performs TCP flow reassembly for inbound payloads in the suspicious flow pool. Moreover, all flows in the suspicious flow pool are sorted regarding their destination port. The signature generation process is initiated when the suspicious flow pool contains more than a threshold number of flows for a specific destination port. During signature generation, Autograph measures the frequency with which non-overlapping payload substrings occur across all suspicious flow payloads, and proposes the most frequently occurring substrings as candidate signatures. To do so, each flow's payload is divided into variable-length content blocks using COntent-based Payload Partitioning (COPP), and the number of suspicious flows in which each content block occurs is counted. The authors refer to this count as the content's prevalence. The COPP mechanism, first introduced in the file system domain, computes a series a Rabin fingerprints over a sliding window of the flow's payload to generate the content blocks. Content blocks that appear only in flows originating from a single source IP address are discarded. The authors found that these content blocks are often caused by misconfigured systems which are not malicious. Moreover, byte strings which are known to cause high false positive rates can be blacklisted by a local administrator. The remaining content blocks are used for signature generation. In a repetitive process the most prevalent content block is selected as signature, and subsequently all flows in which this content block was found are removed. The process repeats with the remaining flows, until some fraction of all flows in the pool has been covered. At the end of this process, Autograph reports the set of selected signatures in Bro's signature format.

The authors present a detailed evaluation of the signatures generated by Autograph. They omit a description of their prototype implementation, but from the conducted experiments one can guess that a prototype exists. First, they investigated the effect of content block size on the quality of generated signatures. In the experiment, Autograph is fed with packet traces from the DMZ of two research labs (2^9 IP addresses each) which contain the full packet payloads. To measure Autograph's true positive rate the same trace was tested first using Bro with well-known signatures for the scanning-based HTTP worms Code-Red, Code-RedII, and Nimda, and then using Autograph's signatures. To measure the rate of false positives, a sanitized trace was created by removing all flows from the trace which were previously identified by Bro as worms. Then Bro with Autograph's signatures was run on the sanitized trace. The experiments show that the performance of generated signatures varies for different parameters, namely the fraction flows covered by the generated signatures, and the content block size. The authors admit that the optimal parameters found may not apply to other traces. Moreover, the authors claim that Autograph can generate very short signatures for worms with limited polymorphism (fixed 56-byte sequence). However, the authors are aware that such short signatures might cause high false positive rates. Secondly, Autograph's distributed signature detection approach is evaluated, measuring how quickly Autograph detects and generates a signature for a newly released worm. For these measurements, a Code-RedI worm was simulated according to Moore et

al. [28]. Autograph monitors were placed randomly at 1% of the ASes that include vulnerable hosts (63 monitors in total). The authors claim that the first monitor in this simulation (a flow is classified as suspicious after the first scan) detects 5 worm payloads before less than 1% of the vulnerable population has been infected. In order to evaluate the speed of signature generation under consideration of the false positive rate of generated signatures, additional trace-driven tests were conducted. The results show that there exists a region of operation where the system generates signatures that do not cause any false positives in a timely fashion.ed with packet traces from the DMZ of two research labs (2^9 IP addresses each) which contain the full packet payloads. To measure Autograph's true positive rate the same trace was tested first using Bro with well-known signatures for the scanning-based HTTP worms Code-Red, Code-RedII, and Nimda, and then using Autograph's signatures. To measure the rate of false positives, a sanitized trace was created by removing all flows from the trace which were previously identified by Bro as worms. Then Bro with Autograph's signatures was run on the sanitized trace. The experiments show that the performance of generated signatures varies for different parameters, namely the fraction flows covered by the generated signatures, and the content block size. The authors admit that the optimal parameters found may not apply to other traces. Moreover, the authors claim that Autograph can generate very short signatures for worms with limited polymorphism (fixed 56-byte sequence). However, the authors are aware that such short signatures might cause high false positive rates. Secondly, Autograph's distributed signature detection approach is evaluated, measuring how quickly Autograph detects and generates a signature for a newly released worm. For these measurements, a Code-RedI worm was simulated according to Moore et al. [28]. Autograph monitors were placed randomly at 1% of the ASes that include vulnerable hosts (63 monitors in total). The authors claim that the first monitor in this simulation (a flow is classified as suspicious after the first scan) detects 5 worm payloads before less than 1% of the vulnerable population has been infected. In order to evaluate the speed of signature generation under consideration of the false positive rate of generated signatures, additional trace-driven tests were conducted. The results show that there exists a region of operation where the system generates signatures that do not cause any false positives in a timely fashion.

Lastly, for accelerating the accumulation of worm payloads, Autograph is extended with a mechanism for sharing suspicious source addresses among all monitors. The so-called tattler protocol is an extension the RTPC protocol, which is used to control multimedia conferencing sessions and has been shown to scale to thousands of senders. The tattler protocol is used to allow monitors to announce the IP address and destination port of scans they received, where each announcement contains between one and 100 port scanner reports. Simulations show that the peak bandwidth consumed by tattler during a Code-RedI epidemic is only 15Kbps, not including the background port scanning.

3.2.2 PADS

Position-aware distribution signatures (PADS) for worm detection were introduced in [42]. The authors use a double-honeypot system to track malicious activities in local networks. All connections made from a high-interaction inbound honeypot are redirected to a low-interaction outbound honeypot. The idea behind this is that a compromised honeypot will sooner or later start to infect other systems. Hence, multiple variants of the worm code can easily be obtained on the low-interaction honeypot. However, the drawback of this approach is that an attack can only be detected in case the high-interaction honeypot is vulnerable to the attack.

The authors introduce a novel relaxed, inexact form of signatures which inherit the positive aspects of both, signature-based and anomaly-based systems. They claim that position-aware distribution signatures (PADS) provide the necessary flexibility to cope with certain forms of polymorphism. In contrast to traditional signatures, PADS signatures include a byte frequency distribution (instead of a fixed value) for each position in the signature string. The PADS signature is computed from a collection of worm variants captured by the double-honeypot system. The authors claim that, provided all captured byte sequences are indeed variants of one worm, the signature that maximizes the matching scores of all worm variants can be computed if the significant regions of the variants are known. The significant region of a byte sequence, is the subsequence that maximizes its matching score. For computing the significant regions of all worm variants the expectation-maximization (EM) algorithm, and the Gibbs sampling algorithm are used. Just briefly, the iterative EM algorithm guesses the starting position of significant regions, computes the signature, uses the signature to compute the new starting positions, and repeats this process until convergence is reached. To solve the known problem of the EM algorithm to get stuck in local maxima, additionally Gibbs sampling is applied. The resulting PADS signature contains the byte frequency distribution of legitimate traffic, and the identified anomalous signature. Applied in a detection system, an alarm is raised if a byte sequence has a higher matching score for the anomalous signature than for the normal signature. The authors, however, do not specify how to obtain the normal signature for legitimate traffic.

The effectiveness of PADS signatures for detecting polymorphic worms has been evaluated by the authors. For the experiments, variants of the MS Blaster payload were artificially generated using polymorphism techniques. More specifically, instruction substitution is performed on 10% of the malicious payload, and garbage payloads are inserted at different random locations accounting for 10% of the total payload. In a first experiment, the matching scores for the EM and Gibbs sampling alorithm are computed for 100 such variants of the MS Blaster worm. The results show that Gibbs sampling generates higher scores. A second experiment tests the quality of the generated signatures. Therefore, 200 MS Blaster variants are generated. Half of the variants were used for signature generation, and the remaining 100 were mixed with normal-traffic bytes sequences to obtain a test trace. In the experiment, the PADS signature identified all new variants of the worm without

false positives. Moreover, they compare their approach with the longest common substring method and position-unaware byte frequency distribution signatures.

3.2.3 PAYL

PAYL [48] is an anomaly detection sensor that detects inbound anomalous payloads, and correlates them with outgoing traffic on the same ports. It uses unsupervised machine learning techniques to model the normal traffic profile of a host. The authors argue that the content data of a zero-day attack will differ significantly from normal data. During the training phase, PAYL computes the statistical distribution of n-grams. An n-gram is a sequence of n adjacent bytes in a packet payload. Since PAYL uses 1-grams in its current version, it actually computes the byte value distribution for each packet. In fact, the byte distribution is computed per port and per packet length. During the detection phase, the Mahalanobis distance is used to calculate the similarity of new data against the pre-computed profile. A threshold for the detection is computed automatically in the training or calibration step.

For the purpose of worm propagation detection and signature generation, the authors extended their system to model both inbound and outbound traffic. The argumentation is the following: If PAYL detects anomalous egress packets to port *i* which are very similar to anomalous ingress packets to port *i*, there is a high probability it is a worm. Hence, the authors reason that their approach can generate a worm signature on its very first propagation attempt. To render this inbound-outbound correlation possible, the content of all anomalous inbound packets is stored in a buffer to which all outbound anomalous packet contents are compared. We reason that, depending on the amount of traffic a sensor receives, limited buffering capabilities could prevent slower worms from being detected by PAYL. As comparison metric, PAYL uses string equality, longest common substring (LCS), and longest common subsequence. The identified byte sequences can then be used as signatures. Finally, all outgoing packets, which have a similarity score that is higher than a given threshold, are blocked or delayed to prevent further spreading of the attack.

The authors evaluated both the worm detection and the signature generation part of their system. For the evaluation of PAYL's worm detection capabilities, a prototype was tested with three real-world data sets containing known worms. To achieve ground truth for the calculation of false positives and false negatives, all known worms detected by Snort were removed from the data set. Furthermore, each data set was split into two parts, one for training and the other for testing. A set of known worms, including CodeRed and CodeRedII, was inserted at random places in the test data. The results show that PAYL produces only very few false positives. For testing the quality of the signatures generated by PAYL, CodeRed and CodeRedII packet traces, which were obtained in a controlled environment, were merged into the three test data sets. The authors claim that the generated signatures do not cause any false positives on the three test data sets. The question

is, however, if PAYL requires an offline training phase on cleaned data - how will it be able to react if the normal traffic profile changes, and will such a change cause more false positives.

To further improve the false positive rate of PAYL, the authors suggest an interesting approach for correlating signatures generated at different sites. Since different site have different normal payload models, it is expected that they generate different false positives as well. Hence, a correlation between multiple sites would allow to extract the true worm payloads. A prototype of this sharing infrastructure was implemented in the Worminator testbed. Moreover, privacy issues are addressed by exchanging not the actual byte distributions, but an ordered frequency distribution, called Z-string. For correlating these Z-strings, the authors suggest to use the Manhattan distance or the LCS of the Z-strings.

3.3 Approaches Using Host Level Attack Detection

3.3.1 COVERS

The COntext-based, VulnERability-oriented Signature(COVERS) [26] system allows to automatically generate attack signatures for control flow hijacking attacks. These signatures can then be used to filter out future occurrences of these attacks in order to preserve the integrity of the service and to improve its availability.

COVERS consists of an attack detection part and a signature generation part. The attack detection part employs the Address Space Randomization(ASR) technique provided by PaX [32]. ASR is used to relocate memory object from their default position to a randomized one. Objects such as libraries and tables with pointers to important operating system functions are often stored on default positions. Consequently, if ASR is used, an attacker has to guess, where a specific piece of code is stored in memory. If he misses the correct location, (usually) a memory access violation error is triggered. But according to [1] and [38], repetitive attacks can defeat probabilistic protection techniques such as address-space and instruction set randomization. A big advantage of ASR over techniques such as dynamic taint analysis (see section 3.3.6) is its low processing and memory overhead. Nevertheless, the authors stress that other techniques such as StackGuard [12] or complete memory-error protection(e.g. [49]) could be used with their signature generation mechanism too.

Signature generation consists of three steps. The correlation step identifies the specific network packet (or flow) involved in an attack, and the bytes within this packet that were responsible for triggering the alert. They argue that all memory error exploits reported so far have been based on pointer corruption, and that the value used to corrupt the pointer must be included in the attack input. Concerning format-string attacks they state that this type of attack shares the key characteris-

tic ”pointer corruption”. There, the forged pointer (i.e., attacker specified pointer value) occurs in the middle of attackerprovided data residing in a buffer.

Based on this observation, a forensic analysis of the victim process memory around the corrupted pointer is proposed to identify the relevant bytes. The forensic analysis consists of using the longest common string method with recent input and the data in memory. The authors identify some cases for which this approach is not likely to produce a meaningfull signature in the end:

1. The input message containing the longest common string doesn't have to be the input message containing the attack. The shorter the longest common string, the higher the probability for this case to occur.

2. In the presence of character encoding/decoding or encryption/decryption of input data (e.g. URL encoding used by HTTP), there is no point in comparing data in memory with input data. They will not match.

3. Integer overflows provide too few data to get a useful result from the longest common string search (too many matches).

4. Data pointer corruption. The attack is detected, if an operation uses the corrupted data pointer. If the pointer is read from a register, data flow analysis would be necessary to identify its location in memory. To mitigate this problem the authors make the following assumption: Most attacks rely on vulnerabilities in commonly used library functions: Heap overflows rely on a section of code in heap management functions (malloc family) and format-string attacks rely on a code section within vfprintf. For these two vulnerabilities, the authors analyzed the data flow in order to determine the origin of the data in the register. Based on this analysis, they constructed a table that enables to map a vulnerability to a method on how to calculate the location of the register's content in memory. A vulnerability is identified by a byte pattern consisting of the instructions preceding the one finally triggering the memory exception. Hence, precedure makes the signature generation process library/application specific.

5. The first ”corrupted” pointer points with a certain probability to a valid memory location containing arbitrary code or even to the injected code, if the attacker is able to place it accordingly. Execution of this code is likely to lead sooner or later to a crash. If the crash is not due to an invalid memoy access [1], it is not possible to proceed to the next step since no relevant bytes could be identified. Even if the crash was due to an invalid memory access, it is very likely that it occurred at a location different from those of the ”corrupted” pointer. Therefore, it is unlikely that any useful data is found in the surroundings. To estimated the probability of this scenario, the authors calculate the probability p that an attacker accesses a valid memory address as

[1] E.g. due to a division by zero

follows: $p = UsedMemory/LogicalAddressSpace$. Hence, generating meaningfull output requires an average of $1/(1-p)$ attacks (the process is restartet after the crash).

In a second step, the input context is identified. An input context is a particular field of a specific type of message. The field is identified using the results from the previous step together with a message format analyzer. Because of time- and resource constraints, only simple message format specifications (consisting of a few lines for most services) are used. To define such a message format, a language that extends regular expressions to support binary protocols was developed.
After the field has been identified, the characteristics of the underlying vulnerability are extracted. This is done by checking if the field is excessively long or if there is binary data within a text-valued field. The reference values are continuously updated using benign input traffic. Finally, the generated signature consists of the message format specifier, the message field carrying the exploit and thresholds for the characteristics.
To evaluate their work, they did a performance, a false positive and a polymorphism analysis. The false positive analysis consisted of a manual inspection of the source code with the goal to determine if a match always implies a successful attack. The results of the performed evaluations were a very small performance overhead (around 10%), a low false positive rate, and that catching polymorphic attacks is very likely.

The contributions of the presented approach are its low performance overhead and the signatures that are suitable to filter polymorphic attacks. Nevertheless, the signature generation process contains steps that are application/library specific while the quality of the signatures is likely to depend on the accuracy/quality of the message format specifications. Since they used simple specifications only and a few common attacks to evaluate their approach, the results of the false positive test is not representative. Another question is what happens, if the correlation step was not successful and hence does not point to the message containing the attack payload [2]. The authors claim that even in this case, a meaningful signature can be generated. They proposed to generate it by parsing all recent input and by searching for abnormal field-characteristics. But to be able to decide if the fall-back mechanism has to be used, it is necessary to detect if the correlation step failed[3].

3.3.2 DIRA

The attack Detection, Identification and Repair(DIRA) [40] system is basically an extension to GCC [43] that enables the therewith compiled programs to detect and identify control-flow hijacking attacks. Moreover, there is a good chance, that the attacked program can recover from the attack without having to restart it. To mit-

[2] see the above examples on how misleading output could be generated
[3] Looking at the cases where correlation mentioned

igate denial of service attacks based on sending the same attack packet again and again, the system can send the network traffic responsible for the attack to an IDS. The IDS could then generate an appropriate filter to prevent future similar attacks.

In a first step, the control-flow hijacking attack has to be detected. DIRA does this by instrumenting control-sensitive data structures in a way that disallowed changes can be detected. More specifically, any application needs to be recompiled in order to insert instrumentation code for the sensitive data structures. To simplify implementation, the authors focused on the ability to check if a return address has been overwritten during a function call and on checking if a function pointer has been altered using direct assignments [4]. If this happens, the attack needs to be identified. Attack identification consists of tracking down the input that is responsible for triggering the attack detector. This is done using a memory update log for global and static variables and for memory transactions in a proxied set of libc functions[5].
The last step is the attack repair step. Its goal is to roll-back the application state to a point before the input containing the attack was read and to restart the application from there. Since a memory log is already used for attack identification, a roll-back does not impose a large additional overhead. Undoing the logged memory modifications is sufficient. Afterwards, the restart point has to be selected ad the last common ancestor from the function receiving the attack input and the function where the attack was detected.
The evaluation was focused on performance measurements for five server applications [6]. For those, the maximum increase in compile time was around 555%. Furthermore, the maximum processing overhead for running the modified application compared to the unmodified one is around 60%. Additionally three of the applications [7] were attacked using one exploit each. Even though all attacks were detected and could be prevented, the repair step worked only for two of the three attacked applications.

A major drawback of DIRA is its dependency on the availability of the source code of the application or library to protect. Weaknesses are that only a few control-sensitive data structures are supervised and that only a limited amount of functions are instrumented. Additionally the memory log tracks direct assignments only and can not be used to identify dependencies that involve any arithmetic expression (e.g. B=A+C). Consequently, the attack input can not always be identified. In order to track such dependencies, a complete data-flow analysis that causes a large processing overhead would be necessary. There are some other problems in case a process forks [8] and the restart point is before the forking point.

[4] Altering a function pointer using memcpy() an example for an indirect assignment

[5] e.g. memcpy(), gets(), read(), fork(),...)

[6] ghttpd, drcatd, named, qpopper, proftpd

[7] named, ghttpd, drcatd

[8] Starts a process (child process) while the starting process (parent process) continues to run. The child process gets a copy of all of the data of the parent process (including e.g. file pointers to

Since the evaluation focuses on some performance measurements, it does not offer enough data to make a sound statement about repair capabilities or false negatives/false positive rates.

3.3.3 DOME

The Detection Of Malicious Executables (DOME) [36] approach allows to detect code injection attacks and attacks originating from executables that were modified with obfuscated code or contain dynamically generated code. The detection mechanism is based on the fact that malicious code often makes use of system calls and is either injected into a running application or tries to obfuscate its presence if included in an executable. More precisely, DOME makes a static analysis of the executable of an application to identify the location of system calls in it and supervises if the locations at runtime differ. Although DOME does not generate any signature describing malicious activities, it can recognize such activities nevertheless by verifying any activity against a signature describing normal/approved behavior. The following three statements characterize the coverage of their approach:

- Any injected code is assumed to be malicious.
- Dynamically generated or obfuscated code containing system calls that are hidden from static analysis is malicious.
- To interact with the operating system, the code uses the Win32 API[9].

In their proof-of-concept implementation, the static analysis is done with the IDA Pro disassembler [17] and handles applications consisting of one executable component only. Moreover, late bound code[10] is not handled by the static analysis step. After the static analysis, DOME uses the Detour wrapper package [21] to monitor Win32 API calls. This wrapper package enables to execute custom code before and after transferring control to the actual Win32 API. With that, it is possible to report call from a location not registered during the static analysis to a logging facility. In the evaluation part, the authors evaluated the false positive and false negative rate of DOME. For the false positive test, a set of eight benign executables was tested with DOME. The only false positives were due to dynamic binding of APIs. For the false negative test a set of six executables containing a virus or worm were run with DOME. The test revealed no false negatives.

Since the authors only implemented a proof-of-concept study, there are some open

opened files). Each process has its own memory and is able to run self-contained. If the parent process is terminated and the child process is still running, the child process becomes a so-called zombie process.

[9] The Win32 Portable Executable File Format is the standard executable format for Windows 2000 and above. This file format usually uses the Win32 API to interact with the operating system.

[10] e.g. dynamic link libraries (dll)

issues. One is the bypassability of the monitoring functionality (wrappers). Therefore the authors propose a kernel-level authentication ensuring that the APIs are only reached after passing through a unmodified wrapper. Since such modifications are not an option for closed-source Software like Windows, kernel-level authentication is likely to remain a rather theoretical solution. Another one is that because the static analysis does not handle dynamically loaded program-parts, it is not yet a solution for real-world software. A final open issue is an evaluation using a large and representative test set.

3.3.4 Minos

The Minos [14] system is basically a modification to the Pentium architecture and to the operating system kernel that allows to stop control data attacks by tagging untrusted input data as tainted, and by propagating these tags through filesystem operations, the main memory and the processor pipeline. In this work, the authors focused on the detection part and did not propose any signature generation mechanism. The reason why Minos was included in this review of signature generation approaches is its new approach in attack detection and its implications for the type of information that is available to a signature generation process. In fact, Minos enables to detect whenever tainted data is used as control data in a control-flow transfer. This is also known as dynamic taint analysis. Minos considers any data which may be loaded into the program counter, or any data used to calculate such data [11] as control data. Hence, Minos is able to detect attacks based on vulnerabilities such as buffer overflows, format string vulnerabilities, or double free()s [2].

A major problem of the dynamic taint analysis is that it causes a large processing and memory access overhead: For each operation (e.g. of type "'z=x op y"'), it has to be checked if the data source(s) (x and y) are tainted to decide if the destination (z) needs to be tainted too. Minos reduces this overhead by mapping the tag propagation functionality of dynamic taint analysis to special purpose hardware. More specifically, it applies Biba's low-water-mark integrity policy [12] to the inputs of every instruction operation in order to determine the integrity of the result. Additionally, whenever an attempt is made to transfer control flow with low integrity data, the hardware triggers an interrupt. But because developing and producing the proposed hardware extension is out of the scope of Minos, the proof-of-concept prototype does not dispose of the proposed modifications to the Pentium architecture. Instead the modified Pentium architecture was emulated using a customized version of the Bochs Pentium emulator [25]. The remaining functionality, the taint-

[11] Including return pointers, function pointers, jump targets, variables such as the base address of a library and the index of a library routine within it used by the dynamic linker to calculate function pointers.

[12] Biba's low-water-mark integrity policy specifies that any subject may modify any object if the object's integrity is not greater than that of the subject, but any subject that reads an object has its integrity lowered to the minimum of the object's integrity and its own. For details see [3]

ing(tagging) of untrusted input and the alert handling, is implemented in the operating system kernel. Minos defines trust on the basis of how long the data has been part of the system. In Minos, the kernel keeps a timestamp called the establishment time before which all libraries and trusted files were established and after which everything created is treated as vitriol and forced low integrity. Static binaries can be created after the establishment time and are trusted for their own control flow and that of their children by being marked high integrity when the executable ELF binary is mounted. Any communication where one process passes data to another process which is not sharing its memory space will be forced low integrity, because it will go through the virtual file system through an inode[13] that was either established or modified sometime after the establishment time. Thus when an attacker's data comes from the network it will stay low integrity in the system even if it goes out to disk and comes back. There is no need to modify the filesystem on the hard drive. More specifically, the read() system call forces the data read by the process to be low integrity unless both the ctime (time of last inode change) and mtime (time of last modification) of the inode are set to a time before the establishment time of the system. A separate Minos implementation for Windows XP marks data as low integrity when it is read from the Ethernet card device, but the integrity information cannot be tracked in the filesystem since the Windows XP source code is not available.

The authors evaluated their work by attacking Minos with eighteen different control-flow hijacking attacks (see [13] for details). All of them were caught while no false positives were reported. Aside from the security related evaluations, the authors assessed the performance overhead of the functionality that needed to be implemented in the operating system kernel. More specifically, the overhead incurred by saving the tag bits during virtual memory swapping was measured[14].

The biggest advantage of Minos over other dynamic taint analysis approaches is its virtually low performance overhead. Nevertheless, because the necessary hardware extensions do not exist, the advantage is only a virtual one. Furthermore, there are limitations as to Minos' ability to catch more advanced control data attacks designed specifically to subvert Minos, mostly related to the possibility that an attacker might be able to arbitrarily copy high integrity control data from one location to another. Minos only stops low level control data attacks that hijack the control flow of the CPU and was not designed to catch higher-level attacks involving, for example, scripting languages. Moreover, kernel level attacks are not

[13] An inode is a structure that stores information about objects in the filesystem, such as files, pipes, or sockets

[14] The Minos hardware tracks tagged data in main memory only. Because information about memory chunks swapped-in or swapped-out is not available at this level, the operating system has to take care of this. It saves the tags for swapped-out memory chunks and restores them in case they get swapped-in again

caught because the kernel adds the tags only when it copies data into a user process' memory space.

3.3.5 Paid

The Program semantics-Aware Intrusion Detection system (Paid) [9] defends applications against control flow hijacking attacks that make use of system calls. It employs the source code of an application to construct an accurate system call model which takes the following three characteristics into account: Location and ordering of system calls and parts of the control flow of the application. By checking run-time system call patterns versus the model, abnormal behavior can be detected. It follows, that this approach does not generate signatures for malicious behavior but a "'signature"' for approved behavior. Similar approaches like e.g. the PDA model proposed by Wagner and Dean's [46] employ more precise but expensive models while policy based models like systrace [35] are less precise but efficient. Paid is sufficiently accurate and efficient as its run-time overhead for protecting an application is around 10%.

To protect an application with Paid, the following steps are necessary. First, the application's source code as much as the source code of the used libraries has to be available. Only then the recompilation step can analyse the system call usage, construct a System Call Site Flow Graph (SCSFG) and include it in the resulting library or executable. The SCSFG is actually a Deterministic Finite-state Automaton (DFA) representing the system call sequences and their location (site) in the program. To get a DFA from the source code, the authors eliminate non-determinism in the control flow (introduced e.g. by if..then..else constructs or function pointers) by apply concepts like system call inclining, graph inclining and custom system call insertion. In a second step, the recompiled application is started and a pointer to the active SCSFG node keeps track of the executed system calls. If the program determines and accesses a function from a dynamic library at run-time only, the SCSFG for the accessed function is dynamically inserted into the main SCSFG.

Even though the evaluation contains some proof-of-concept attack detection tests, its focus is on performance measurements. For the attack detection test, two small custom programs and one real world exploit were used. One of the custom programs allows to overwrite the return address of a function, the other allows to make a function pointer point to injected code. The real world exploit was one for the wu-ftpd-2.6.0 vulnerability. In all the three cases Paid detected and prevented the attack. The performance measurements for five applications (Qpopper, Apache, Sendmail, Wu-ftpd, Proftpd, pure-ftpd) showed a maximum overhead for sendmail of about 11.2%. Additionally they evaluated the increase in executable size for recompiled applications and found a maximum increase of around 250% for apache.

The proposed approach has some limitations and drawbacks. First, the need for the

source code of an application narrows the field of application. Second the workload is moderate if Paid should be used with a new version of an application and it is high if it should be used with a new kernel version or a new compiler version. This is due to the fact that changes to the source code of the compiler GCC and some libraries (LIBIO, libc) as much as recompilation of the applications to protect are necessary. Other limitations are that malicious code that contains no system calls or that simply modifies arguments to system calls in the application's code is not detected [15]. Another possibility to fool Paid are so-called mimicry attacks [47] which allow a sophisticated attacker to cloak their intrusion e.g. by using exactly the expected system call sequence. But because of the randomly introduced null system calls [16], an attacker has to guess the correct sequence unless he has access to the recompiled application binaries. Finally a more detailed evaluation of Paid with attacks in the wild would have been desirable.

3.3.6 TaintCheck

TaintCheck [31] is based on the idea of dynamic taint analysis. It protects a designated application by marking input originating form suspicious sources like e.g. the network interface as tainted. Afterwards, it tracks how the tainted data is used within the application. Finally, whenever tainted data is used in a way that is disallowed by the installed policy, TaintCheck generates an alert and launches the signature generation process. The output is a three byte long string signature.

TaintCheck performs dynamic taint analysis on a program by running the program in its own emulation environment. This allows TaintCheck to monitor and control the programs execution at a fine-grained level. Specifically TaintCheck is implemented using Valgrind [29]. Valgrind is an open source x86 emulator that supports extensions which can instrument a program as it is run. Whenever program control reaches a new basic block [17], Valgrind first translates the block of x86 instructions into its own RISC-like instruction set, called UCode. It then passes the UCode block to TaintCheck, which instruments the UCode block to incorporate its taint analysis code. TaintCheck then passes the rewritten UCode block back to Valgrind, which translates the block back to x86 code so that it may be executed. Instrumenting the code extracted from an applications binary, is often referred to as binary rewriting. To improve the performance of the described procedure, whenever a block has been instrumented, it is kept in Valgrinds cache in order to avoid its re-instrumentation every time it is executed. The whole dynamic taint analysis process is split into three modules: TaintSeed, TaintTracker and TaintAssert.
TaintSeed marks any data that comes from an untrusted source of input as tainted.

[15] The later is e.g. addressed by systrace [35]

[16] A custom system call that does not lead to a switch into kernel mode

[17] A sequence of instructions forms a basic block if the instruction in each position dominates, or always executes before, all those in later positions, and no other instruction executes between two instructions in the sequence

By default, TaintSeed considers input from network sockets to be untrusted, since for most programs the network is the most likely vector of attack. TaintSeed can also be configured to taint inputs from other sources considered untrusted by an extended policy, e.g., input data from certain files or stdin.
TaintTracker tracks each instruction that manipulates data in order to determine whether the result is tainted. The default policy of TaintTracker is as follows: for data movement instructions, the data at the destination will be tainted if and only if any byte of the data at the source location is tainted; for arithmetic instructions, the result will be tainted if and only if any byte of the operands is tainted. In order to track the propagation of tainted data, TaintTracker adds instrumentation before each data movement or arithmetic instruction. Optionally, TaintTracker provides a log that allows to follow the propagation of tainted data back to its entry point.
TaintAssert checks whether tainted data is used in ways that its policy defines as illegitimate. TaintAsserts default policy is designed to detect format string attacks, and attacks that alter jump targets including return addresses, function pointers, or function pointer offsets. When TaintAssert detects that tainted data has been used in an illegitimate way, signaling a likely attack, it invokes the Exploit Analyzer to further analyze the attack.
The Exploit Analyzer identifies the three most significant bytes used to overwrite a return address or a function pointer and determines the input data from which they originate. If the original content and the three bytes do not match, some decoding or other data transformation operations have been applied between data input and attack detection. That's why in this case, the original content is used as a signature.

To evaluate TaintCheck, the authors tried to exploit three real world [18] and three synthetic [19] vulnerabilities. All attacks were detected. Moreover, tests with an Apache webserver protected by TaintCheck showed that the increase in response time stayed below factor 25 [20]. Another test with bzip2 for which a 15MB input file was marked as tainted resulted in a slow down factor of 37.

A weak spot of TaintCheck is the three byte long string signature. It is a weak spot because benign input is likely to contain the same three byte sequence[21]. To address this limitation, the authors proposed to use an improved semantic analysis [22] that identifies filler bytes. Filler bytes are bytes that are part of the attack traffic but that are actually irrelevant (e.g.: parts of the attack traffic that do not affect directly or indirectly the control flow of the attacked application or service

[18] ATPhttpd (buffer overflow), cfingerd (format string), wu-ftpd (format string)

[19] function pointer, buffer overflow and format string vulnerability

[20] The increase is more significant for small requests. Factor 25 was found for a request size of 1KB.

[21] Assuming e.g. an uniform byte distribution there is on average one false positive per 16MB of traffic

[22] One example of such an analysis is e.g. to identify which protocol field(s) contained the malicious input and to determine the (minimal) input length for the attack to be successful

during a control flow hijacking attack, attack payload). A different problem is the possibility to clean tainted data if the described TaintTracker policy is applied. An example how this can happen is the following: An application decodes its input because it supports multiple character sets. It does this by using the tainted input characters as index into a translation table. It follows that because the content of the translation table is untainted, the same holds for the decoded input. Moreover, the large slow down factor of 1.5 to 37 limits its applications to unproductive systems like e.g. honeypots. But according to the authors there are various ways to speed up TaintCheck.

3.3.7 Vigilante

Vigilante is an end-to-end approach to contain fast spreading worms using collaborative worm detection at end hosts. Furthermore the approach includes an automatic filter generation component to protect the end hosts from subsequent attacks and an overlay network for fast alert distribution. But the major contribution of Vigilante is the concept of self-certifying alerts (SCAs). In the past, most approaches to contain fast spreading Internet worms relied on automatic signature generation and fast signature distribution. As a consequence, the hosts that exchange signatures have to trust each other. More precisely, they have to be sure that a signature, received from another host, has not been crafted and sent maliciously to block e.g. specific benign traffic (DoS attack). With the concept of SCAs, the need for trust between collaborating end hosts is (virtually) removed: A self-certifying alert contains a description of an attack that is detailed enough, to allow other hosts to verify, if they are vulnerable to it. If yes, the host can generate a filter to protect itself from the attack that triggered SCA generation. Otherwise, they can safely ignore the SCA.
A Vigilante host consists of the following components: A detector, a SCA verificator, a filter generator and an alert distributor:

Attack detector: Vigilante developed SCAs for the following three common vulnerabilities:

- Arbitrary Execution Control (AEC): Describes how to execute a piece of code whose address is supplied in a message sent by the attacker (execution redirection).
- Arbitrary Code Execution (ACE): Describes how to execute a piece of code that is supplied in a message send by the attacker (code injection).
- Arbitrary Function Argument (AFA): Describes how to invoke a specified critical function with an argument value that is supplied in a message sent by the attacker.

To generate these SCAs, Vigilante proposes and uses two different methods: NoneXecutable (NX) pages [32] and dynamic data flow analysis. NX pages can be used

to generate the first two SCA types while with dynamic data flow analysis it is possible to generate all three SCA types. Dynamic data flow analysis is based on the same principle as the dynamic taint analysis used by TaintCheck. The NX pages method is based on the idea that memory pages containing the code of an application are marked as executable and all other pages as non-executable. Whenever an attacker attempts to execute (injected) code in a protected page, the SCA generation process is triggered. To generate an SCA, the received messages are searched for the code that was about to be executed or for the address of the faulting instruction (or the critical argument). If it is found, its position is stored together with the relevant message in a candidate SCA.

SCA verificator: The SCA verificator replays the message(s) in the SCA versus a sandboxed version of the targeted service and replaces the section that is marked as critical with a nonce. Verification is successful, if the nonce is activated[23]. Otherwise verification fails. For self-generated SCAs it is checked, if adding messages preceding the message containing the critical data, leads to a successful verification (for attacks that require more than one message). Other SCAs are dropped.

Filter generator: Every new SCA that was successfully verified, triggers the generation of a general and a specific host-based filter. Vigilante uses execution path analysis to generate the specific filter. The general one is derived form the specific one using heuristics. In contrast to dynamic data flow analysis for SCA generation, execution path analysis requires instrumentation of the whole instruction set. This is a rather complex and time-consuming technique which requires another replay of the attack versus the vulnerable service. This time, the sandbox in which the vulnerable service is run, logs all decisions in the execution path and all instructions, where tainted data is involved. Furthermore, the corresponding data flow graph is constructed. Vigilante defines a full traversal of this graph as their filter condition. From this graph, a piece of assembler code is compiled that reflects the filter condition and it is added to the filter bank. By intercepting messages at the socket interface and by executing the filters in the filter bank with the message as input, any message matching a filter condition is dropped.

Alert distributor: Each new SCA is sent to all other hosts using flooding over a secure Pastry overlay [6]. This overlay, in combination with SCA forwarding rules from Vigilante, enables fast, resilient and secure distribution of SCAs. To counter flooding with bogus SCAs or with old SCAs, hosts forward only SCAs that are new to them and that could be successfully verified.

[23] AEC-Nonce: Address of a specific function. If it is called, verification is successful.
ACE-Nonce: Piece of code that contains e.g. a jump to a specific function. If the function is called, verification is successful.
AFA-Nonce: Function argument. If the in the SCA specified function is called with this argument, verification is successful.

For their evaluations, the authors used three well-known and well-understood worms: Slammer, Blaster and CodeRed. The preformed evaluations are the following:

Simulation of the percentage of infected hosts: They used the epidemic model described in [19] with minor modifications to take detectors into account. First, they present the percentage of infected hosts in dependence of the ratio of detectors for all three worms using model parameters that reflect the observed behavior of these worms in the wild and the respective SCA generation and verification times. It can be seen that a fraction of 0.001 is enough to contain the worm infection to less than 5%. Second they evaluated for the Slammer worm, what changes to the signature verification time, the infection rate and the number of initially infected hosts influence the percentage of infected hosts. The result of their evaluation is that Vigilante is able to contain the spreading of a worm for scenarios that are a magnitude worse than the worst of real worms in the past.

Filter generation time: They evaluated the filter generation time for the three worms. The maximum generation time was around 3.4 for the CodeRed worm.

Filter quality: Manual inspection of the generated filters showed that the filter for the Slammer worm has no false positives or negatives. The filters for Blaster and CodeRed do not cause false positives as well but they are not able to filter all attacks on the vulnerabilities exploited by Blaster and CodeRed.

A weakness of Vigilante is its vulnerability to DoS attacks when the time needed to verify an alert is significant. The authors propose to address this problem by distributing the alerts over a secure Pastry [6] overlay. To participate in the overlay, a host has to get a certificate from a trusted offline certification authority. Basically, Pastry combined with the policy to forward only verified SCAs, it is assured that a malicious host can only launch DoS attacks on its neighbors in the overlay network and that the host can not influence the topology of the network. Nevertheless, these measures are only efficient, if the number of malicious hosts participating in the overlay is small. In addition, the authors propose the use of super-peers [24]. In this scenario, the neighbors of an ordinary host is always a super-peer. Therewith, a malicious ordinary host can not attack other ordinary hosts but the super-peers. But despite the measures against DoS based on malicious SCAs, there still remains the possibility to use ordinary DoS/DDoS attacks one super-peers.

[24] In Vigilante, theses are hosts that are not vulnerable to most worm attacks because they run only the overlay code and a set of virtual machines with sandboxed versions of the vulnerable services to verify the SCAs

3.4 Approaches Using Network Level and Host Level Attack Detection

3.4.1 HoneyStat

The only system, at the time of this writing, which combines network and host level attack detection methods is HoneyStat [16]. This approach is specifically targeted at attack detection in local networks. HoneyStat nodes are emulating multiple operating systems using VMware GSX Server. They detect three different types of events: memory events, network events, and disk events. Memory events are detected by a buffer overflow detection software running on the honeypot. A network event is triggered in case a honeypot generates outgoing traffic. Finally, disk events are triggered if a process writes to specified parts of the filesystem.

HoneyStat does not generate any signatures from the generated events. However, information which is recorded during a HoneyStat event includes:

- OS/patch level of the host
- Type of event and relevant capture data (memory events - stack state, network events - outgoing packet(s), disk events - file changes)
- Trace file for all prior network activity

The gathered information is then forwarded to a central analysis node. This central node correlates all received HoneyStat events and performs a logistic regression. Instead of correlating two continuous variables (such as traditional linear regression), logistic regression considers a dichotomous variable (such as boolean states) and continuous variables. The analysis tries to explain the changes in the honeypot state (e.g. asleep or awake) with one or more of the continuous variables. Unfortunately, the authors do not clearly state which continuous variables they use. We assume, however, that they mainly rely on incoming network traffic.

Any logistic analysis involves the following steps: For each particular HoneyStat event the set of variables that minimize the prediction error is determined using maximum-likelihood evaluation. Second, the Wald statistic is used to evaluate each variable, and remove those below a user-selected significance threshold. If the analysis results in a single variable explaining the changes in the honeypot state, an alert is generated. Otherwise, the event data is stored until additional events are observed, triggering a renewed analysis. Although HoneyStat does not explicitly generate any signatures, the continuous variable identified by logistic regression (e.g. some byte sequences in the network traffic) could serve as a signature.

HoneyStat's attack detection mechanism relies, among other criteria, on observing outgoing network traffic. Hence, the false positive rate of the attack detection should be zero or at least very low. The authors state, however, that errors such as identifying benign traffic as the source of an alert could occur when applying logistic regression. Signatures based on this falsely identified network traffic would generate false positives when used in a signature-based detection system. HoneyStat was tested with a 13-month activity honeypot log from the Georgia Tech Honeynet Project. A worm attack was manually injected in this trace to test

whether HoneyStat would generate a false positive. The authors state that HoneyStat did not generate any false positives for this trace. However, they do not explain their evaluation methodology in detail.

Chapter 4

Classification of Reviewed Existing Approaches

This chapter classifies the reviewed approaches according to different criteria regarding attack detection capabilities, signature generation capabilities and evaluation methodology. Moreover, the classification allows us to reason about which of the approaches are most interesting in the focus of NoAH. More precisely, each of the sections 4.1, 4.2 and 4.3 contains a summary where we identify one or more approaches that are closest to an ideal attack detection or signature generation system. Therefore, the summary presents too an outline of the requirements for an idealized attack detection or signature generation systems. A more precise description of what an ideal system means in the context of NoAH can be found in Chapter 5.

4.1 Classification Regarding Attack Detection Capabilities

This section presents the classification criteria and parameters for the attack detection part, and classifies the reviewed approaches. This section refers always to the detection of yet unknown attacks. Some of the proposed approaches detect (and filter) attacks they have already seen in a different way. Moreover, only approaches that actually have mechanisms for attack detection are considered. More precisely this concerns Honeycomb and Nemean because of their assumption that all traffic received by a honeypot is malicious, Polygraph because it requires a classificator separating malicious from benign traffic without naming/presenting one and Earlybird because it uses some sort of attack detection (based on address dispersion) only in a second stage. For details refer to section 3.1.

4.1.1 Attack Detection Mechanism

This criterion classifies the signature generation approaches according to the attack detection mechanism they use. We identified the following attack detection mechanisms:

- Scan-based detection
- Redirecting/observing outgoing connections
- Machine learning of byte frequency distributions
- Observing filesystem and stack state changes
- Address Space Randomization (ASR)
- Non-Executable pages
- Source code instrumentation
- Discrepancies from static per application system call model
- Dynamic taint analysis

Table 4.1 contains the relevant information.

	Attack Detection Method
Autograph	Scan-based detection
PADS	Redirecting outgoing connections
PAYL	Machine-learning of byte frequency distributions
COVERS	Address Space Randomization (ASR)
DIRA	Source code instrumentation
DOME	Discrepancies from static per application system call model
Minos	Dynamic taint analysis
Paid	Discrepancies from static per application system call model
TaintCheck	Dynamic taint analysis
Vigilante	Dynamic taint analysis or Non-Executable pages
HoneyStat	Observing outgoing connections, filesystem and stack state changes

Table 4.1: Attack Detection Mechanisms

4.1.2 Detected Attack Types

This criterion describes different attack types that can be detected by the respective approach. They are characterized by the following parameter:

- Transport Protocol: An attack has to use one of these: TCP, UDP or ANY
- Propagation Method: Either Random Scanning, Hitlist, Topology or ANY
- Exploitable: Yes/No. If Yes, only attacks for which the target is exploitable are detected.

Table 4.2 contains the relevant information.

	Transport Protocol	Propagation Method	Exploitable
Autograph	TCP	Random Scanning	No
PADS	TCP, UDP	Random Scanning	Yes
PAYL	TCP, UDP	Random Scanning, Hitlist, Topology	Yes
COVERS	ANY	ANY	Yes
DIRA	ANY	ANY	Yes
DOME	ANY	ANY	Yes
MINOS	ANY	ANY	Yes
Paid	ANY	ANY	Yes
TaintCheck	ANY	ANY	Yes
Vigilante	ANY	ANY	Yes
HoneyStat	ANY	ANY	Yes

Table 4.2: Capabilities regarding Characteristics of Detected Attacks

4.1.3 Attack Detection Input

This criterion classifies the approaches according to the input that is required by the proposed attack detection method. We identified the following input types:

- Raw network traffic
- Reassembled TCP flows
- Information about executed system calls
- Information about outgoing connections
- Filesystem and stack logs
- Memory locations of return addresses and function pointers
- Memory read/write operations
- Memory access violations

Table 4.3 contains the relevant information.

4.1.4 Type of Attack Detection System

This criterion classifies the type of the end system that serves as bait for an attacker. We identified the following end system types:

- Real
- Virtualized
- Emulated
- Simulated

Table 4.4 contains the relevant information.

	Input Data
Autograph	Raw network traffic
PADS	Raw network traffic
PAYL	Raw network traffic
COVERS	Memory access information
DIRA	Memory locations of return addresses and function pointers
DOME	Information about executed system calls
Minos	Memory read/write operations
Paid	Information about executed system calls
TaintCheck	Memory read/write operations
Vigilante	Memory read/write operations or Memory access violations
HoneyStat	Filesystem and stack logs, information about outgoing connections

Table 4.3: Input for Attack Detection

4.1.5 Expected Attack Detection Delay

This criterion describes if there is a delay from the first attack instance observed at the detection system until the attack is finally reported. The results are summarized in table 4.5. Whenever there is a delay, a short explanation for its occur

4.1.6 Summary

An ideal attack detection system fulfills the following two requirements:

1. It should detect any attack on itself and/or the systems it supervises at the moment they take place without false positives/false negatives.

2. Its resource consumption should be zero

The criteria Detected Attack Types and Expected Attack Detection Delay are relevant to the first requirement while the criteria Attack Detection Mechanism, Type of Attack Detection System and Attack Detection Input are relevant to the second requirement. Comparing the criteria for the first requirement leads us to the conclusion that Minos, TaintCheck and Vigilante are the most interesting approaches. In contrast to the other approaches, they can detect attacks on applications on the system that they supervise without delay and with close to zero false positives. Nevertheless, their focus on control-flow hijacking attacks is a limitation since they do not detect ANY attack. Another important aspect is that they detect attacks only if the supervised system is vulnerable to it. Furthermore, a final statement about the false positive/false negative rate is not possible and we refer to the respective publication for details. As far as the second requirement is concerned, Minos, TaintCheck and Vigilante are actually a bad choice since emulation, simulation and virtualization usually require a lot of resources.

	End System Type
Autograph	Real
PADS	Real/Simulated [a]
PAYL	Real
COVERS	Real
DIRA	Real
DOME	Real
Minos	(Emulated) [b]
Paid	Real
TaintCheck	Emulated
Vigilante	Virtualized
HoneyStat	Emulated

[a] Double honeypot system. The second honeypot is a low-interaction honeypot

[b] Actually Minos consists of a real system with some special purpose hardware. But since this hardware does not (yet) exist, it is emulated.

Table 4.4: Type of Attack Detection System

	Delay	Reason
Autograph	Yes	In order to classify traffic from a specific source as attack traffic, it is necessary that it initiated at least a predefined number of unsuccessful connections.
PADS	Yes	Detected only in case the attacked system initiates itself a connection.
PAYL	Yes	Detected only in case the attacked system initiates itself a connection.
COVERS	No [a]	
DIRA	No	
DOME	No	
Minos	No	
Paid	No	
TaintCheck	No	
Vigilante	No	
HoneyStat	Yes	Significance threshold for event correlation

[a] But there are some (rare) cases, where ASR introduces a delay. See 3.3.1.

Table 4.5: Attack detection delay

4.2 Classification Regarding Signature Generation Capabilities

As the review of approaches using host level attack detection in section 3.3 shows, Minos and DIRA do not generate any signatures and are therefore unaccounted for in this section. The same applies for HoneyStat. Furthermore, DOME and Paid do not generate signatures that characterize attacks. Nevertheless they still use some kind of a signature. More specifically, they characterize the approved/expected/normal behavior of an observed system or one of its parts. Nevertheless it is basically easy to derive a signature characterizing attacks with the approved/expected/normal behavior signature at hand (e.g. by taking the deviation from the approved/expected/normal behavior as signature), the authors purpose was not the generation of such signatures. Therefore we apply our classification criteria to the approved/expected/normal behavior signature.

4.2.1 Signature Generation Input

This criterion describes the input that is required by the proposed signature generation method. We identified the following input types:

- Single packet payloads
- Per-flow reassembled payloads
- Per-flow reassembled, service normalized packets
- Flow payload ("'Flow"' is not further specified)
- Source code (of any application against which attacks should be detected)
- Application binary (of any application against which attacks should be detected)

Table 4.6 contains the relevant information.

	Signature Generation Input
Honeycomb	Per-flow reassembled payloads
Polygraph	Per-flow reassembled payloads
Earlybird	Single packet payloads
Nemean	Per-flow reassembled, service normalized packets
Autograph	Per-flow reassembled packet payloads
PADS	Not specified in detail
PAYL	Single packet payloads
COVERS	Single packet payloads or flow payload
DOME	Application binary
Paid	Source code
TaintCheck	Single packet payloads or flow payload
Vigilante	Single packet payloads or flow payload

Table 4.6: Input for Signature Generation

4.2.2 Signature Type

This criterion classifies the approaches according to the type of signature they generate. We describe the signature type with two parameters: The system layer at which the signatures are applied, and the representation format of the signature.

Possible values for the system level are:

- Network layer (e.g. byte patterns, application semantics, byte frequency distributions)
- Host layer (e.g. memory access information, filesystem changes, process characteristics)
- Network and host layer (any combination of the approaches from above)

The representation of signatures identifying an attack can be:

- Continuous byte pattern
- Discontinuous byte pattern (DBP)
- Byte probability distribution
- Regular expression (RegEx)
- Finite state automaton
- Message type/field characteristics: Specifies disallowed characteristics for a specific message field [1].
- Disallowed control-flow modifications: A policy specifying disallowed control-flow modifications.
- All the traffic needed to trigger the detector plus attack specific information[2]

The representation of signatures identifying approved/expected/normal behavior can be:

- Policy specifying disallowed control-flow modifications
- Per application system call usage model

Table 4.7 contains the relevant information.

4.2.3 Applicability to Polymorphic Attack Payload

This criterion describes the applicability of the created signatures to polymorphic attack payloads. [3]

- (limited) metamorphism
- (limited) oligomorphism
- ”total” polymorphism

Table 4.8 contains the relevant information

4.2.4 Expected Signature Generation Delay

This criterion describes the expected signature generation delay. That is, the delay from attack detection until a signature is finally generated. To identify the total de-

[1] Message-format-aware or protocol-aware signature

[2] see section 3.3.7 for details

[3] Section 3.2 specifies how the terms ”polymorphism” and (attack) payload are used in this report.

	Target Layer	Representation
Honeycomb	Network layer	Continuous byte pattern
Polygraph	Network layer	DBP, RegEx, Byte probability distribution
Earlybird	Network layer	Continuous byte pattern (Snort)
Nemean	Network layer	Finite-State-Automaton
Autograph	Network layer	Continuous byte pattern (Bro)
PADS	Network layer	Byte frequency distribution
PAYL	Network layer	Continuous and discontinuous byte pattern
COVERS	Network layer	Message type/field characteristics
DOME	Host layer	Per application system call usage model
Paid	Host layer	Per application system call usage model
TaintCheck	Network layer	Continuous byte pattern (3 Bytes)
Vigilante	Network layer/ Host layer [a]	All the traffic needed to trigger the detector plus attack specific information

[a] Self-Certifying alerts

Table 4.7: Type of Generated Signature

lay from the first occur We identified the following sources of delay in the signature generation process:

- Minimal number of attack instances: A signature can only be generated if more than one instance of the same attack is required.
- Signature verification: The signature has to be verified prior to deployment.
- No additional delay: The delay depends on processing complexity only. If available, the maximum processing delay [4] is added in brackets.

Table 4.9 contains the relevant information.

[4] The maximum delay is often determined experimentally by launching different attacks against the system

	Applicability to Polymorphic Attack Payloads
Honeycomb	limited oligomorphism and limited metamorphism
Polygraph	oligomorphism and metamorphism
Earlybird	limited metamorphism
Nemean	limited metamorphism
Autograph	limited oligomorphism and limited metamorphism
PADS	metamorphism
PAYL	oligomorphism and metamorphism
COVERS	”total” polymorphism
DOME	”total” polymorphism [a]
Paid	”total” polymorphism [b]
TaintCheck	”total” polymorphism
Vigilante	”total” polymorphism

[a] because the signature does not describe the attack but the allowed/expected/approved behavior of the system

[b] because the signature does not describe the attack but the allowed/expected/approved behavior of the system

Table 4.8: Applicability of Generated Signatures to Polymorphism

	Source of Signature Generation Delay
Honeycomb	Minimal number of attack instances
Polygraph	No additional delay [a]
Earlybird	Minimal number of attack instances
Nemean	Minimal number of attack instances
Autograph	No additional delay
PADS	Minimal number of attack instances
PAYL	No additional delay
COVERS	Minimal number of attack instances [b]
DOME	No additional delay
Paid	No additional delay
TaintCheck	No additional delay
Vigilante	No additional delay (<3s)

[a] But signatures are likely to get the more specific/accurate the more benign and attack traffic is available

[b] In most cases, only one. See item 5 in the list in section 3.3.1 for details.

Table 4.9: Expected Signature Generation Delay

4.2.5 Summary

An ideal signature generation system fulfills the following two requirements:

1. It generates one signature per vulnerability. The signature allows to identify any attack aiming at the corresponding vulnerability without false positives/false negatives.

2. Signature generation requires zero time.

The criteria Signature Generation Input, Signature Type, Applicability to Polymorphic Attack Payload are relevant to the first requirement while the criteria Expected Signature Generation Delay is relevant to the second requirement. Comparing the criteria for the first requirement leads us to the conclusion that COVERS and Vigilante are the most interesting approaches. In contrast to the other approaches, they create not only signatures that are applicable to polymorphic attack payload but also use signature types that have (potentially) low false positive/false negative rates [5]. Additionally, since COVERS and Vigilante have a reasonable delay depending only on the characteristics of an attack and the parameters of the signature generation system and not on the availability of multiple instances of an attack like e.g. Honeycomb and Earlybird, they fulfill the second requirement quite well.

4.3 Classification Regarding other Criteria

Whenever a proposal is thoroughly evaluated, it is likely that it can be compared to other well-evaluated approaches. It is therefore helpful to know how much effort was put into evaluation, or more precisely, what was evaluated and what was the applied evaluation methodology.

4.3.1 Performed Evaluations

The ”Performed Evaluations” criterion allows to identify if one or more of the following evaluations is presented in the paper:

- Attack detection: An evaluation that gives information about how good the attack detection mechanism is in detecting attacks.
- Signature quality: An evaluation that gives information about the usefulness of the generated signatures.
- Performance: An evaluation that provides information about the performance of (a part) of the system.

Table 4.10 contains the relevant information.

[5] See sections 3.3.7 and 3.3.1 for details

	Attack Detection	Signature Quality	Performance
Honeycomb	-	X	-
Polygraph	-	X	-
Earlybird	-	X	X
Nemean	-	X	-
Autograph	-	X	-
PADS	-	X	-
PAYL	-	X	-
COVERS	X	X	X
DIRA	X	-	X
DOME	X	-	X
Minos	X	-	X
Paid	X	-	X
TaintCheck	X	(X)	X
Vigilante	X	X	X
HoneyStat	(X)	-	-

Table 4.10: Performed Evaluations

4.3.2 Evaluation Methodology: Attack Detection

Here, we give criteria that characterize the methodology that was applied to evaluate attack detection quality. The following criteria characterize the methodology used to evaluate attack detection:

- Attack Detection Setup: The setup used to evaluate attack detection. Parameters can be either:
  - real-world deployment: The system was deployed in a productive network. The authors had no influence if and how the system was attacked.
  - lab deployment, real-world attacks: The role of the attacker is assigned to a host/user in the lab environment. He uses real world attacks.
  - lab deployment, synthesized attacks: The role of the attacker is assigned to a host/user in the lab environment. He uses synthetic attacks.
  - lab deployment, replayed attacks: Replay of real-world traffic traces. This includes modifications such as sanitization and attack injection.
- Measured Parameter. It can be either:
  - FP: False Negatives
  - FN: False Positives
  - TP: True Positives
  - TN: True Negatives
- Probe Size: The number of different attacks (true positives/false negatives) or the number of benign interactions (true negatives/false positives) if available.
- Establishment of Ground Truth. Parameters can be either:

  - manual analysis: The authors did a manual analysis to find the actual number of attacks.
  - reference system: The number of identified attacks was compared to the number of attacks found by a reference system.
  - attack injection: A specific number of attacks was injected.
  - attack only: All interactions with the system belonged to attacks. Hence, this allows true positive/false negative analysis only.
  - benign only: All interactions with the system were benign. Hence, this allows true negative/false positive analysis only.
  - concept: The applied attack detection concept guarantees a minimum/maximum value for the measured parameter(s).

Table 4.11 sums up the relevant information for these criteria. Systems without attack detection mechanism are omitted.

4.3.3 Evaluation Methodology: Signature Quality

The following criteria characterize the methodology used to evaluate signature quality:

- Signature Generation Setup: The setup used to generate the set of signatures for the signature quality assessment. Parameters can be either:
  - real-world deployment: The system was deployed in a real-world network environment while generating the set of signatures.
  - lab deployment, synthesized traffic: The traffic used to generate the signature set was either hand-crafted or dynamically generated using a testbed.
  - lab deployment, replayed traffic traces: The traffic used to generate the signature set was generated by replaying a set of (real-world) traffic traces.
  - lab deployment, replayed but modified traffic traces: A set of (partially) modified (real-world) traffic traces was replayed.
  - analytically assessed: No experimental setup. The signature quality was assessed analytically.

The classification of the different approaches according to these criteria can be found in Table 4.12. Only systems that generate attack specific signatures are listed. If a system generates e.g. a signature that identifies allowed/approved actions/behavior, an evaluation of its signature generation mechanism is actually done when evaluating the attack detection mechanism.

Further criteria to characterize the methodology used to evaluate signature quality are:

- Quality Assessment Setup. Parameters are the same as as for the Signature Generation Setup. If not stated otherwise, the network traffic for the assessment is not identical to the network traffic used to generate the set of signatures.

- Measured Parameter. It can be either:
  - FP: False Negatives
  - FN: False Positives
  - TP: True Positives
  - TN: True Negatives
- Establishment of Ground Truth. Parameters can be either:
  - manual analysis: Either the authors analyzed the network traffic manually to find the number of attacks contained in it or they analyzed if the total set of traffic that the signature/filter describes/filters contains only attacks.
  - reference system: Either the number of identified attacks was compared to the number of attacks found by a reference system or the attacks reported by the reference system were removed to ”sanitize” the traces.
  - attack injection: A specific number of attacks was injected into network traffic.
  - attack only: The network traffic consisted of attacks only.

The classification of the different approaches regarding theses criteria can be found in table 4.13. Again, only systems that generate attack specific signatures are listed.

4.3.4 Collaboration among Sensors

This criterion specifies whether and how the different approaches use collaboration among multiple sensors. Possible values are:

- no collaboration
- attack detection
- signature generation: Collaboration to reduce the signature generation load
- signature quality: Collaboration to improve the quality of the generated signatures

4.3.5 Summary

The main purpose of an evaluation is to identify how well the presented systems/algorithms perform in different set-ups and therewith to enable other people to identify its qualities and its limitations. Ideally, the evaluation results allow others to decide if the system/algorithm suits their needs and if it works in their setup as well. Even though most of the reviewed approaches do some evaluations, they do only very specific ones, meaning that only few parameters were measured and that the presented results are derived from a small test set. Additionally, whenever a reference system was used (e.g. Nemean, Autograph, PADS) to measure the false positive/false negative rate, these were systems that themselves are known to have a false positives/false negatives problem. After all, the evaluations presented by COVERS and Vigilante are well established but nonetheless their generality is questionable due to the small test sets.

	Attack Detection Setup	Measured Parameters	Probe Size	Establishment of Ground Truth
Autograph	(not evaluated)	-	-	-
PADS	(not evaluated)	FP	-	concept [a]
PAYL	lab deployment, replayed attacks	FP, FN	?	reference system (Snort) & attack injection
COVERS	lab deployment, real-world attacks	TP	7	attack only
DIRA	lab deployment, real-world attacks	TP	3	attack only
DOME	lab deployment, real-world attacks	TP, FP	6, 8	attack only, benign only
Minos	real-world deployment & lab deployment, synthesized- and real-world attacks	TP, FP	18, -	attack only, manual analysis
Paid	lab deployment, synthesized/real-world attacks	TP	3	attack only
TaintCheck	lab deployment, synthesized/real-world attacks	TP, FP&FN	6, -	attack only, concept
Vigilante	lab deployment, real-world attacks	TP	3	attack only
HoneyStat	lab deployment, synthesized or real-world attacks[b]	FP	1	none

[a] Because of the double-honeypot technology, there are no false positives (see 3.2.2)
[b] The evaluation methodology is very poorly described

Table 4.11: Attack Detection: Evaluation Methodology

	Signature Generation Setup	Quality Assessment Setup
Honeycomb	real-world deployment	(not assessed)
Polygraph	lab deployment, replayed traffic traces	lab deployment, replayed but modified traffic traces
Earlybird	real-world deployment	real-world deployment
Nemean	lab deployment, replayed traffic traces	lab deployment, replayed traffic traces
Autograph	lab deployment, replayed traffic traces	lab deployment, replayed but modified traffic traces
PADS	lab deployment, synthesized traffic	lab deployment, replayed but modified traffic traces
PAYL	lab deployment, replayed traffic traces	lab deployment, replayed but modified traffic traces
COVERS	lab deployment, synthesized traffic	lab deployment, synthesized traffic
TaintCheck	(not deployed) [a]	analytically assessed
Vigilante	lab deployment, synthesized traffic	analytically assessed

[a] Concept for a signature generation mechanism

Table 4.12: Signature Generation: Evaluation Methodology I

	Measured Parameter(s)	Establishment of Ground Truth
Honeycomb	(None [a])	(None)
Polygraph	FN, FP	manual analysis
Earlybird	FN, FP	reference system (snort-inline)
Nemean	FN, FP	reference system (Snort)
Autograph	TP, FP	reference system (Bro)
PADS	TP, FP	reference system (Snort) and attack injection
PAYL	FP, FN	reference system (Snort) and attack injection
COVERS	TP, FP	attack only, manual analysis
TaintCheck	FP	reference system [b]
Vigilante	FP, FN	manual analysis

[a] The authors evaluated the number of generated signatures

[b] estimation based on frequency measurements of 3-byte sequences in specific network traffic types

Table 4.13: Signature Generation: Evaluation Methodology II

	Collaboration among Sensors
Honeycomb	no collaboration
Polygraph	no collaboration
Earlybird	no collaboration
Nemean	no collaboration
Autograph	attack detection [a]
PADS	no collaboration
PAYL	signature quality [b]
COVERS	no collaboration
DIRA	no collaboration
DOME	no collaboration
Minos	no collaboration
Paid	no collaboration
TaintCheck	no collaboration
Vigilante	signature generation [c]
HoneyStat	attack detection [d]

[a] sharing of suspicious source addresses

[b] suggests signature correlation (Z-strings)

[c] by exchanging signatures and use them to filter out attacks

[d] Additionally to event correlation, the OS/patch level configuration of sensors can be adapted.

Table 4.14: Collaboration among Sensors

Chapter 5

Requirements and Design Goals

5.1 Attack Detection

5.1.1 NoAH as Early-Warning System

The purpose of having an early-warning system is to increase the time available for installing countermeasures, or more generally protection, against a previously unknown attack. Therefore, each early-warning system must aim at minimizing the time t_{det} between the outbreak of such an attack and the generation of an appropriate alert. This time can be divided in the period t_{obs} before a new attack is observed on one of the sensors of the early-warning system, and the period t_{gen} after observation until an alert is finally generated.

The time between the outbreak of a previously unknown attack and its observation on one of the sensors t_{obs} depends mainly on the number and placement of the deployed NoAH sensors. It is clear that with increasing the number of NoAH sensors (or the traffic volume directed towards our NoAH sensors), we augment our chances for an early-detection. At least, this is the case for random-scanning attacks. Considering hitlist- and topology-based propagation mechanisms, we can increase our chances for observing such an attack by advertising the honeypots as potential targets, e.g., as DNS servers.

The time after a first observation until an alert is generated t_{gen} depends on the attack detection mechanism used, and on its performance. We should aim for applying a detection mechanism which is able to generate an alert upon the first observation of a previously unknown attack. Hence, threshold-based attack detection mechanisms used in [23] are usually not suitable for being used in an early-warning system. But since NoAH aims at having a lot of sensors in different places, these thresholds could eventually be reached fast enough Host-based attack detection mechanisms such as [31] are able to detect a previously unknown attack at the moment it reaches the host. But because they detect the actual exploitation of a host, it has to be vulnerable to the attack. Moreover, we should consider the performance overhead induced by alert generation. Hence, the detection process should be kept as simple as possible.

5.1.2 Detection of Novel Attacks in NoAH

As we have seen in the preceding classification (see 4.1), attack detection mechanisms are based on different attack characteristics. Some of the detection mechanisms, [23] for instance, rely on scan detection. These approaches presume that all present and future attacks will rely on random-scanning as propagation mechanisms. It widely agreed, however, that random-scanning will diminish as propagation mechanism as soon as IPv6 (with its 128-bit long addresses) is widely deployed in the Internet. At this time, topology- and hitlist-based worms will take over the scene since random scanning will not be efficient anymore.

In the preceding section, we have identified host-based attack detection as the most suitable approach for an early-warning system. Host-based attack detection systems such as [31] base their detection mechanism on observing the actual exploit that takes place. Today, the most common exploit is still the well-known buffer overflow. However, there is evidence that buffer overflows will become much less frequent in the future. This is due to the fact that functions which are vulnerable to buffer overflows (e.g. printf in C) get replaced by secured functions. So in the future we will see new forms of exploits emerging. Some of these novel exploits have already been observed in the wild such as format string attacks and heap overflows.

Hence, we argue that NoAH should use an attack detection mechanism that is as general as possible or at least easily extensible to novel exploits and attack forms. Regarding host-based attack detection, which we identified as the most suitable approach in the preceding section, this means that we should not rely on buffer overflow detection but also to consider other exploit forms such as format string attacks. Depending on the wish list of NoAH participants, attack detection can be extended with detectors for other types of attacks like e.g. Denial-of-Service (DoS) or password guessing attacks.

5.2 Signatures and Signature Generation

In this section we discuss the goals of NoAH regarding signatures and signature generation. One of the goals of NoAH is to contain the spread of cyberattacks on the Internet by automatically generating signatures of cyberattacks in a machine-readable form that could be easily read by firewalls and Intrusion Prevention Systems(IPS). All of the designs presented in section 6.2 provide enough information to meet this goal. We plan to fulfill it by implementing a generator for Snort signatures. Another goal of NoAH is to study the issue of generating simple but effective signatures that could be deployed in commodity routers/switches in the core of the network. This goal is discussed separately in section 5.2.4. Nevertheless, our review of existing approaches on automated attack detection and signature generation as well as reports about todays anti-virus products, firewall and IPS systems show that the problem of detecting unknown or polymorphic attacks and generating precise and unambiguous signatures for them is only partially solved. Most

notably, this applies to signatures that are easy to distribute and whose use does not require special hardware or complex software environments. After all, most of the widely available firewall or IPS systems use signatures of this type and generate either high false positive or false positive rates or both.

5.2.1 A ”Perfect” Signature

To overcome the above limitations, our designs also include new ideas aiming at generating signatures that meet our definition of the term ”signature” (see chapter 2) better. Concretely, this means that the generated signatures have to meet the following requirements:

- Unambiguity (no false positives): A system using signatures designed to match/prevent malicious activities should match/prevent malicious activities only (e.g. all attack using injection vector X).
- Sufficiency (no false negatives): A system using signatures designed to match/prevent malicious activities should match/prevent all malicious activities using the same injection vector to be resistant against polymorphic payloads.

A precondition to meet this requirements is accurate automated attack detection combined with sophisticated information about the detected attack. Chapter 6 presents the information that is required by our signature generation process designs. Furthermore, our review of existing approaches and the study of past worm outbreaks [51], [45], [5], [27] justify the following requirements for the signature generation process:

- Fast signature generation and distribution
- Signature verification

5.2.2 Fast Signature Generation and Distribution

In order to slow down or stop the spreading of fast worms like Code Red, it is necessary to generate and distribute signatures within certain time limits. These limits depend on factors like initially infected hosts, spreading technique and the number of vulnerable hosts (average infection rate) and are therefore highly variable. In fact, there exist simulations for specific setups and countermeasures like e.g. for a SQLSlammer outbreak simulation described in [10]. In their simulation a small fraction of detectors (0.001) is enough to contain the worm infection to less than 10 percent of the vulnerable population if signature generation takes less than 5 seconds and is afterwards immediately distributed over an overlay network. For their simulation they assumed the epidemic model [44] for the spreading and an average infection rate of 0.117 [1].

[1] This means that every 8.5 seconds the number of infected machines doubles

5.2.3 Signature Verification

Basically, signature verification has two applications. First, it can be used to decide if a signature should be installed or not. Second, it is used to decide if a signature produces false positives/false negatives.

The first application is a modification of the concept of self-certifying alerts (SCA) presented by Vigilante [2] and could be named self-certifying signatures (SCS). A SCS contains a signature along with enough information to automatically replay the attack which the signature should identify. Hence, a system receiving a SCS from another one could replay it within a containment environment and verify, if it is actually vulnerable to the attack and if the signature identifies the attack. This means that the system has to be able to replay somehow the provided attack. But attack replay can be very easy or very hard if e.g. time, system or location specific information is present in the data part of the communication flow. There exist different approaches, like e.g. the one from Vei and Paxon [15], addressing the replay problem. Another thing to consider is that SCSs themselves open up new ways to attack a system. Using the outlined simple approach allows an attacker to provide a signature that matches not only the provided attack but also benign activities. This is where the second application comes into play.

The second application is the measurement of the false positive and false negative rate for a specific signature. If both, the false negative and the false positive rate are zero, the signature is considered to be useful and is e.g. put into or remains a part of a signature database. One method to measure the false positive rate is to replay a set of representative benign traffic while it is checked if the signature identifies one or multiple attacks. Another method is to compare the reports with those of one or more reference systems. While there exist feasible methods to assess the false positive rate of a specific signature, it is difficult to assess its false negative rate for the following reason: A signature without false negatives matches ideally all attacks on a specific vulnerability, hence it is an accurate description of the injection vector. But since measuring its false negative rate by attacking the corresponding vulnerability requires also knowledge about the injection vector, this is rather a tail biter. A more promising approach would be to have an attack detector that is able to distinguish the attacked vulnerabilities and to verify if the therewith related signatures identified the attacks too. An attack detector that could be suitable for this purpose is TaintCheck. It allows to identify the attacked application or service as much as the extraction of information about the use of attack-relevant (tainted) data (see section 3.3.6 for details).

[2] see section 3.3.7 and [10] and [11]

5.2.4 Signatures for Special Hardware

As stated at the beginning of section 5.2, the issue of generating signatures that could be deployed in commodity routers/switches in the core of the network is of interest to NoAH. Since routers usually have to handle lots of traffic and have rather limited resources for additional data processing, they should be given simple and fast-to-use signatures. Nevertheless, because they are deployed in the core of the network, they have the potential to be very powerful mechanisms in containing the spread of a cyberattack. If core routers interact with edge firewalls and intrusion prevention systems, e.g. by tagging suspicious traffic, an efficient pre-processing and/or priorization of traffic may be implemented already at the perimeter of a network. Towards using or generating signatures based on network level information on routers, we identified the following three major problems:

- Router resources/performance
- Secure and reliable service deployment on router infrastructure
- Secure and reliable service update (signatures)
- Violation of the end-to-end principle/Traffic ownership

While there are indications that simple pattern matching and tagging of traffic could be possible without the need for special purpose hardware, this seems unlikely to be the case for signature generation or would require an EU project for itself. Therefore, we consider deploying and using network traffic based signatures only by included it as an option in our design. The argumentation is based on the dissertations of Matthias Bosshardt [4] and Lukas Ruf [37] about service deployment and design on routers or other special purpose hardware.

5.3 Requirements for the Honeypot Architecture and the Containment Environment

5.3.1 Requirements for the NoAH Architecture

A description of the NoAH architecture can be found in deliverable D1.1. To meet the design goals it is necessary that it fulfills the following requirements:

Clever sensor placement
For an early attack detection clever placement of NoAH sensors and redirectors to shorten detection time is the most critical requirement.

Core with high-interaction honeypots covers a lot of different software (and hardware) configurations
Since the proposed attack detector can only detect attacks to which the honeypot on which it resides is vulnerable, and because different software versions may have different vulnerabilities, the architecture should allow to manage honeypots with different software (and hardware) configurations efficiently.

Protecting the core from DoS attacks
Because of the resource and performance critical design of the high-interaction honeypots it is relatively easy to launch a DoS attack against these systems. Hence, the architecture should include measures to protect the core from such attacks without having to block other traffic than the DoS traffic.

Prevent that the honeypots are recognized as such
If a system is recognized as a honeypot, it is likely that it is either avoided or systematically fed with "attacks". Therefore, attacks from one attacker to a specific (virtual) destination should always be directed to the same target machine[3].

5.3.2 Requirements for the NoAH Containment Environment

A description of the containment environment used in NoAH can be found in deliverable D1.3. To meet the design goals regarding attack detection and signature generation, it is necessary that it fulfills the following requirements:

Support for host-based attack detection
To achieve attack detection with virtually zero false positives, the containment environment must support attack detection mechanisms that report successful attacks only. Additionally, allowing to pinpoint malicious content in memory and to associate it with the corresponding network traffic allows automated generation of useful network traffic based signatures.

Support for detection of future attacks
The containment environment is required to support attack detection mechanisms for future attacks as well as for current attacks. That is support for arbitrary code execution (i.e. code injection attacks such as buffer overflows), arbitrary execution control (i.e. redirection of control flow), as well as arbitrary function argument (i.e. changing arguments to critical functions) attacks.

Efficient and resource-saving containment
To make attack detection and signature generation as fast as possible, it is necessary that the containment environment is optimized for speed while its resource consumption is kept low.

[3] In fact it is sufficient, if the attacker can be tricked into believing this (e.g. by faking the fingerprint of a system)

Chapter 6

Design Proposal: Attack Detection and Signature Generation in NoAH

6.1 Attack Detection

As already mentioned in Section 5.1, we propose to use host-based attack detection for NoAH. Moreover, we favor a Dynamic Taint Analysis (DTA) based approach since it is a general approach that allows for detecting different kinds of attacks (code injection, control flow hijacking, as well as function argument attacks), and provides detailed information about attacks. Argos [34], the containment system which was developed by the NoAH partner VU, already provides a DTA analysis. The main limitation of Argos is that it is not able to detect attacks that inject some data into an arbitrary memory location (e.g. using a buffer overflow or format string vulnerability) but do never execute attacker-supplied code. This attack type is called non-control-data attack. An example of such an attach is overwriting a user ID or a password in order to get priviledged access instead of non-priviledged. As is shown in [7], such attacks are realisit but according to our knowledge, no such attacks have yet been seen in the wild. Extensions to the attack detection mechanisms should require minimal effort to integrate and should not require any changes to the signature generation mechanisms. In our signature generation design, this holds if the attack detection system for non-control-data attacks allows to pinpoint malicious content in memory and to associate it with the corresponding network traffic as Argos does for control-data attacks. For details about Argos, we refer to D1.3. Other attack types that are not catched with Argos but that can easily be detected using other mechanisms are e.g. password-guessing attacks.

Most of the attack detection approaches identified in 4.1.1 (including DTA) detect an attack [1] only if it's injection was successful. This guarantees a low false

[1] With respect to section 4.1.1, "attack" refers mainly to control flow hijacking attacks

positive rate since alerts are only generated in case the system is really compromised. However, such compromise-based detection systems can only detect those attacks that a sensor is vulnerable to. Thus, depending on the sensor's software configuration, probably a significant amount of attacks is not recognized.

Root Cause Analysis (RCA), proposed by the Leurré.com project [18], is a clustering algorithm which can be used to identify individual attack tools based on observed network traffic. Thus, for RCA it is not required that the system is vulnerable to an attack in order to detect it. However, the focus of such a system is not early-warning but rather attack activity monitoring.

We believe that the deployment of such a system together with host-based attack detection can provide additional information about attacks, and it can also serve as a comparison system for evaluating the false positive/negative rate of Argos. Moreover, while NoAH will deploy high interaction honeypots, the Leurré.com project relies fully on low interaction honeypots which provide less information. Another interesting option that such an activity monitoring system offers, is to use the information provided by RCA for adapting the vulnerability profile of NoAH sensor's according to the ongoing attack activity.

Hence, we propose to use a combination of host- and network-based attack detection algorithms, namely Dynamic Taint Analysis and Root Cause Analysis. The interaction between both systems is depicted in Figure 6.1. Traffic which is directed to the high interaction honeypot will be split and redirected to a root cause analysis engine. This engine monitors the attack activity on the network level, and generates an alert in case a new root cause is detected. For deciding whether a root cause is novel or not, a central database will be contacted.

In a second step the alert correlator matches the DTA and RCA alerts. Alert correlation can be done based on simple timing information, i.e. when was an alert triggered. However, since the alert generation time can differ significantly for RCA and DTA, it would be better to correlate only alerts that were generated by the same network traffic. This is especially important in case a sensor is attacked very frequently. Therefore, it is necessary that the DTA system is able to identify the network traffic that was responsible for a generated alert.

6.2 Signature Generation

In the following sections we present three designs that (partially) meet the design goals of NoAH. We identify implementation challenges and look at the limitations of the different designs. Depending on the experiences from the upcoming implementation and evaluation phase of NoAH, we will select a final design. It is supposed that this design will borrow from all of the three presented designs. Common to all designs is that signature generation is always done on high-interaction honeypots. This is due to the fact that systems who generate signatures only on the basis of network level information suffer from a large false positive problem.

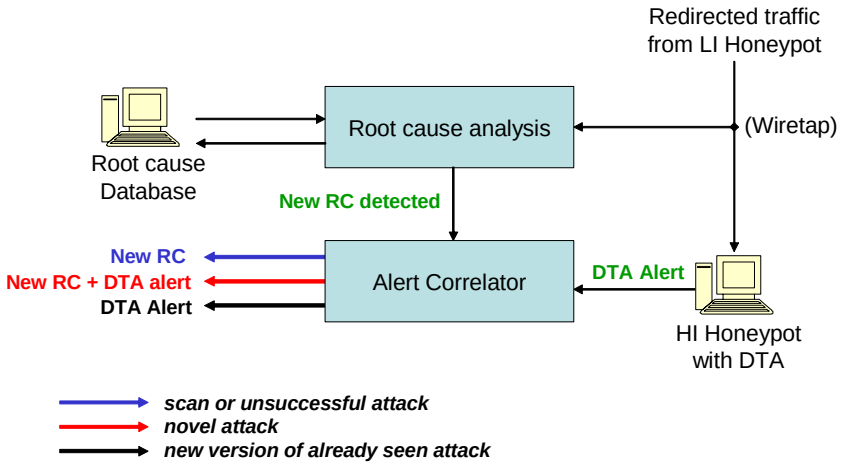


Figure 6.1: Interaction of attack detection systems

6.2.1 Design 1: Single Host

Figure 6.2.1 outlines a design where attack detection and all the necessary steps towards signature generation are done all on one host. Nevertheless, it allows to outsource all of the building blocks but the attack detector. The basic building blocks and their functionality are as follows:

Input Filter: The task of the input filter is to reduce the amount of network traffic that is forwarded to the attack detector. Since this design uses network traffic based signatures for the input filter, the primary objective is to reach zero false positives. This is achieved by filtering only traffic to services that are not installed/provided by the honeypot and by applying only filters that showed no false positives for a certain timespan T in the past. The signatures on which these filters are based are called ”active” signatures. Other signatures are called ”passive” signatures. If a passive signature is matched, the traffic is not blocked but it is verified if the detector reports an alert in connection with this traffic. In case there is no alert, a false positive is reported to the centralized database. If the used sensors (honeypots) are non-homogeneous, the false positive has to be linked to the corresponding configuration identifier. To become an active signature, a passive signature has to meet the following condition: Responsible for at least one true positive and no false positives during timespan T.

Output Supervisor: The output supervisor can be used if a strong guarantee must be given that the honeypot does not attack other systems. This component blocks any connection initiated by the honeypot system and any packet to connectionless services for which no preceding incoming packet has been seen.

Containment: This block is responsible for preventing attacks from corrupting the honeypot. Here, the containment environment is combined with the attack detector. But ideally, the Containment component should not only rely on detecting and blocking attacks to protect the system but should also be able to let an attack do its job once it is detected. Only then we could gain the kind of information about an attack that is relevant to security researchers and security professionals[2]. Furthermore, multiple levels of protection like e.g. protection from control-flow hijacking and protection from malicious file system changes should be introduced. The containment environment proposed by NoAH is based on the sandboxing strategy and runs the honeypot system entirely in QEMU, an x86 emulator. Although this approach allows a maximum of control over the honeypot system, it causes a significant slow-down of the whole system. Moreover, it does not support the afore mentioned mode of operation that allows to let an attack continue after it's detection. A more detailed description of the Containment component in NoAH can be found in deliverable D1.3.

Attack Detector: This component is responsible for detecting attacks on the honeypot on which the component resides. Additionally, whenever an attack is detected, it informs the Signature Generation component and provides information stored at the Alert <> Network Traffic component. More precisely, the attack detector is able to detect control-flow hijacking attacks by marking input from the network interface as tainted and by checking its influence on the control-flow[3]. Furthermore, to determine if operations using tainted data lead to tainting of yet untainted data, they have to be checked versus a "tainting policy". Additionally, by logging operations on tainted data in memory, it is possible to link a tainted memory location to its origin in the network traffic. A detailed description of the used techniques and mechanisms can be found in D1.3. since the attack detector is integrated in the containment environment. Extensions like a detector for password guessing or denial of service attacks can be included if necessary.

Signature Generator: The signature generator generates network traffic based signatures. More precisely, it uses information from the Alert <> Network Traffic component to identify the relevant network traffic. At first, it analyzes if and how the network traffic has been transformed on its way from the network interface to its use in the target application/service. Transformations are e.g. due to channel encryption or due to different character decoding/encoding (transformed) on application level. The former is identified by checking if the receiving service allows encryption [4] or could be done by analyzing the byte frequency distribution of the traffic. The latter is identified by comparing the data in memory with the corre-

[2] E.g. information about what the attack does if it is successful. Does it alter files on the filesystem? Or is there disallowed communication with the outside world afterwards?

[3] If e.g. the program counter points to such data, an alert is triggered

[4] Since we know exactly which services run on our honeypots, this is easy

sponding data in the network traffic. This information allows to decide which of the three signatures generated in the next step are suitable:

- Byte-sequence signature: Consists of the byte sequence that according to the Alert <> Network Traffic component is responsible for triggering the alert. In case the traffic is encrypted, this signature is not very useful. Moreover, its usefulness is very limited if the alert to network traffic mapping pinpoints polymorphic attack payload only. This may e.g. happen if an attacker manages to inject his own code into the execution flow without having to hijack it. But if the mapping pinpoints e.g. parts of the injection vector[5], it could be of some use.
- Byte-frequency distribution signature: Describes the byte-frequency distribution of the relevant part of the network traffic. This signature is able to identify attacks using different encodings from instance to instance.
- Flow signature: Describes the characteristics of the flow that was identified to contain the data responsible for triggering the alert. More precisely it contains the source and destination IP and port as well as the transmitted bytes, the duration and the number of packets of the flow. Indifferent to transformations but only useful in combination with one of the previous two signatures.

Afterwards, a meta signature is created consisting of a unique identifier, the transformation information and the alert information from the Alert <> Network Traffic component. Then it is checked if a similar[6] signature exists. Finally, the Network Based Signatures component is updated accordingly. But what about host based signatures? In this design, host based signatures are included as an option and are based on system call sequence information borrowing from DOME or Paid (disallowed sequences) or rely on (limited) control flow information (variation from allowed paths) made available e.g. by adopting DIRA. One reason reason why these extensions are only optional is that they are actually extra attack detectors with severe impact on performance. A design that is more appropriate for generating host-based signatures is the third design: Multi-Host with Attack-Replay.

Network Based Signatures: Each honeypot system has its own local signature database. This allows to check whether a specific signature has already been generated. If not, it synchronizes its database with the centralized database. In case of inhomogeneous honeypots, the synchronization downloads only signatures that were generated by systems using the same software (and eventually hardware) configuration. Additionally, information about the signatures state like its creation time, the number of reported false positives and if it is active or passive, are stored locally and globally.

Host Based Signatures: This component does the same as the Network Based

[5] like the data used to overwrite the return address of a function via a buffer overflow

[6] It remains to be defined in which case meta signatures are similar

Signatures component but for host based signatures.

Alert <> Network Traffic: This key component allows to map alerts to the network traffic that contains the corresponding attack. More precisely, it allows either to identify the location of the payload and/or the location of the content that overwrote a return address in it. Therefore it stores the tainted data (and data nearby) responsible for triggering the alert as well as its position in the corresponding network traffic. Localization has to be as precise as possible and in case of multi-stage attacks, references to multiple network traffic data should be returned. The therefore required information is provided by the Attack Detector component.

Centralized Database: This database stores the network traffic based signatures. Whenever a system generates a new signature, it checks if the signature is already stored. The design of the centralized database depends heavily on the architecture of the network of affined honeypots. More precisely, an network of inhomogeneous participants is far more complex to handle. In case there exists a core network of only a few different high-interaction honeypot systems, the database should contain at least the signatures along with an identifier of the type of the honeypot system who created it. Only then, installing signatures for attacks on vulnerabilities that are not present in certain honeypot systems can be avoided. Additionally it allows to identify if an attack has an impact on multiple or only one specific configuration.

6.2.2 Design 2: Multi-Host with Focus on Network Traffic Analysis

A way to reduce the false positive rate when using network-level signatures is to implement protocol- and content-awareness. Nowadays, this approach is widely used in commercial applications and is an issue in several research projects (see e.g. [26]). Even some open-source IPS allow for protocol awareness since they support regular expressions [33]. Protocol- and content-awareness can provide a lot of useful information (e.g. protocol states, field lengths, and field types, expected content characteristics,...) for generating more accurate and meaningful signatures. The ultimate goal of this approach is to identify the injection vectors and to create signatures for them. Signatures describing injection vectors are interesting because polymorphism in injection vectors is usually not an issue. Exploiting a specific vulnerability requires to follow a strict path towards the vulnerability with none or a few possible deviations. But even without signatures for injection vectors, protocol awareness should be able to reduce the false positive rate significantly. This is due to the fact that identifying the state and the type of message in which attack relevant traffic is delivered, reduces the amount of traffic versus which e.g. simple byte-sequence signatures are matched. Figure 6.2.1 outlines a design that extends the first design with the Connection State Tracker component that implements protocol- and content-awareness. Because of performance and se-

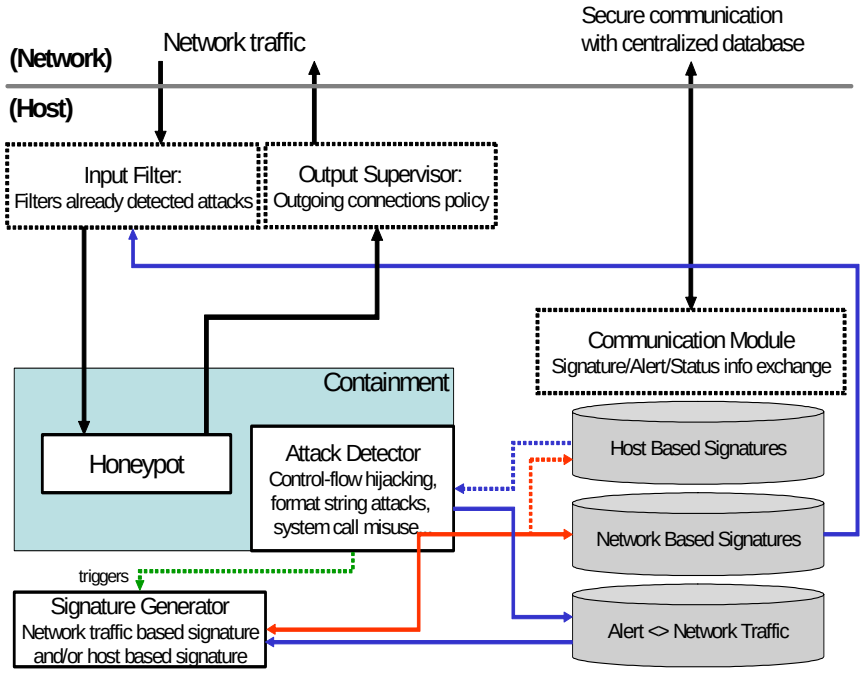


Figure 6.2: Design 1: Single host

curity considerations, this component resides on an additional host.

Connection State Tracker: The Connection State Tracker is a component that sniffs the network traffic to and from a honeypot. In this context, a ”connection” is identified by its source IP/source port pair on the network layer and, if the connection is bi-directional, its destination IP/destination port pair. Nevertheless, on higher layers, multiple of these connections can be associated with each other to form the basis for the state description of multi-connection protocols like e.g. FTP. For each of those connections, the state is updated and logged on the different protocol layers whenever a packet is seen. State information for IP includes e.g.:

- Protocol (Encapsulated Security Payload(ESP),Authentication Header(AH),IPv6,...)
- Time passed since last packet arrival
- Packet is valid/invalid

State information for TCP connections include e.g.:

- Status: SYN/ESTABLISHED/RESET/CLOSED
- Fragmented (size)/ not fragmented (size)
- Packet is valid/invalid
- (others)

To do this for an arbitrary system would be hard since we do not know the set of used protocols and would therefore have to implement full protocol knowledge for all of the supported protocols. By contrast, in a honeypot setting the set of used

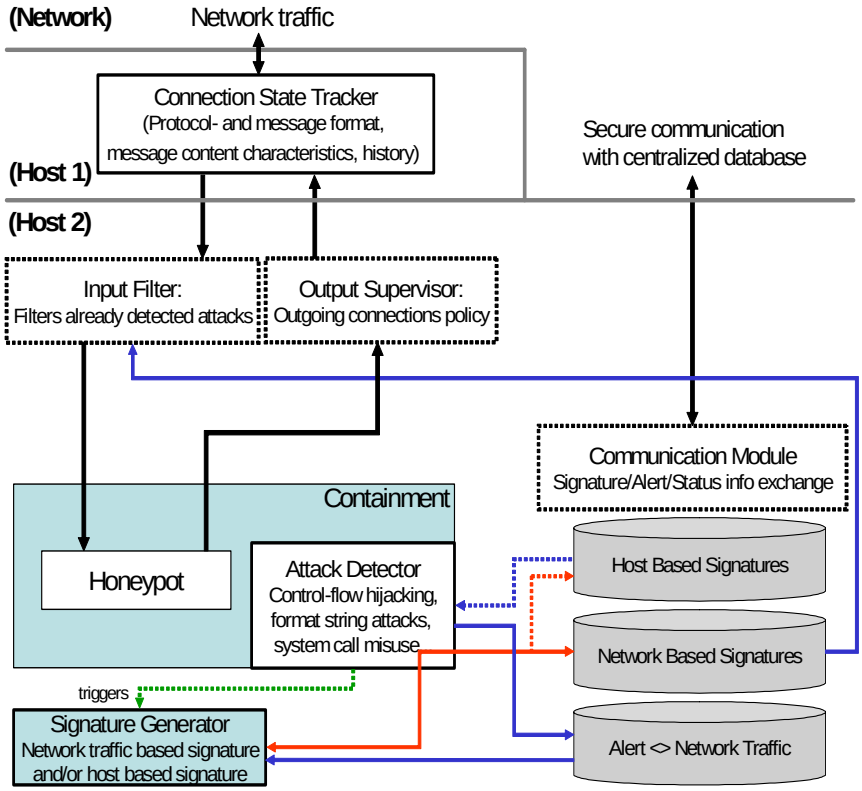


Figure 6.3: Design 2: Multi-Host with focus on network traffic analysis

protocols is known and implementing full protocol knowledge for these is possible. Another feature of the Connection State Tracker is its ability to provide information on content characteristics for data fields with respect to a specific position in the data field. This information could e.g. include:

- Deviation from normal byte frequency distribution
- Regular expression describing the expected content format

The Connection State Tracker component provides information to the Signature Generation component on demand. Whenever signature generation is triggered, the Signature Generation component sends the packet(s) responsible for triggering the alert along with a pointer to the relevant data (as reported by the attack detector) to the Connection State Tracker and gets the corresponding state(s). Optionally, information about the content characteristics corresponding to the pinpointed position are provided. Therewith, a signature that is linked to a state and that takes into account the content characteristics can be generated. Furthermore, any signature/rule format used by statefull intrusion detection systems (e.g. Bro [33]) is a potential candidate for the format of the generated signatures.

6.2.3 Design 3: Multi-Host with Attack-Replay

The idea of this design is to do a detailed control-flow analysis in order to identify the vulnerable point in the attacked service or application. A feasible way to prevent future exploitation of this vulnerability is to modify the application's binary so that the vulnerability is no longer exploitable. The generated signatures describe the necessary modifications in a way that they can be incorporated using binary rewriting. That this is a feasible and promising approach is supported by the results from the Vigilante project [7]. We mapped this idea to the design showed in figure 6.2.3. Compared to the Single Host design, there are three new components:

- Sniffer
- Replayer
- Log

and two additional hosts:

- Host 3: Mirror Honeypot
- Host 2: Replay System

The reasons why it is hard if not impossible to implement the design idea on a single host are the following:

- Performance: As already mentioned in the Single Host design, the proposed attack detection and containment environment requires a significant share of the available resources. It is therefore not feasible to do rather complex and time-consuming things like a just-in-time control-flow analysis on the same machine.
- Identifiability: If we have an always-on just-in-time control-flow analysis running on the honeypot, it is very likely that due to the additional slowdown and introduced delays, it is easier for an attacker to detect that the system is actually a honeypot.

By leaving the network based signature generation task to the honeypot that is connected to the outside world and by assigning the task of generating control-flow analysis based signatures to another host, we can mitigate the performance and identifiability problem. But splitting the tasks to two different hosts implies the introduction of a device that is able to store and replay communication (in our design Host 2, the Replay System) with the Honeypot component on Host 1 to an identical Honeypot component on another host (Host 3 in our design). An additional advantage of such a design is that because a detailed control-flow analysis is performed on demand only, we can reduce the overall system load significantly. The steps conducted in case the Attack Detector on Host 1 detects an attack are the following:

1. The Attack Detector on host 1 triggers the signature generation process on the same host.

2. It informs the Replayer component on Host 2 about the attack and the network traffic involved.

[7] See section 3.3.7 and publications [13] and [14]

3. The Replayer component selects the relevant traffic from the Log database and replays the attack to host 3, the Mirror Honeypot.

4. Host 3 performs a detailed control-flow analysis and generates the host-based signature

The newly introduced building blocks have the following functionality: **Sniffer:** It logs all the traffic to and from the honeypot system. Optionally, this could include timing information to support accurate replays.

Log: A database that contains the traffic collected by the Sniffer component.

Replayer: This component replays logged communication on demand. This may be very easy or very hard depending on the presence of ”dynamic” fields containing e.g. time or target machine dependent (IP info,...) information that needs to be replaced accordingly. As it is e.g. shown in [15] doing this without protocol specific knowledge can be very challenging. This is why combining the previous design with this one could be promising. The protocol knowledge would be very useful for the Replay component.

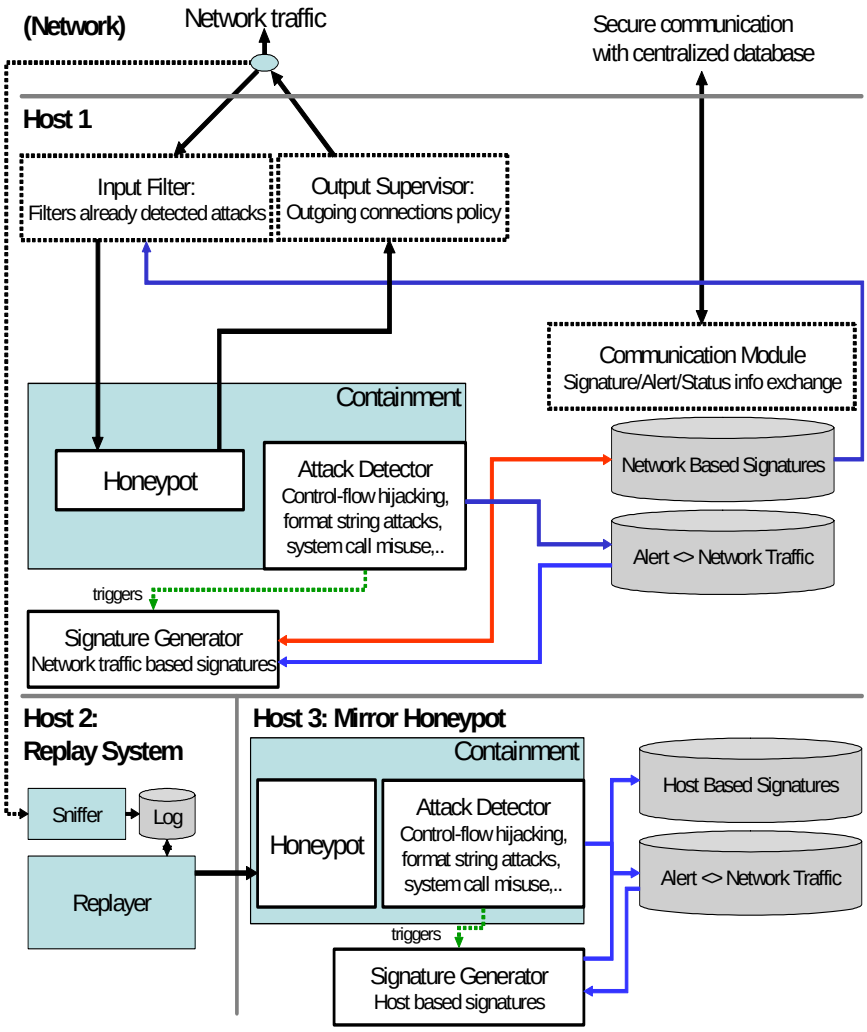


Figure 6.4: Design 3: Multi-host with attack-replay

Chapter 7

Conclusions

In this deliverable we have analysed drawbacks and advatages of existing attack detection and signature generation algorithms. Moreover, we have developed a novel classification scheme for automated signature generation mechanism. Based on our analysis we have proposed several designs for automated signature generation within NoAH with regards to the goal of NoAH: The detection and containment of zero-day cyberattacks.

In our review section, we classified the approaches according to the attack detection mechanism they rely on. We found that earlier approaches did not apply any attack detection prior to signature generation, but generated signatures for all traffic that entered a honeypot. The second, more sophisticated, generation of approaches used network-level attack detection prior to signature generation, whereas the most recent approaches rely on host-based attack detection mechanisms.

Our analysis of drawbacks and benefits of the individual approaches revealed that host-based attack detection methods such as memory tainting are most reliable and provide the most detail about an attack since they observe the actual exploit taking place. On the other hand, these approaches are very resource-intensive and software-specific. Consequently, using such an approach requires protection against Denial-of-Service attacks, and deployment of a variety of different software profiles.

Bibliography

[1] D. E. Ana Nora Sovarel and N. Paul. Where's the feeb? the effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium*, pages 145 – 160, Baltimore, US, August 2005.

[2] Anonymous. Once upon a free(). *Phrack*, 11(57):?, August 2001.

[3] K. Biba. Integrity considerations for secure computer systems. Mitre, Technical Report MTR-3153, MITRE, Redfort MA, April 1977.

[4] M. Bossardt. *Composition and Deployment of Services in Heterogeneous Programmable Networks*. Dissertation no. 16450, ETH Zurich, Jan. 2006.

[5] F. Castaneda, E. C. Sezer, and J. Xu. Worm vs. worm: preliminary study of an active counter-attack mechanism. In *WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 83–93, New York, NY, USA, 2004. ACM Press.

[6] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for structured peer-to-peer overlay networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, December 2002.

[7] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Usenix Security Symposium*, pages 177 – 192. USENIX, August 2005.

[8] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*, pages 32–46, Oakland, CA, USA, May 2005. ACM Press.

[9] L. chung Lam and T. cker Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Proceedings of the 7th International Symposium in Recent Advances in Intrusion Detection (RAID)*, volume 3224 of *Lecture Notes in Computer Science*, pages 1–20, Sophia Antipolis, September 2004. Springer-Verlag GmbH.

[10] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Can we contain internet worms? In *Proceedings of HotNets III*, San Diego, California, USA, November 2004.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 133–147, New York, NY, USA, 2005. ACM Press.

[12] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Usenix Security Symposium*, pages 63–78, January 1998.

[13] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO-37 2004)*, pages 221–232, Portland, USA, December 2004. IEEE Computer Society.

[14] J. R. Crandall, S. F. Wu, and F. T. Chong. Experiences using minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *Proceedings of DIMVA'05*, pages 32–50, 2005.

[15] W. Cui, V. Paxson, N. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the 13th Network and Distributed System Security Symposium*, 2006.

[16] D. Dagon, X. Qin, G. Gu, W. Lee, J. B. Grizzard, J. G. Levine, and H. L. Owen. Honeystat: Local worm detection using honeypots. In *Proceedings of RAID'04*, volume 3224 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2004.

[17] Data Rescue. *Data Rescue. IDA Pro Disassembler.*

[18] M. D. F. Pouget. Honeypot-based forensics. In *Proceedings of the Asia Pacific Information technology Security Conference*, Brisbane, Australia, May 2004.

[19] H. W. Hethcote. The mathematics of infectious diseases. *SIAM Rev.*, 42(4):599–653, 2000.

[20] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley Professional, February 2004.

[21] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, USA, July 1999.

[22] M. Jordan. Dealing with metamorphism. *Virus Bulletin*, 1(10):4–6, October 2002. The original version can be found at the URL specified in the 'pdf' field.

[23] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the USENIX Security Symposium*, pages 271–286, 2004.

[24] 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)*, Boston, November 2003.

[25] K. Lawton. Bochs, the cross platform ia-32 emulator, http://bochs.sourceforge.net/. Sourceforge Project, 2006.

[26] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

[27] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[28] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings of the 22th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'03)*, San Francisco, USA, April 2003.

[29] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2):?, 2003.

[30] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 226–241, Washington, DC, USA, 2005. IEEE Computer Society.

[31] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.

[32] PaX Project. *The Pax Project Team. http://pax.grsecurity.net/.*

[33] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463, 1999.

[34] G. Portokalidis, A. Slowinska, and H. Bos. Argos: An emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[35] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, August 2003.

[36] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid Malcode*, pages 76–82, New York, NY, USA, 2003. ACM Press.

[37] L. Ruf. *Network Services on Service Extensible Routers*. Dissertation no. 16323, ETH Zurich, Nov. 2005.

[38] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 298–307, 2004.

[39] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *OSDI*, pages 45–60, 2004.

[40] C. T. Smirnov A. Dira: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of NDSS05: Network and Distributed System Security Symposium Confrence Proceedings*, San Diego, California, February 2005. Internet Society.

[41] snort.org. *Snort, an open source network intrusion detection system.*

[42] Y. Tang and S. Chen. Defending against internet worms: A signature-based approach. In *Proceedings of IEEE INFOCOM'05*, March 2005.

[43] T. G. team. Gcc, the gnu compiler collection ,http://gcc.gnu.org/. Online, 2006.

[44] N. T.J.Bailey. *The Mathematical Theory of Infectious Diseases and its Applications*. Griffin, 2nd edition, 1975.

[45] A. Wagner, T. Dübendorfer, B. Plattner, and R. Hiestand. Experiences with worm propagation simulations. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode*, pages 34–41, New York, NY, USA, 2003. ACM Press.

[46] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

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

[48] S. S. Wang K. Anomalous payload-based network intrusion detection. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection*, Sophia Antipolis, France, September 2004.

[49] D. C. D. Wei Xu and R. Sekar. An efficient and backwardscompatible transformation to ensure memory safety of c programs. In *Proceedings of ACM SIGSOFT/FSE*, Newport Beach, USA, November 2004.

[50] V. Yegneswaran, J. T. Giffin, and a. S. J. Paul Barford. An architecture for generating semantic-aware signatures. In *Proceedings of the 14th USENIX Security Symposium*, pages 97–112, August 2005.

[51] C. C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 138–147, New York, NY, USA, 2002. ACM Press.